GQL ESSENTIALS

Dominik Tomaszuk



Acknowledgments

Writing this book has been a long journey, and I could not have completed it without the patience, love, and understanding of my family.

To my wife, Justyna, thank you for your unwavering support and for being so understanding during the many moments when writing took my time and attention.

To my daughter, Gabriela, and my son, Filip, thank you for your warmth, your smiles, and for reminding me every day what truly matters.

I am also deeply grateful to Artur Korniłowicz for his valuable substantive feedback and insightful suggestions, which helped shape this book and made it significantly better.

With love and gratitude,
Dominik Tomaszuk

Contents

Book Overview 1

I Introduction 2

1. History 3

2. Property Graph Databases 6

2.1. Use Cases and Real-World Applications . . . 8

2.2. Limitations of Property Graph Databases . . . 10

3. Positioning 11

3.1. Scholarly Analysis . . . 11

3.2. Relational Databases and SQL . . . 12

3.3. RDF and SPARQL . . . 15

3.4. Tree-based Languages . . . 16

3.5. Key-value and Document Databases . . . 18

4. GQL by Example 20

4.1. Visual Representation . . . 20

4.2. Data Definition Language . . . 22

4.3. Data Manipulation Language . . . 24

4.4. Data Query Language . . . 26

II Core Model and Patterns 33

5. A Conceptual Glossary of GQL 34

5.1. Foundations . . . 34

5.2. Graphs . . . 36

5.3. Context . . . 38

5.4. Values . . . 39

5.5. Expressions . . . 41

5.6. Schemas . . . 43

5.7. Paths . . . 44

5.8. Path Search . . . 46

5.9. Path Expressions . . . 46

5.10. Matching . . . 48

5.11. Algebra . . . 51

5.12. Queries . . . 52

5.13. Result Shaping . . . 54

5.14. Updates . . . 55

5.15. Constraints . . . 57

5.16. Effects . . . 59

5.17. Routines . . . 60

5.18. Transactions . . . 61

5.19. Authorization . . . 62

5.20. Diagnostics . . . 64

6. Data Model **67**
6.1. Universes and Notation . . . 67
6.2. Property-Graph Structure . . . 68
6.3. Values and Types . . . 70
6.4. Basics of Paths as Data . . . 71
6.5. Lists and Structural Access . . . 72
7. Patterns and Matching **74**
7.1. Graph Patterns . . . 74
7.2. Pattern Syntax . . . 78
7.3. Variable Exposure and Safety . . . 80
7.4. Matching Semantics . . . 82
7.5. Binding Tables . . . 86
7.6. `MATCH` Statements . . . 87
7.7. Key Scopes and Finiteness . . . 92

III Expressions and Algebra **95**
8. Expressions and Evaluation **96**
8.1. Expression Domains . . . 96
8.2. Typing and Evaluation . . . 96
8.3. Truth Values and `NULL` . . . 97
9. Algebra of Binding Tables **101**
9.1. Selection and Projection . . . 101
9.2. Joins and `OPTIONAL` (Left-Outer Join) . . . 102
9.3. `FOR` and List Unnesting . . . 104
9.4. Subqueries and Scope . . . 104
9.5. Whole-Query Denotation . . . 106
9.6. Composite Queries and Set Operators . . . 108

IV Paths and Regular Path Expressions **113**
10. Path Semantics **114**
10.1. Modes: Walk, Trail, Simple, Acyclic . . . 114
10.2. Finiteness and Selection . . . 115
10.3. Path Accessors and Results . . . 115
11. Regular Path Expressions **120**
11.1. Syntax and Denotation . . . 120
11.2. Shortest-Path Semantics . . . 127

V Aggregation and Ordering **130**
12. Grouping and Aggregation **131**
12.1. `GROUP BY` and Aggregates . . . 131
12.2. `HAVING` and `DISTINCT` . . . 135
12.3. Primitive Result Statements . . . 136
13. Ordering and Pagination **139**
13.1. `ORDER BY` and Ordered Tables . . . 139

13.2. Determinism and Peer Ordering . . . 140
13.3. `LIMIT`/`OFFSET` (Slice) . . . 140
13.4. Determinism Guarantees . . . 142

VI Data Modification 143
14. Data Modification 144
14.1. `INSERT` and `CREATE` . . . 146
14.2. `DELETE` . . . 149
14.3. Read-then-Update Composition . . . 151
14.4. Write-Set Coalescing and Conflicts . . . 152
14.5. Update Model for Property Graphs . . . 161
14.6. Dynamic Constraint Validation . . . 161

VII Types, Schemas, and Constraints 163
15. Types and Schemas 164
15.1. Catalog and Name Universes . . . 164
15.2. Property Types . . . 164
15.3. Node and Edge Types . . . 166
15.4. Graph Types . . . 168
15.5. Graph Entries and Constraining Types . . . 169
15.6. DDL Semantics . . . 171
15.7. Schema Versioning . . . 172
16. Constraints 174
16.1. Keys and Uniqueness . . . 174
16.2. Constraint Enforcement and Effects . . . 177
16.3. Effect System and Write-Purity . . . 178

VIII Catalogs, Sessions, and Procedures 184
17. Catalog, Names and Session State 185
17.1. Named Objects and Namespaces . . . 185
17.2. Qualified References and `USE` . . . 187
17.3. Session State and Sites . . . 188
18. Procedures 189
18.1. Procedure Signatures and Body Typing . . . 189
18.2. Calls and Scoping . . . 190

IX Transactions and Concurrency 194
19. Transaction Model 195
19.1. Demarcation and Visibility . . . 195
19.2. Serializability and Isolation . . . 198
19.3. Errors and Aborts . . . 199

X Security and Diagnostics 201
20. Authorization 202
20.1. Roles and Privileges . . . 202
20.2. Privilege Obligations and Enforcement . . . 203
20.3. Authorized Object Access . . . 205
20.4. Write Checks . . . 205
21. Diagnostics 208
21.1. Errors and Exceptions . . . 208
21.2. Status Codes . . . 209
21.3. Control Flow . . . 211

XI Future Directions 213
22. Future Enhancements 214
22.1. Compositional Enhancements . . . 214
22.2. Schemas in GQL . . . 216
22.3. Extending the Data Model and Integrating AI . . . 217
23. Research Directions 221
23.1. Expressiveness and Complexity . . . 221
23.2. Query Processing and Optimization . . . 222
23.3. Design Decisions and Alternatives . . . 222
23.4. Updates and Transaction Processing . . . 223
23.5. Graph-to-Graph Queries . . . 223
23.6. Interoperability and Data Integration . . . 223
23.7. Schema and Metadata Management . . . 224

Bibliography 226

Acronyms 234

Symbols and Notations 237

Index 249

Book Overview

This book combines descriptions of the normative requirements of the Graph Query Language (GQL) as standardized in ISO/IEC 39075, a synthesis of relevant literature, and an explanatory mathematical reconstruction and model developed for this book; the GQL standard itself remains the normative source. Part I combines cited historical literature with an example-based exposition of the language, Parts II to X primarily develop the explanatory reconstruction of the standard's data model and requirements, and Part XI synthesizes cited extension proposals and research literature. The remainder of this book is organized as follows:

- **Part I – Introduction** sketches the motivation, background, and running example.
- **Part II – Core Model and Patterns** fixes the property-graph data model and basic patterns.
- **Part III – Expressions and Algebra** introduces expressions and the algebra of binding tables.
- **Part IV – Paths and Regular Path Expressions** explains paths, path modes, and regular path expressions.
- **Part V – Aggregation and Ordering** covers grouping, aggregation, ordering, and pagination.
- **Part VI – Data Modification** defines inserts, deletes, and updates over property graphs.
- **Part VII – Types, Schemas, and Constraints** presents typing, DDL, and integrity constraints.
- **Part VIII – Catalogs, Sessions, and Procedures** describes catalog objects, sessions, and procedures.
- **Part IX – Transactions and Concurrency** develops the transaction model and isolation guarantees.
- **Part X – Security and Diagnostics** treats authorization, errors, and diagnostic reporting.
- **Part XI – Future Directions** outlines possible extensions and open research directions.

Part I

Introduction

1 History

The history of graph databases can be traced back to foundational work in network models and the standardization of database languages. A notable early system was the Integrated Data Store (IDS), designed by Charles Bachman, which laid the groundwork for the network model of databases. This model was further formalized by the Conference on Data System Languages (CODASYL) through its Data Base Task Group (DBTG). In the late 1960s, CODASYL created DBTG, which issued influential reports in 1969 and 1971, proposing a standard sublanguage for database applications to be embedded in COBOL (North 1968).

In a parallel development, IBM introduced the Information Management System (IMS) in 1968, employing a hierarchical data model. IMS structured data into record types, each comprising a set of named fields with defined data types, where a subset of fields served as keys, uniquely identifying each record instance. This structure imposed a strict tree hierarchy, requiring each record (except the root) to have a single parent. Such a requirement created challenges, necessitating data duplication and limited support for certain cases. For instance, part information had to be repeated across suppliers, increasing the risk of inconsistency, while certain relationships, such as a part without suppliers, were not supported. These limitations highlighted the rigid nature of hierarchical models, spurring the exploration of more flexible structures (Klein et al. 2011).

CODASYL, recognizing the limitations of a strictly hierarchical approach, introduced a network data model in its 1971 and 1973 reports. This model abandoned the tree structure of IMS, allowing each record instance to have multiple parent records, thus forming a more interconnected network rather than a single-rooted hierarchy. In this model, connections were represented by "sets", which defined 1-to-n relationships between an owner record type and its associated child record types. While referred to as sets, these structures were closer to directed relationships, where each owner could relate to multiple instances of a child record type. This network approach allowed for greater data flexibility, enabling a record to have multiple parents and eliminating the limitations imposed by hierarchical constraints (CODASYL. Systems Committee 1971).

Simultaneously, the American National Standards Institute (ANSI) played a pivotal role in creating a standardized framework for database languages. In 1978, ANSI formed the committee known as ANSI X3H2, now called INCITS DM32, or "H2". This committee focused on developing a "Network Data Language" (NDL) based on the 1971 CODASYL DBTG report, aiming to establish a competitive and compatible market for database products. The work of ANSI and CODASYL significantly influenced the evolution of

database technologies, setting the stage for more advanced data models that eventually led to the development of graph databases (Gorman 1978).

Amid these advancements, Conceptual Graphs (CGs) emerged as another significant innovation in graph-based knowledge representation. Proposed by John F. Sowa in the mid-1970s (Sowa 1976), CGs were formally introduced in his seminal book published in 1984 (Sowa 1984). Conceptual Graphs are designed to represent knowledge in a way that is intuitively understandable to humans while being efficiently processable by computers, acting as a bridge between natural language and machine languages. This form of representation plays a crucial role in the conceptual modeling of domains and enhances the semantic processing capabilities of information systems.

Parallel to the developments in traditional database models, Tim Berners-Lee envisioned the Semantic Web in the mid-1990s, shortly after initiating the development of the World Wide Web (Berners-Lee 1998). The concept was formalized by establishing the World Wide Web Consortium (W3C) and gained significant momentum with the publication of a vision in a 2001 Scientific American article (Hendler et al. 2001). Despite the ambitious visions, the Semantic Web had only been partially realized, with a limited portion of the web domains incorporating Semantic Web markup. SPARQL[1] (Seaborne and Prud'hommeaux 2008), introduced in 2004 and refreshed in 2008, emerged as a dedicated query language that facilitated the retrieval of data stored in the Resource Description Framework (RDF) (Carroll and Klyne 2004), offering capabilities such as "property paths", which resemble features of regular path queries in graph databases.

The late 20th century saw significant research into graph query languages, with the introduction of regular path queries (RPQs) (Cruz et al. 1987) in 1987 and their extension into conjunctive regular path queries (CRPQs) (Consens and Mendelzon 1990) by 1990. These developments underscored the growing interest in efficiently querying complex graph structures beyond the limitations of Structured Query Language (SQL) and RDF for graph-related problems.

The Neo4j property graph database was launched in 2007 (Angles 2012; Hunger 2012), and in 2011 its Cypher query language was introduced (Francis et al. 2018; Lindaaker 2018), to allow declarative querying and manipulation of graph data. The property graph data model, as distinct from RDF, places records of properties (very similar to SQL rows) on both nodes and relationships, and is particularly aligned with the needs of modern applications such as knowledge graphs for artificial intelligence (AI) and data mesh architectures. The evolving standards continued with the recent introductions of SQL Property Graph Queries (SQL/PGQ) in 2023 (International Organization for Standardization 2023) and GQL in 2024 (ISO/IEC 2024), reflecting a trend toward formal standardization in graph query languages.

[1]SPARQL is a recursive acronym for SPARQL Protocol and RDF Query Language.

Standardization has been crucial in the database industry, ensuring interoperability and fostering a competitive market environment. As graph databases have grown in importance and are driven by AI and big data applications, developing standardized query languages such as GQL is essential. This facilitates technological innovation and supports scalable, efficient, and less complex data architectures, enabling businesses to leverage graph data effectively.

From the network models and NDL of CODASYL and ANSI's early efforts to the sophisticated graph query languages like SPARQL and Cypher, the journey of graph databases reflects a broader evolution toward more structured, interoperable, and efficient data management practices. As these technologies continue to develop, they promise to underpin next-generation data infrastructures and applications.

Chapter 1 — Takeaways

- **Purpose:** Recounted the main line from early network data models to modern graph databases, clarifying why standardization pressure eventually produced GQL.
- **Introduced:** The navigational lineage of graph-like database systems; the rise of declarative database standardization; the later split into RDF-style graphs and property-graph practice.
- **Key distinctions:** Navigational access versus declarative querying; RDF-style triple-centric graphs versus property graphs with labeled, attributed edges.
- **Consequences:** Motivated interoperability as a first-order requirement (portable queries and predictable behavior); justified why a formal semantics mattered beyond vendor documentation; established the historical context for GQL's design constraints.
- **Bridge:** Transitioned from history to the concrete capabilities and limitations of today's property graph database systems.

2 Property Graph Databases

Property graph databases are commonly situated within the broader NoSQL landscape, although NoSQL denotes a diverse family of systems rather than a single formal data model (Sadalage and Fowler 2012). They are designed for situations in which connections between data items are just as important as the items themselves. Instead of thinking in terms of tables, rows, and foreign keys, they invite us to view data as a graph: a network of entities and the relationships that link them. In contrast to traditional relational databases, which are table-oriented, and document stores, which are collection-oriented, property graph databases are relationship-oriented. Their central idea can be summarized as follows: if the problem you are solving is fundamentally about connections, then your database model should reflect that fact.

The core abstraction of a property graph database is a labeled, attributed multigraph that may contain directed and undirected edges. In practical terms, this means that the database is built around nodes and edges that are both richly described and explicitly connected. Nodes, sometimes called vertices, represent entities in the domain: a person, a company, a product, a city, a bank account, a research paper, a device, or any other identifiable object. Each node can carry zero or more labels that classify it, such as `Person`, `Customer`, `Company`, `Movie`, `Paper`, or `Device`. Labels group nodes into sets and are used heavily in queries and indexing. It is common, for example, for a single node to have both Person and Customer labels, which captures polymorphism in the domain in a very natural way.

Edges, often referred to as relationships, connect pairs of nodes and may be directed or undirected. They express facts such as that one person is friends with another, that a customer has purchased a product, that an author has written a paper, or that a server relies on a particular service. Every edge has two endpoints and a directionality; a directed edge additionally distinguishes its source from its destination. Each edge has an edge label set; its labels classify the edge, for example, `FRIENDS_WITH`, `PURCHASED`, `LIVES_IN`, `WORKS_FOR`, `WROTE`, or `CITES`. An edge label does not by itself constitute an edge type. In GQL, the minimum and maximum cardinalities of an edge label set are implementation-defined, so an implementation may limit such a set to at most one label. Property graph databases are multigraphs, so multiple edges with possibly different label sets can connect the same pair of nodes. One person might both `WORKS_FOR` and `INVESTED_IN` the same company, or two papers might both `CITES` and `EXTEND` each other through different relationships that are relevant for different analyses. Crucially, edges are first-class citizens: they are stored natively, have their own identity, and can bear data just like nodes.

The term "property graph" refers to the fact that both nodes and edges carry properties, which are key-value pairs that describe the attributes of those elements. A `Person` node might have properties such as `name`, `age`, and `email`, while a `Product` node might have properties such as `name`, `category`, and `price`. Property values are typically typed: strings, numbers, booleans, temporal values, and often also collections such as lists and maps. A list property might store several tags, and a map property might capture structured metadata. A key differentiator from many other models is that edges can also possess properties. For instance, a `PURCHASED` relationship might record the timestamp of the transaction, the quantity, and the price paid; a `FRIENDS_WITH` relationship might store the date the friendship started or a measure of connection strength; a `WROTE` edge could record the contribution percentage of a particular coauthor. This ability to attach rich metadata directly to relationships allows the graph to capture not only that two entities are connected, but also how, when, and to what degree they are connected.

Node and edge labels play a central role in how property graphs are queried and optimized. Node labels, such as `Person` or `Movie`, and edge labels, such as `ACTED_IN` or `DIRECTED`, allow the system to index and constrain subsets of the graph efficiently. Queries can start from all nodes labeled `Customer`, or follow only edges carrying the label `LIVES_IN`, or enforce constraints so that every node with label `Account` has an `accountNumber` property and appropriate indexes on it. Multi-label nodes enable the modeling of heterogeneous or overlapping categories without requiring schema modifications. Similarly, labeled edges make the domain's structure clear and expressive at the database level.

Sometimes, beyond individual nodes and edges, property graph databases treat paths as first-class concepts. A path is a sequence of alternating nodes and edges that represents a traversable route through the graph. Many queries are really questions about such paths: whether any path exists between two entities, what the shortest path is, whether there is a path whose labels match a certain pattern, or how many distinct paths of a particular shape can be found. Property graph systems typically offer built-in support for reachability queries, shortest path computations, and regular path queries that constrain the allowed sequence of edge-label tests. In some systems, entire paths, not just their endpoints, can be returned and manipulated as query results, which is vital for tasks such as explaining recommendations or tracing the provenance of data.

Interacting with a property graph database typically involves specialized graph query languages designed for pattern matching and path navigation. Cypher is the most widely known example, originally created for Neo4j and one of the principal precursors of the standardized GQL (International Organization for Standardization 2024) that is the subject of this book. Cypher uses an ASCII-art-inspired syntax, in which nodes appear in parentheses and relationships in brackets, with arrows, so that queries visually resemble the structures they match. For instance, a pattern resembling

`(p:Person)-[:PURCHASED]->(pr:Product)` expresses a person purchasing a product. Queries specify not only which patterns to match, but also which properties to filter on and how to shape the returned results as nodes, edges, paths, or aggregates. Other systems offer similar capabilities, but the common theme is that the primary unit of querying is not a table but a graph pattern. Many property graph databases offer ACID transactions, allowing graph updates and queries to participate in familiar transactional workflows with guarantees of atomicity, consistency, isolation, and durability.

From a performance perspective, property graph databases are engineered to excel at graph-shaped workloads. They commonly maintain indexes over nodes to provide fast entry points for traversals, and sometimes over relationships as well. Once a set of starting nodes has been found, queries typically proceed by following adjacency lists, that is, by exploring the edges attached to those nodes. Storage layouts are optimized so that related nodes and edges are stored physically close together, which reduces the cost of following relationships. The result is that the cost of a query tends to be proportional to the amount of the graph that is actually traversed, rather than to the total size of the database. Deep traversals, which might require several joins in an RDBMS (Relational Database Management System), become straightforward pattern matches that remain efficient as the graph grows. Many systems also integrate or expose graph algorithms such as centrality measures, community detection, and various forms of path finding, making complex analytics on connectivity a routine part of querying.

Often, flexibility in schema is another hallmark of property graph databases. The model is schema-optional: although constraints and indexes can and often should be defined, the system does not require a rigid, global schema in the same way that a relational database does. New node and edge labels and properties can be introduced incrementally without disrupting existing data; where a constraining graph type is used, its node and edge types must be changed accordingly. This makes property graphs particularly attractive in evolving domains where data structures are not fully known in advance or where they change rapidly under business pressure. Developers can start with a simple model and refine it over time, adjusting new relationships and attributes as their understanding of the system deepens.

2.1 Use Cases and Real-World Applications

The expressive power and performance characteristics of property graph databases make them especially well-suited to a range of application domains where connected data is central. Social networks are perhaps the most intuitive example, with users linked by friendships, follows, group memberships, likes, and content interactions. Features such as “people you may know” or “mutual friends” become simple graph traversals that look for certain patterns among paths of length two or three. Recommendation engines in e-commerce and media services utilize paths that connect users, products, ratings,

and categories to suggest items that are similar or complementary. Fraud detection in banking and financial services benefits significantly from viewing accounts, devices, transactions, and merchants as a graph and identifying suspicious substructures, such as rings of accounts, unusual transaction paths, or clusters that match known fraud patterns.

Knowledge graphs form another major category of applications. Large technology companies use massive property graphs to connect facts about people, organizations, places, events, and abstract concepts, providing the backbone for search engines and digital assistants. In an enterprise context, knowledge graphs integrate data from multiple systems, enabling analysts to navigate from a customer to contracts, interactions, products, support tickets, and payments within a single, coherent structure. Master data management similarly utilizes property graphs to link and reconcile multiple representations of the same real-world entity across different systems, thereby building a unified view of customers, products, or locations. Network and IT operations benefit from modeling infrastructure as a graph of devices, services, dependencies, and communication flows, enabling impact analysis when a component fails and supporting change planning. Identity and access management can be expressed as a graph of users, groups, roles, resources, and permissions, which makes it much easier to answer questions such as who effectively has access to which system through what chain of assignments.

Beyond these now classical domains, there are many specialized areas where property graphs have proven effective. Supply chain management uses graphs to represent suppliers, parts, products, logistics routes, and dependencies, allowing organizations to assess risk and analyze the impact of disruptions. Biomedical and bioinformatics applications build graphs of genes, proteins, interactions, pathways, and phenotypes to support discovery and hypothesis generation. Healthcare providers can model patients, diagnoses, treatments, and outcomes as a graph that helps to identify treatment patterns or adverse interactions. Cybersecurity teams maintain cyber threat graphs that link assets, vulnerabilities, exploits, and attackers to support threat hunting and incident response. Dependency and impact analysis in software engineering and manufacturing can utilize property graphs to represent components and their relationships within a bill of materials, enabling the tracing of consequences of a defect or vulnerability across a complex system.

Although each of these applications can be implemented with a relational database, a conventional relational schema represents connections through foreign-key values and association tables rather than stored, identity-bearing edges. Multi-hop, variable-length, or topology-driven questions therefore require chains of joins or recursive queries that reconstruct those connections, making such queries less direct as the traversed shape grows or varies. Property graphs do not displace relational systems for tabular workloads; they make relationships and traversal first-class, which aligns more closely with the connectivity-centered uses above (Sadalage and Fowler 2012).

2.2 Limitations of Property Graph Databases

Despite their strengths, property graph databases are not a universal solution and come with their own trade-offs. For workloads that consist primarily of simple, flat data with few relationships, a traditional relational database or a document store may be more straightforward and resource-efficient. Analytics that require scanning very large proportions of the data for aggregation, rather than traversing specific paths, can often be executed more effectively in columnar or data warehouse systems that are optimized for such scans. The explicit storage of edges, together with rich properties on both nodes and edges, can introduce storage overhead compared to a highly normalized relational design. Distributing a graph across multiple machines is also challenging because the natural structure of a graph does not always align with clean partitions, and traversals may need to cross physical boundaries, raising the cost of some queries and complicating the design of sharding strategies.

Chapter 2 — Takeaways

- **Purpose:** Characterized property graph databases as systems optimized for connected data and traversal-heavy workloads.
- **Introduced:** Property graphs with identity-bearing nodes and edges; labels and key–value properties as built-in modeling primitives; pattern matching and path navigation as native query mechanisms.
- **Key distinctions:** Relationships as first-class objects versus join-reconstructed relationships; schema-optional practice versus schema-governed expectations.
- **Consequences:** Framed why pattern matching and path semantics had to be standardized precisely; highlighted practical portability issues caused by vendor divergence; motivated a separation between the data model, query semantics, and updates.
- **Bridge:** Prepared the conceptual baseline needed to position GQL against other data models and query paradigms.

3 Positioning

This chapter positions GQL within the broader evolution of database models and query languages. We trace the main conceptual lineages that shaped contemporary graph querying, ranging from logic- and graph-oriented formalisms to relational, Semantic Web, and semi-structured paradigms. The aim is to clarify which ideas GQL inherits, which it adapts to property graphs, and where it deliberately departs from established traditions.

The running library example makes these contrasts concrete: following the chain `Reader`–`LibraryAccount`–`Loan`–`Book` requires a sequence of joins over foreign keys in a relational representation. A document representation can make one access path local through nesting, but shared entities then introduce duplication or references that require cross-document joins, whereas a graph retains the relationships as directly traversable edges (Sadalage and Fowler 2012).

3.1 Scholarly Analysis

The evolution of database models, particularly through the lens of query languages, illustrates a profound shift toward more expressive and flexible methods for data retrieval and manipulation. This evolution is evident in the significant scholarly contributions that have broadened the capabilities of database systems to accommodate complex, non-traditional data structures, especially those represented in graph-based models.

Kuper and Vardi pioneered the integration of formal logic into database systems, significantly enhancing the expressiveness of query languages (Kuper and Vardi 1984). By establishing a logical foundation, their approach facilitated complex queries that were previously infeasible with earlier database models, marking a significant advancement in the theory and application of database logic.

Continuing this trend, Gyssens, Paredaens, and Van Gucht introduced a graph-oriented object model that fundamentally altered how queries could interact with databases (Gyssens et al. 1990). Their model enabled end-users to intuitively query databases through a visual representation of data and schema as graphs, thereby simplifying complex query constructions and making database systems more accessible and easier to manipulate.

The hypernode model developed by Levene and Poulovassilis further expanded the querying capabilities of graph databases by introducing a nested graph structure (Levene and Poulovassilis 1990). This structure enabled the development of a powerful associated query language that could efficiently handle queries over hierarchical and recursive data structures, which are common in applications such as semantic web data and complex networked systems.

Güting introduced GraphDB, providing a specialized framework for modeling and querying graphs within databases, tailored specifically to applications such as geographical information systems and network analysis (Güting 1994). The query language designed for GraphDB was adept at handling connected data and complex relational queries, thus facilitating more sophisticated analyses and optimizations.

The G-Log query language, introduced by Paredaens et al., was another significant advancement, offering a declarative, graph-based language that allowed for the flexible querying of complex objects with identity (Paredaens et al. 1995). G-Log's ability to express changes in the database schema and instance as graph transformations opened new avenues for dynamic query processing and was particularly impactful in areas requiring advanced pattern matching and traversal capabilities.

Sheng and the Ozsoyoglu team further developed graph query processing techniques, which were crucial for efficiently executing complex queries in large-scale databases (Sheng et al. 1999). Their graph query language addressed the challenges associated with large and interconnected datasets, optimizing query execution to improve both performance and scalability.

Lastly, the nested-graph model by Poulovassilis and Levene provided a robust framework for representing and manipulating complex objects, which was complemented by a highly capable query language (Poulovassilis and Levene 1994). This language facilitated the manipulation of nested structures and complex object relationships, which are essential in fields such as computational biology and document databases.

Earlier work on semistructured data also treated data as labeled graphs and made path navigation central. The Object Exchange Model (OEM) underpinned the Lore repository, whose Lorel language extended the Object Query Language (OQL) with flexible path expressions, while UnQL supplied a language and algebra for rooted edge-labeled graphs based on structural recursion (Abiteboul et al. 1997; Buneman et al. 2000).

An earlier formal treatment of GQL covered a restricted read-only fragment—basic `FROM`–`MATCH`–`RETURN`–`WHERE` queries with fixed-length rigid patterns—based on the ISO working draft dated 2 February 2020 (Morra 2021). It therefore provides historical context rather than a normative account of the standard used in this book.

3.2 Relational Databases and SQL

The SQL standard has established itself as the central technology for relational database management, with decades of evolution reflected in successive ISO/IEC standardization efforts (ISO/IEC 2023b). Its longevity is due not only to its ubiquity in commercial and open-source systems, but also to a carefully designed combination of data model, typing discipline, transaction model, and declarative query language. As such, SQL constitutes the primary point of comparison for GQL: many of the design decisions

in GQL can be read as adaptations of long-established relational ideas to the setting of property graphs, while others deliberately depart from the relational tradition.

At the core of SQL lies a tabular, record-oriented data model. Relations (tables) can be understood as collections of records with a common column structure, and can be naturally viewed as files of records, spreadsheets, or tabular reports. In practice, tables are treated as multisets rather than sets: SQL admits duplicate rows, a feature that proves convenient for representing transactional and event data as they are generated by applications in finance, logistics, or web analytics. Conceptually, such tables can be mapped to text files with a header row and delimited fields, or to logical tables within reporting tools and business intelligence systems. GQL inherits this record orientation: collections of nodes or edges whose property records share the same type are logically equivalent to SQL tables. Each such typed collection can be seen as a possibly overlapping typed subpopulation of the node set V or of the edge set E, and these subpopulations are natural sources of binding-table rows in query evaluation. In contrast to SQL, however, GQL's binding tables disallow multiple columns with the same name, reflecting the importance of unambiguous variable bindings in pattern-based semantics.

An important aspect of SQL's pragmatic success is its treatment of incomplete information via null values. The language allows individual fields within a row to be marked as having no known value, resulting in three-valued logic in query conditions and a rich, albeit sometimes controversial, semantics for comparison and aggregation in the presence of missing data. This design acknowledges that real-world datasets are often partially known, subject to delayed acquisition, or constrained by privacy and access-control considerations. Although property graphs usually handle the absence of information via missing properties rather than explicit null values, the underlying concern is similar: query languages must remain usable and robust when confronted with incomplete or heterogeneous data.

SQL's query sublanguage is fundamentally compositional. Queries take one or more input tables and produce an output table, and this closure property underlies both subqueries and views. Nested queries enable the decomposition of complex retrieval tasks into smaller subproblems, while the view mechanism treats fixed queries as virtual tables. Views, whether virtual or materialized, provide multiple perspectives over the same underlying data, support modular query design, and encapsulate physical design decisions such as indexing or partitioning. Compositional closure of the language under its own result type is one of SQL's most powerful and enduring features. By contrast, present-day GQL is not yet a fully compositional language in this sense, as it lacks a general mechanism for graph-returning subqueries and views that define new persistent graphs. The inability to define graph projections or named graph views within the language constitutes a significant gap relative to SQL, and we return to this issue in our later discussion of possible future enhancements to GQL.

Another pillar of SQL's design is its schema-governed, closed-world nature. Data must conform to a declared schema that specifies tables, columns, types, and constraints. This strong typing and constraint infrastructure supports data quality, integrity, and consistency: primary keys, foreign keys, uniqueness constraints, and check predicates can all be enforced within the database itself. The metadata describing a schema can be communicated compactly, which in turn supports a regular translation from conceptual data models, such as entity-relationship diagrams (Chen 1976; Elmasri and Navathe 2015; Thalheim 2018) or UML class models (Fowler 2003), to physical database designs. A well-defined type system for field values provides a foundation for portable applications and interoperable tools. At the same time, schema information underpins query planning and optimization: cardinality estimates and other statistics are attached to schema objects, enabling cost-based optimizers to select efficient execution strategies. GQL shares the ambition of being schema-aware, via graph, node, and edge types and associated key and unique constraints, but its current constraint repertoire is considerably weaker than that of mature SQL implementations, especially in areas such as referential integrity, cardinality constraints, and rich domain constraints.

Transactional updates form another domain in which SQL has set long-standing expectations. The language is paired with a transaction model that supports atomic, consistent, isolated, and durable (ACID) execution, typically realized via serializability or well-understood relaxations thereof. Updates operate over schema-defined structures, and the semantics of `INSERT`, `UPDATE`, and `DELETE` statements are defined at the logical level of sets or multisets of rows, even though physical implementations may employ complex logging and indexing schemes. Crucially, transaction boundaries localize the effects of constraint failures. GQL adopts a similar high-level view of transactions over property graphs: its data-modifying operations dynamically validate the applicable graph constraints and graph-type conformance before completing successfully, while broader constraint forms and implementation-defined isolation extensions remain areas for evolution.

The declarative nature of SQL's data query language is central to its influence. Queries are written as concise statements of interest, multiset comprehensions over tables, rather than as procedural programs with explicit loops and navigational updates. This separation between logical intent and physical access path enables the database system to utilize query optimization techniques that consider storage layout, access methods, and buffering strategies, all of which are transparent to the application. Joins, selections, projections, and aggregations are composed algebraically, and the optimizer is free to reorder and transform these operations as long as the multiset semantics are preserved. Within this declarative framework, relationships between entities are computed through joins based on the equality of attribute values, typically guided by foreign key constraints. This "computed relationship" model is flexible and powerful, but it treats connections between entities as emergent from value equality, rather than as first-class structures.

GQL both inherits and departs from this relational tradition. On the one hand, the property graph data model is firmly record-oriented: nodes and edges carry property records whose types can be aligned with tabular structures. Collections of nodes or edges of the same type can be viewed as tables, supporting familiar ETL pipelines and easing data integration from files, spreadsheets, and relational sources. This record orientation underlies, for example, the tabular graph features in SQL/PGQ (ISO/IEC 2023a), where graph views are projected over existing tabular data. The ease with which property graphs can be layered over existing relational and file-based infrastructures is arguably one of the reasons for their commercial success.

On the other hand, GQL grants relationships a direct representation as edges in a graph, with their own identity and properties. Foreign keys in SQL can be understood as an indirect representation of such relationships via shared values, often requiring additional link tables to represent attributes associated with relationships. In GQL, these attributes can be attached directly to edges, resulting in more transparent and often more concise data models. This direct representation of relationships can be leveraged by systems optimized for edge and path exploration, enabling the efficient discovery of neighborhoods, reachability regions, and complex network patterns.

Finally, the graph-pattern matching sublanguage of GQL, a declarative, graph-oriented pattern language often referred to as GPML (Deutsch et al. 2022), offers expressive capabilities that are difficult to emulate elegantly in SQL. While SQL supports recursive queries, typically via recursive common table expressions, representing multi-hop path patterns, variable-length traversals, or complex topological constraints over large graphs quickly becomes verbose and opaque. In GQL, path patterns and graph patterns enable such traversals to be specified "topologically", that is, in terms of the shape of the desired paths and neighborhoods, rather than as low-level fixpoint computations over joins. This gives GQL a decisive advantage for tasks centered on graph connectivity, even as SQL remains the more mature and compositional language in the broader landscape of general-purpose database querying.

3.3 RDF and SPARQL

In the domain of the Semantic Web, the development of query languages has been profoundly influenced by established database languages such as SQL and OQL. Standardized as part of the Object Data Management Group (ODMG) object data standard, OQL is a declarative language over an object model with object identity, relationships, collections, and path navigation (Cattell et al. 2000). A notable early effort in this area was RQL (Karvounarakis et al. 2002), which introduced a robust framework combining elementary queries and iterators akin to those found in object query languages (Blakeley et al. 1991). RQL was designed to facilitate data and schema access, allowing queries to target specific types or navigate the RDF graph structure. Despite its comprehensive

functionality, RQL's broad feature set and syntactic complexity led to the development of SeRQL (Broekstra and Kampman 2003), which streamlined query execution by introducing shortcuts like object-property and object lists. Additionally, eRQL (Tolle 2006) emerged as a simplified adaptation of RQL, focusing on a keyword-based query interface that enhanced user accessibility.

Parallel developments included SquishQL and RDQL (Miller et al. 2002), which introduced subgraph matching filters that drew parallels to SQL's capabilities, and TriQL (Bizer et al. 2005), which extended these ideas by allowing queries over named graphs, thereby enabling evaluations across multiple sets of triples rather than just a single set. Despite their innovations, these languages commonly faced limitations in areas like store management and the definition of protocols for store access, which restricted their practical implementation.

Among these various RDF query languages, SPARQL (Harris and Seaborne 2013; Passant et al. 2013) has ascended as the preeminent standard, recognized for its comprehensive suite of features enabling detailed querying and manipulation of RDF data. SPARQL's design facilitates complex querying mechanisms that allow users to extract, query, and manipulate data across diverse and heterogeneous data sources. These sources may natively store data in RDF format or present non-RDF data as RDF via middleware solutions, thus extending SPARQL's utility across different data ecosystems. The influence of SPARQL on GQL is evident in its graph pattern-matching capabilities and the ability to handle complex queries across heterogeneous data sources.

The `SELECT` query in SPARQL is foundational, enabling the retrieval of raw data values from endpoints and presenting results in a structured tabular format. This function is central to most data retrieval operations, allowing for the precise selection and return of specific data elements. The `CONSTRUCT` query builds upon this capability by fetching data and reorganizing it into valid RDF structures, thereby facilitating the creation of new RDF graphs from existing data. This feature is particularly beneficial for tasks involving data transformation and integration. Lastly, the `ASK` query provides a unique function by returning a Boolean result (True or False), which is crucial for performing existence checks or for incorporating conditional logic within applications driven by complex data needs.

3.4 Tree-based Languages

Tree-based data models like JSON and XML have significantly influenced the development of database technologies by providing flexible, hierarchical structures ideal for many modern web and data interchange applications. These formats are inherently non-relational and support nested arrays and objects, which contrasts sharply with the flat, tabular structures typical of SQL databases.

XPath (Robie et al. 2017) and XQuery (Dyck et al. 2017) are two of the most prominent query languages designed for XML data. XPath is used to navigate through

elements and attributes in an XML document. It provides powerful syntax to pinpoint parts of an XML document, supporting complex querying capabilities such as Boolean operations, numerical operations, and node testing. XQuery, an extension of XPath, was designed to query and transform collections of XML data. It acts like a database query language but is tailored specifically for XML's hierarchical structure. Both XPath and XQuery leverage the tree-like nature of XML to perform sophisticated document manipulations and retrievals. GQL draws inspiration from these languages in its approach to handling nested and hierarchical data within graphs, enhancing its capability to model complex relationships.

In JSON, a similar trend is observed with languages such as JSONPath (Gössner et al. 2024), which provides capabilities analogous to XPath but is adapted for JSON's structure. JSONPath allows for the traversal of JSON objects using a path expression, making extracting and manipulating data from complex JSON structures straightforward. JMESPath (Saryerwinnie 2015) is another JSON query language that extends JSONPath's capabilities with additional operators and functions for more sophisticated querying and transformation tasks.

SQL:2016 (ISO/IEC 2016) has integrated robust support for JSON. This includes native JSON data types and functions like `JSON_VALUE`, `JSON_QUERY`, and `JSON_EXISTS`, allowing for efficient extraction and manipulation of JSON data within SQL queries. The standard also defines a JSON path language, similar to JSONPath, for navigating and querying JSON documents. Functions such as `JSON_OBJECT` and `JSON_ARRAY` facilitate the construction of JSON data from SQL query results, while predicates like `IS JSON` enable validation of JSON data. This integration allows seamless querying and manipulation of JSON data within a relational database context, combining the strengths of relational and non-relational data models.

LINQ (Language Integrated Query) (Albahari and Albahari 2008) is another influential technology that bridges the gap between programming languages and data queries. It provides a consistent model for managing data across different data sources, including XML and JSON. LINQ enables developers to use familiar list comprehension syntax from languages like C# to query various data structures, treating them as collections in the language itself. This integration enhances the accessibility and power of data manipulation directly within the programming environment.

These tree-based query languages have dramatically shaped the landscape of data querying and manipulation, making it easier for developers to interact with hierarchical data formats directly. Their influence extends to the design of modern database systems and querying interfaces, which increasingly cater to complex, nested data structures prevalent in contemporary applications. GQL's ability to handle such structures is a testament to these influences, providing a versatile and robust framework for modern data management needs.

3.5 Key-value and Document Databases

Key-value and document databases have also influenced the development of modern query languages by providing models that emphasize flexibility and simplicity, catering to various use cases where traditional relational databases may fall short.

Key-value databases, such as Amazon DynamoDB (Sivasubramanian 2012) and Redis (Da Silva and Tavares 2015), store data as a collection of key-value pairs. This simple data model allows for rapid read and write operations, making key-value databases suitable for high-performance applications and caching mechanisms. The query capabilities in key-value stores are often basic, focusing on efficiently retrieving values based on their keys. However, the key-value concept aligns closely with the property model in property graphs, where each node or edge can have associated key-value pairs representing various properties.

Document databases, such as MongoDB (Banker et al. 2016) and Couchbase (Brown 2012), extend the key-value model by storing structured data (documents) in formats like JSON. Each document contains key-value pairs but can embed complex data structures, such as nested arrays and objects. Document databases represent an expanded concept of key-value databases, allowing for a more nuanced and detailed representation of data. GQL benefits from these concepts by enabling the representation of complex data structures within graph nodes and edges, thus facilitating more detailed and expressive data models.

MongoDB's query language (Botoeva et al. 2018) offers extensive support for filtering, projecting, and aggregating data, enabling users to perform sophisticated data manipulations and analytics directly within the database. Couchbase supports advanced indexing and querying capabilities by using views and design documents, which allow users to define custom queries, transformations, and aggregations. Other document query languages include ArangoDB's AQL (ArangoDB 2024), which supports complex querying across JSON documents.

N1QL (Brown 2012) is Couchbase's query language designed to combine the power and familiarity of SQL with the flexibility of JSON. N1QL allows users to query, manipulate, and transform JSON data using a declarative syntax similar to SQL. Yet another query language is SQL++ (Ong et al. 2014), an extension of SQL designed for semi-structured data models. It builds on the foundation of SQL and expands it to handle nested, schema-less, and flexible data structures commonly found in JSON documents. SQL++ aims to provide a unified query language that bridges the gap between relational and NoSQL databases, allowing seamless integration and querying of heterogeneous data sources.

The influence of these databases on GQL is seen in how GQL handles semi-structured data. GQL's ability to represent properties on nodes and edges and to manage dynamic and evolving schemas draws inspiration from the document-oriented approach of these databases. Additionally, the efficient query execution and indexing strategies

developed for key-value and document databases have informed the design of GQL's querying mechanisms, ensuring they can handle large, complex graph datasets with high performance.

Chapter 3 — Takeaways

- **Purpose:** Positioned GQL within the ecosystem of database models and query languages by tracing conceptual lineages and contrasts.
- **Introduced:** A comparative map across relational SQL, RDF/SPARQL, and property-graph traditions; the idea of grounding graph querying in compositional, algebraic foundations.
- **Key distinctions:** Closed-world, constraint-oriented database reasoning versus open-world, entailment-oriented RDF reasoning; graph-native pattern/path querying versus tree/document navigation.
- **Consequences:** Clarified which semantics GQL inherited and which it deliberately did not; justified later choices such as bag-based intermediate results and explicit scoping rules; set expectations for how querying, updating, and catalogs fit into one semantic framework.
- **Bridge:** Led into a concrete running example that anchored later formal definitions.

4 GQL by Example

This chapter introduces GQL through a concrete, self-contained library domain that we use as a running example throughout the book. Starting from an informal picture of the schema and a small instance, we show how to express the same structure using GQL's data definition and manipulation features. We then formulate a series of queries over this graph, illustrating the core ideas of pattern matching, paths, and aggregation in the language.

4.1 Visual Representation

Schema illustration. Figure 4.1 presents the schema as a type-level property graph, abstracting away from individual borrowers and concrete books to reveal the structural backbone of the library domain. Each rectangular box denotes a node type, specifying which attributes are available for entities of that kind. The `Loan` type, for example, states that every loan node must carry at least one identifying property, `loanid: String`, which uniquely characterizes a loan within the system. The `LibraryAccount` type introduces an analogous identifier `num: String`, representing the account code under which borrowings are recorded. The `Reader` type is richer: it combines a numeric identifier `id: Int` with textual attributes `firstname: String` and `lastname: String`, together with a `birthdate: Date` that captures the temporal dimension of the patron's identity. A separate `LibraryCard` type models the physical card used at the circulation desk and self-service terminals; it has a single property `num: String` describing the card number printed on the plastic. Finally, the `Book` type specifies three bibliographic properties, `isbn: String`, `title: String`, and `author: String`, which together determine how book entities are represented in the graph.

The schema also defines its edge types and their property type sets. Edge type specifications carrying the labels `owns`, `uses`, `check_out`, and `returns_to` define typed connections between the corresponding node types: a `Reader` may own a `LibraryAccount`; the same `Reader` may use a `LibraryCard`; and a `Loan` is associated with a `LibraryAccount` via operations that record checking items out and returning them. Particularly noteworthy is the directed edge type from `Loan` to `Book` whose edge type label set is `:INCLUDES`. Unlike the other edge types, it declares the property types `due_date: Date` and `late_fee: Float`. This makes explicit that due dates and late fees are not inherent attributes of loans or books in isolation, but of the specific association between a given loan and a particular book instance. Taken together, the schema figure can be read as a compact contract for all valid library graphs in this model: it prescribes the node types that may exist, the edge types that may connect them, and the types of values

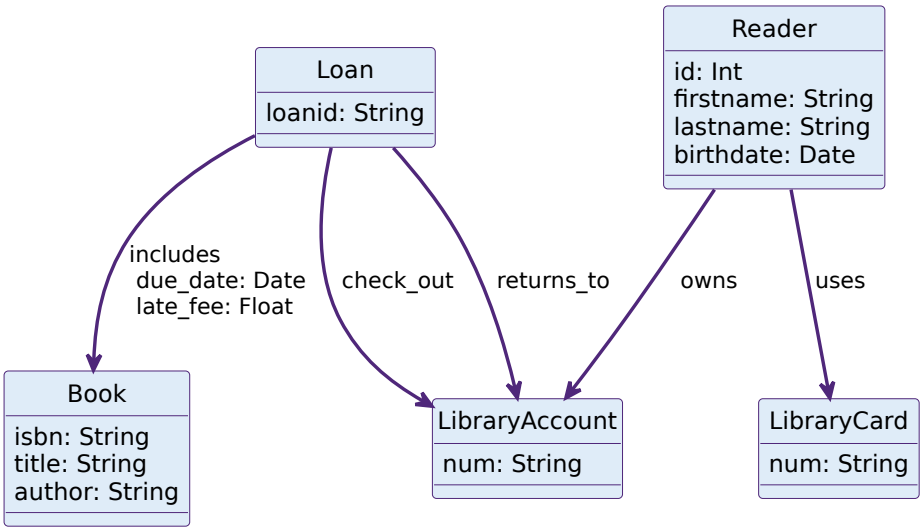


Figure 4.1.: Schema illustration

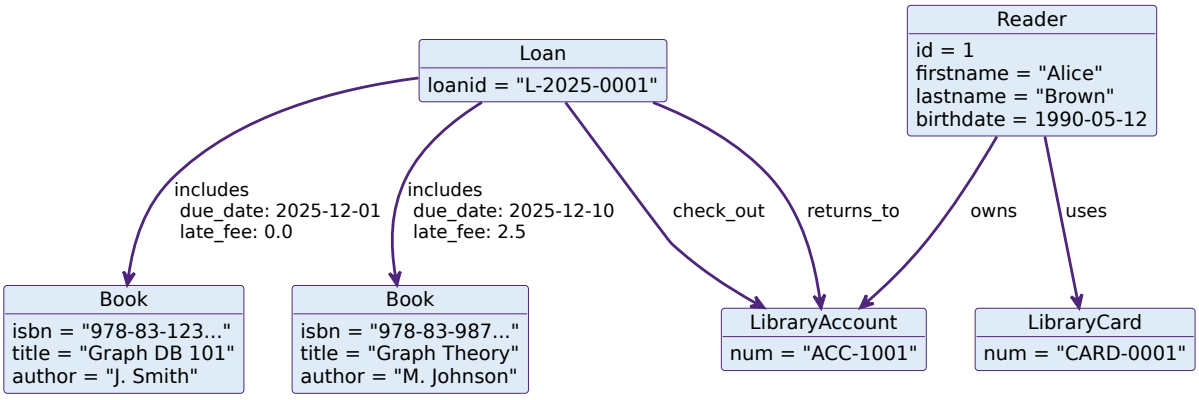


Figure 4.2.: Instance illustration

that may decorate both nodes and relationships. The underlying property-graph structure and the formal definitions of node, edge, and graph types are given in Sections 6.2, 15.3 and 15.4.

Instance illustration. Figure 4.2 instantiates this schema with a small, concrete example of a library situation. At its center stands a single loan node, identified by the property `loanid: "L-2025-0001"`. This loan is tied to one library account node whose `num: "ACC-1001"` indicates the account under which the borrowing is registered. The account belongs to a specific reader, represented by a `Reader` node that carries basic personal data: an internal numeric identifier (`id: 1`), the first name `"Alice"`, the last name `"Brown"`, and the date of birth `1990-05-12`. A separate `LibraryCard` node, with `num: "CARD-0001"`, captures the physical card that Alice uses to interact with the library system.

Two distinct `Book` nodes complete the picture. Each represents a different catalog item, described by an ISBN, a title, and an author. One book has `isbn:`

"978-83-123...", title: "Graph DB 101", and author: "J. Smith". The other is recorded as isbn: "978-83-987...", title: "Graph Theory", and author: "M. Johnson". The edges stitching these nodes together mirror the schema's relationship types. An edge labeled owns connects the reader to the library account, expressing that this account is Alice Brown's. A second edge labeled uses links the reader to the library card, indicating that this card is the one she presents at the library.

The lifecycle of the borrowing process is then captured via edges from the loan node to the account node, with labels such as check_out and returns_to distinguishing the act of borrowing from the act of returning. The association between the loan and the individual books is modeled by includes edges from the loan node to each of the two book nodes. In the example, these relationship instances carry their own properties: one includes edge has due_date: 2025-12-01 and late_fee: 0.0, indicating that no late fee is recorded for that item, while the other has due_date: 2025-12-10 and late_fee: 2.5, showing that a fee has accrued for that particular item. In this way, the instance diagram can be read as a concrete graph that conforms to the structural constraints of the schema: every node and edge has exactly the label sets and property sets prescribed in the first illustration, turning the abstract specification into a tangible example of a library encoded as a property graph. The formal representation of this instance as sets of nodes, edges, labels, properties, and endpoints is given in Section 6.2.

4.2 Data Definition Language

The Data Definition Language (DDL) of GQL allows us to turn the informal schema of Figure 4.1 into an explicit graph type that constrains all admissible library graphs. In this example, we introduce a single graph type libraryGraphType. Its body specifies the node types (with their labels and property types), the edge types (with their labels, direction, and any edge properties), and a set of key constraints that capture the intended identifiers for loans, accounts, readers, cards, and books.

```
CREATE GRAPH TYPE libraryGraphType
AS {
  /* Node types */
  (Loan :Loan { loanid STRING NOT NULL }),
  (LibraryAccount :LibraryAccount { num STRING NOT NULL }),
  (Reader :Reader { id INT NOT NULL, firstname STRING,
                    lastname STRING, birthdate DATE }),
  (LibraryCard :LibraryCard { num STRING NOT NULL }),
  (Book :Book { isbn STRING NOT NULL, title STRING, author STRING }),

  /* Edge types */
  (Reader)-[:OWNS]->(LibraryAccount),
```

```
  (Reader)-[:USES]->(LibraryCard),
  (Loan)-[:CHECK_OUT]->(LibraryAccount),
  (Loan)-[:RETURNS_TO]->(LibraryAccount),
  (Loan)-[:INCLUDES { due_date DATE, late_fee FLOAT }]->(Book),

  /* Key constraints */
  CONSTRAINT LoanHasKey
    FOR (l:Loan)
    REQUIRE (l.loanid) IS KEY,
  CONSTRAINT LibraryAccountHasKey
    FOR (a:LibraryAccount)
    REQUIRE (a.num) IS KEY,
  CONSTRAINT ReaderHasKey
    FOR (r:Reader)
    REQUIRE (r.id) IS KEY,
  CONSTRAINT LibraryCardHasKey
    FOR (c:LibraryCard)
    REQUIRE (c.num) IS KEY,
  CONSTRAINT BookHasKey
    FOR (b:Book)
    REQUIRE (b.isbn) IS KEY
}
```

The body of the `CREATE GRAPH TYPE` statement declares the node types, edge types, and constraints that constitute `libraryGraphType`. Each parenthesized declaration, such as `(Reader :Reader { id INT NOT NULL, firstname STRING, ... })`, defines a node type. The identifier `Reader` is a local node type alias, `:Reader` specifies its label set, and the declarations inside braces specify the available properties together with their value types.

Similarly, the declaration

`(Loan)-[:INCLUDES { due_date DATE, late_fee FLOAT }]->(Book).`

defines a directed edge type from `Loan` nodes to `Book` nodes. The edge has the label `INCLUDES` and may carry the properties `due_date` and `late_fee`, whose values must have the types `DATE` and `FLOAT`, respectively.

The `CONSTRAINT` clauses declare key constraints for selected node types. In each constraint, the `FOR` clause identifies the node type to which the constraint applies and binds a variable to matching nodes. For example, `FOR (l:Loan)` applies the constraint to `Loan` nodes and binds them to the variable `l`. The corresponding `REQUIRE (l.loanid) IS KEY` clause declares the `loanid` property as the key of that node type.

Consequently, `loanid` is the key of `Loan` nodes, `num` is the key of `LibraryAccount` nodes, `id` is the key of `Reader` nodes, `num` is the key of `LibraryCard` nodes, and `isbn` is the key of `Book` nodes. Graphs constrained by `libraryGraphType` must satisfy these key constraints in addition to the declared node and edge type requirements. If a modification would violate an applicable constraint or the constraining graph type, the offending operation does not complete successfully. The formal treatment of graph types, graph-type creation, key constraints, and their enforcement is given in Sections 15.4, 15.6, 16.1 and 16.2.

The following statement creates an empty graph constrained by this graph type:

```
CREATE GRAPH libraryGraph TYPED libraryGraphType
```

4.3 Data Manipulation Language

The Data Manipulation Language (DML) of GQL comprises the data-modifying statements `INSERT`, `SET`, `REMOVE`, and `DELETE`, together with procedure calls that execute data-modifying procedures. Where the Data Definition Language fixes the shape of admissible graphs via graph types and constraints, the DML operates on a particular graph instance. The graph named `libraryGraph` created above has `libraryGraphType` as its constraining graph type. All data-modifying statements act on the current working graph, which is selected by a `USE` clause at the beginning of a *linear data-modifying statement*. The selection of the current working graph by `USE` is formalized in Section 17.2.

Populating the graph. The following focused linear data-modifying statement inserts exactly the nodes and edges shown in Figure 4.2 into `libraryGraph`. The `INSERT` statement contains a single *insert graph pattern*, written as a collection of insert path patterns that share element variables such as `l`, `a`, `r`, `c`, `b1`, and `b2`. Each element pattern specifies labels and an initial property set using a `{ property key : value }` notation.

```
USE libraryGraph
INSERT
  (l:Loan { loanid: 'L-2025-0001' })-
  [:CHECK_OUT]->(a:LibraryAccount {
    num: 'ACC-1001'
  }),
  (l)-[:RETURNS_TO]->(a),
  (r:Reader {
    id: 1,
    firstname: 'Alice',
    lastname: 'Brown',
    birthdate: DATE '1990-05-12'
  })-[:OWNS]->(a),
```

```
  (r)-[:USES]->(c:LibraryCard {
    num: 'CARD-0001'
  }),
  (l)-[:INCLUDES { due_date: DATE '2025-12-01',
                   late_fee: 0.0 }]->(b1:Book {
    isbn:  '978-83-123...',
    title: 'Graph DB 101',
    author: 'J. Smith'
  }),
  (l)-[:INCLUDES { due_date: DATE '2025-12-10',
                  late_fee: 2.5 }]->(b2:Book {
    isbn:  '978-83-987...',
    title: 'Graph Theory',
    author: 'M. Johnson'
  })
```

The first pattern creates a loan node `l` with property `loanid` and connects it, via a `CHECK_OUT` edge, to a library account node `a` whose identifier is the account number `'ACC-1001'`. The next two patterns introduce the reader node `r` and the library card `c`: `r` is labeled `:Reader` and initialized with the personal attributes from the instance figure, and is connected to `a` and `c` by `OWNS` and `USES` edges, respectively. The final two patterns create two distinct book nodes, `b1` and `b2`, each labeled `:Book` and equipped with its bibliographic properties, and attach them to the loan via `INCLUDES` edges. The edge property maps inside the braces specify the due date and late fee for each item. Because the schema in Figure 4.1 is declared as a constraining graph type, the evaluation of this statement is required to produce a graph that still conforms to `libraryGraphType`: all inserted elements must respect the declared element types, labels, and property types. The applicable constraints are validated before the statement completes successfully. The typing and operational semantics of this insertion are given in Section 14.1.

Updating properties with `SET`. Once the graph contains data, the `SET` statement can be used to change properties or labels of existing nodes and edges. In a typical pattern, a `MATCH` statement first binds element variables to the graph elements of interest, and a following `SET` statement updates their properties. The two statements together form a single linear data-modifying statement over the same working graph.

```
USE libraryGraph
MATCH (l:Loan { loanid: 'L-2025-0001' })-[i:INCLUDES]->(b:Book)
WHERE b.isbn = '978-83-987...'
SET   i.late_fee = 0.0
```

Here MATCH selects the INCLUDES edge i connecting loan L-2025-0001 to the book with the second ISBN. The SET statement then assigns a new value to i.late_fee, clearing the fee for this particular item. Because late_fee is declared as a FLOAT property type of the edge type whose label set is :INCLUDES, the value 0.0 is type-correct, and the update is required to preserve graph-type conformance. The read-then-update behavior and write-set semantics of this operation are given in Section 14.4.

Deleting graph elements. Finally, the DELETE statement removes nodes and edges from the working graph. When used with the DETACH keyword it also removes all edges incident to a deleted node. The following example deletes the loan node L-2025-0001 and all of its incident edges, while leaving the reader, account, card, and book nodes intact:

```
1 USE libraryGraph
2 MATCH (l:Loan { loanid: 'L-2025-0001' })
3 DETACH DELETE l
```

The MATCH clause binds the variable l to the unique loan node with the given identifier. The subsequent DETACH DELETE removes that node together with every edge attached to it, including the CHECK_OUT, RETURNS_TO, and INCLUDES edges shown in Figure 4.2. As with insertion and updates, the semantics of DELETE require that the resulting graph conforms to its constraining graph type and that its applicable constraints are satisfied before the statement completes successfully. In this example, deleting l and its incident edges preserves the declared property types and key constraints. The formal distinction between plain deletion and DETACH DELETE, including the treatment of incident edges, is given in Sections 14.2 and 16.2.

Taken together, these examples show how GQL's DML can construct, evolve, and clean up concrete library graphs, while the graph type libraryGraphType and its constraints continue to govern the shape and well-formedness of every admissible state.

4.4 Data Query Language

The Data Query Language (DQL) of GQL provides read-only statements for navigating, filtering, and aggregating the data stored in a working graph. Syntactically, the examples in this section are instances of *linear query statements* in the sense of ISO 39075: they start with an optional USE clause and then apply a sequence of query statements such as MATCH, LET, and FILTER, followed by a primitive result statement of the form RETURN ... optionally combined with an ORDER BY and LIMIT clause. All queries below assume that the working graph is the graph libraryGraph constrained by the graph type libraryGraphType introduced earlier. Each query is evaluated independently on the initial instance in Figure 4.2; the illustrative SET and DELETE statements above are not

applied to that instance. Unless an `ORDER BY` clause is present, the displayed row order in the result tables is not significant.

Simple Rigid Pattern. A simple rigid pattern is a fixed arrangement of node and edge patterns without any path quantifiers or alternation. Such a pattern describes a specific configuration—a small neighborhood—that queries attempt to match exactly. In the library example, we might want to retrieve all items whose due date precedes a chosen reference date, together with the reader who borrowed them and the account through which the loan was recorded:

```
USE libraryGraph
MATCH (r:Reader)-[:OWNS]->(a:LibraryAccount),
      (r)-[:USES]->(c:LibraryCard),
      (l:Loan)-[i:INCLUDES]->(b:Book),
      (l)-[:CHECK_OUT]->(a)
WHERE i.due_date < DATE '2025-12-31'
RETURN r.firstname AS firstname,
       r.lastname  AS lastname,
       b.title     AS title,
       i.due_date  AS due_date,
       i.late_fee  AS late_fee
```

Expected result for the instance in Figure 4.2:

firstname	lastname	title	due_date	late_fee
Alice	Brown	Graph DB 101	2025-12-01	0.0
Alice	Brown	Graph Theory	2025-12-10	2.5

The `MATCH` statement here consists of a graph pattern with four path patterns, separated by commas, that all share variables. The reader `r` is connected to a library account `a` via an `OWNS` edge and to a library card `c` via a `USES` edge. A loan `l` is connected by `CHECK_OUT` to the same account `a` and `INCLUDES` a book `b` via an edge `i`. The `WHERE` clause of the graph pattern restricts attention to those `INCLUDES` edges whose `due_date` precedes the given calendar date. The `RETURN` clause then projects a binding table whose columns correspond to the reader's name, the title of the borrowed book, and the due date and late fee carried as edge properties on `i`. Because the pattern does not contain any path quantifiers, each solution row corresponds exactly to one concrete instance of this rigid configuration in the graph. The formal definitions of graph patterns, matching, binding-table formation, and result projection are given in Sections 7.1, 7.4 to 7.6 and 12.3.

Simple rigid patterns can be enriched with intermediate derived values using `LET` and subsequently filtered with `FILTER`. The next query looks for loans that contain both of the two example books and computes the total late fee incurred across those two items:

```
USE libraryGraph
MATCH (l:Loan)-[i1:INCLUDES]->(b1:Book),
      (l)-[i2:INCLUDES]->(b2:Book)
WHERE b1.isbn = '978-83-123...'
  AND b2.isbn = '978-83-987...'
LET  total_late_fee = i1.late_fee + i2.late_fee
FILTER WHERE total_late_fee > 0.0
RETURN l.loanid       AS loanid,
       total_late_fee AS totalLateFee
```

Expected result for the instance in Figure 4.2:

loanid	totalLateFee
L-2025-0001	2.5

The graph pattern selects pairs of `INCLUDES` edges `i1` and `i2` that originate from the same loan `l` but point to different books `b1` and `b2` with the specified ISBN values. The `LET` statement introduces a value variable `total_late_fee` computed by adding the two edge-local late fee amounts. The `FILTER` statement then retains only those bindings for which the derived total is strictly positive. Here, the `WHERE` clause is part of the graph pattern and restricts that pattern's matches. By contrast, `FILTER [WHERE]` is a separate simple query statement that filters the current working table and can use bindings introduced by preceding statements, such as `LET`. The final `RETURN` projects the loan identifier and the computed `totalLateFee`, illustrating how rigid patterns, local arithmetic, and binding-table filters combine within a single linear query. The corresponding selection and projection operations, including the three-valued interpretation of filtering predicates, are formalized in Sections 8.3 and 9.1.

Conjunctive Regular Path Queries. Conjunctive Regular Path Queries (CRPQs) extend simple patterns by allowing conjunctions of multiple path patterns that share variables. In logical terms, a CRPQ describes several paths through the graph that must be simultaneously present and must meet joint constraints on shared endpoints or intermediate nodes. Even without explicit path quantifiers, the library domain already exhibits meaningful CRPQs.

Consider the task of finding pairs of readers who have, possibly at different times, borrowed the very same book. We are interested in such co-borrowers of the same book instance, but we want to list each unordered pair only once:

```
USE libraryGraph
MATCH (r1:Reader)-[:OWNS]->(a1:LibraryAccount),
      (r2:Reader)-[:OWNS]->(a2:LibraryAccount),
```

```
      (l1:Loan)-[:CHECK_OUT]->(a1),
      (l2:Loan)-[:CHECK_OUT]->(a2),
      (l1)-[:INCLUDES]->(b:Book),
      (l2)-[:INCLUDES]->(b)
WHERE r1.id < r2.id
RETURN DISTINCT r1.firstname AS reader1_firstname,
       r1.lastname  AS reader1_lastname,
       r2.firstname AS reader2_firstname,
       r2.lastname  AS reader2_lastname,
       b.title      AS common_title
```

Expected result for the instance in Figure 4.2:

Result	Rows
Empty binding table	0

This query consists of several path patterns that are conjoined by commas into a single graph pattern. Two readers `r1` and `r2` own accounts `a1` and `a2`, respectively. Each account has at least one loan (`l1` or `l2`) that checks out that account, and each of those loans includes the same book node `b`. The shared variable `b` enforces that the two paths meet at a common book, which is exactly the kind of join between paths that CRPQs support. The inequality predicate `r1.id < r2.id` in the `WHERE` clause removes the second ordering of each reader pair. `RETURN DISTINCT` also coalesces duplicate projected rows that can arise from multiple matching loans or parallel matches, so each projected reader-pair and title combination is reported once. The `RETURN` clause surfaces both readers' names and the shared book title.

In full generality, regular path patterns in GQL can also use path quantifiers and alternation to describe arbitrarily long routes through the graph. The present example already illustrates the key idea of CRPQs: multiple path patterns, possibly of different lengths, are matched jointly, with constraints expressed over the shared nodes and edges. The joint matching of path-pattern components through shared variables is formalized in Sections 7.1 and 7.4.

Linear Composition. Linear composition refers to the way in which a GQL query can be built as a sequence of simple query statements that gradually transform the current working table. Each step—a `MATCH`, `LET`, or `FILTER` statement—takes the bindings produced so far, extends them with new variables, or discards those that fail a predicate. The final step is a primitive result statement that projects and optionally orders or pages the result.

The following query finds all books borrowed by a reader with the last name `'Brown'`, restricts attention to items whose due date precedes a reference date, and orders the result by due date:

```
USE libraryGraph
MATCH (r:Reader)-[:OWNS]->(a:LibraryAccount)
WHERE r.lastname = 'Brown'
MATCH (a)<-[:CHECK_OUT]-(l:Loan)-[i:INCLUDES]->(b:Book)
FILTER WHERE i.due_date < DATE '2025-12-31'
RETURN l.loanid   AS loanid,
       b.title    AS title,
       i.due_date AS due_date,
       i.late_fee AS late_fee
ORDER BY due_date
```

Expected result for the instance in Figure 4.2:

loanid	title	due_date	late_fee
L-2025-0001	Graph DB 101	2025-12-01	0.0
L-2025-0001	Graph Theory	2025-12-10	2.5

The first `MATCH` and its `WHERE` clause define the initial binding table: each row binds `r` and `a` to a reader with last name `'Brown'` and one of their accounts. The second `MATCH` uses these bindings as input and extends each row with a loan `l`, an `INCLUDES` edge `i`, and a book `b` that form the following fixed path:

```
(a)<-[:CHECK_OUT]-(l)-[:INCLUDES]->(b)
```

The `FILTER` statement then prunes those rows where `i.due_date` is not before the given reference date. Finally, the `RETURN` projects a smaller record—loan identifier, book title, due date, and late fee—and the trailing `ORDER BY` clause orders the resulting binding table by the due date column. This example shows how successive `MATCH` and `FILTER` statements can replace the kind of pipeline-oriented `WITH` clause found in other graph query languages, while remaining within the linear composition framework defined by the standard. The formal treatment of successive `MATCH` operations and whole-query pipeline composition is given in Sections 7.6, 9.5 and 12.3.

Order, Limit, and Aggregation. Aggregation, ordering, and pagination complete the picture by allowing queries to summarize larger result sets and to return just the most relevant rows. In GQL, aggregation is expressed via aggregate functions such as `SUM` and `COUNT`, and is combined with a `GROUP BY` clause inside the `RETURN` statement body. Ordering and limiting are provided by an `ORDER BY` and `LIMIT` clause, which together form the ⟨*order by and page statement*⟩ attached to the `RETURN`.

The next query computes, for each author, the total late fees incurred on loans of their books, and then returns the ten authors with the highest total late fee:

```
USE libraryGraph
MATCH (l:Loan)-[i:INCLUDES]->(b:Book)
LET   authorName = b.author
RETURN authorName           AS author,
       SUM(i.late_fee)      AS total_late_fee
GROUP BY author
ORDER BY total_late_fee DESC
LIMIT 10
```

Expected result for the instance in Figure 4.2:

author	total_late_fee
M. Johnson	2.5
J. Smith	0.0

The `MATCH` statement produces bindings of loans, `INCLUDES` edges, and books. The `LET` statement then introduces a value variable `authorName` that extracts the `author` property of each matched book. Inside the `RETURN` statement, the `GROUP BY` clause groups the current working table by the return-item alias `author`, and the aggregate function `SUM(i.late_fee)` computes, for each such group, the total amount of late fees associated with that author's books. The two return items are projected under the aliases `author` and `total_late_fee`; these aliases become columns of the outgoing binding table. The trailing `ORDER BY`–`LIMIT` combination forms an order-by-and-page statement: it sorts the grouped rows in descending order of `total_late_fee` and retains only the first ten records. The overall linear query statement thus exemplifies how grouping, ordering, and pagination interact in a way that is faithful to the formal semantics of `RETURN`, `GROUP BY`, and the order-by-and-page statement in the GQL standard. The formal semantics of grouping, aggregate evaluation, result formation, ordering, and limiting is given in Sections 12.1, 12.3, 13.1 and 13.3.

Chapter 4 — Takeaways

- **Purpose:** Demonstrated GQL end-to-end on a self-contained running example to ground later formalization.
- **Introduced:** A library-domain example schema and dataset; representative DDL, DML, and query idioms used throughout the book; a consistent narrative for graph creation, population, and querying.
- **Key distinctions:** Schema specification versus instance data; schema-level edge types versus the labels and properties of individual edge instances.

- **Consequences:** Made concrete what the later semantics had to explain (bindings, nulls, multiplicities, paths, updates); fixed a shared vocabulary of constructs; established realistic expectations about how users compose graph workloads.
- **Bridge:** Motivated the need to freeze an abstract data model as the reference point for all subsequent semantics.

Part II

Core Model and Patterns

5 A Conceptual Glossary of GQL

This chapter collects the technical vocabulary used in the semantic chapters of the book. It is not arranged alphabetically, because an alphabetic list is convenient for lookup but poor for learning a model. Instead, the terms are ordered as a reading path: elementary notions come first, and later sections build on the words that have already been introduced. The aim is that a reader can move through the chapter without encountering a central technical term before it has been explained.

The definitions are intentionally short. They do not replace the full formal development in the main chapters, but they provide a compact conceptual map of that development. When a term has a precise formal treatment elsewhere, the definition below states the role of the term and the intuition needed to follow later references. The sequence also indicates how the parts of GQL fit together: values and names support stored structures, stored structures support patterns, patterns produce tables, tables are transformed by queries, while updates, transactions, authorization, and diagnostics govern execution.

5.1 Foundations

We begin with the vocabulary that is needed before any particular data model is discussed. A statement manipulates data by means of values, and the language needs types to say which values are admissible in a given position. Names and identifiers then give the text of a statement a way to refer to things, while variables and bindings explain how evaluation records intermediate choices. The last terms in this section move from single bindings to mappings and merges, because later evaluation rules repeatedly combine partial assignments into larger assignments. No structural assumptions are needed yet; the point is only to establish the naming and binding machinery on which the rest of the chapter depends. The basic terms for this layer are:

- **Data**: facts or stored content available to GQL for reading, returning, or changing.
- **Type**: a classification that determines which data items are admissible in a particular place.
- **Value**: a definite, immutable, and irreducible unit of data of some type.
- **Null value**: the special value used to indicate the absence of other data.
- **Data type**: a set of elements with shared characteristics representing data.
- **GQL-program**: a complete top-level syntactic unit of GQL.

- **GQL-request**: a GQL-program together with a possibly empty dictionary of named, typed parameter values supplied for that request.
- **Execution**: carrying out of a GQL-request to compute a result and possibly change runtime state.
- **Operation**: an identifiable action performed while carrying out a GQL-request.
- **Side effect**: a change caused during execution of a GQL-request that can be detected by execution of another operation in the same or another GQL-request.
- **Evaluation**: a computation of a result that is not permitted to cause side effects.
- **Procedure**: a description of a computation on input arguments whose execution reports success or failure and may cause side effects.
- **Command**: an operation not specified by a procedure written in GQL.
- **Statement**: an operation executed as part of a procedure that may update runtime state and cause side effects.
- **Implementation-defined**: possibly differing between GQL implementations, but specified by the implementer for each particular implementation.
- **Implementation-dependent**: possibly differing between GQL implementations, but not required to be specified by the implementer.
- **Identifier**: a value by which something is identified.
- **Name**: an identifier used by people and by statement text to refer to something.
- **Variable**: a name used inside a statement to hold a value or formal item found during evaluation.
- **Binding**: an association between a variable name and a value.
- **Mapping**: a partial assignment from variables to values.
- **Unbound variable**: a variable for which the current mapping has no binding; this is distinct from a variable bound to the null value.
- **Compatible mappings**: two mappings that agree on every variable they both bind.
- **Merge**: the union of two compatible mappings.
- **Qualified name**: a name made from several components, used when a short name would not be precise enough.
- **Object**: something that the system can refer to, type, protect, or manipulate.
- **GQL-object**: an object capable of being manipulated directly by execution of a GQL-request.
- **Global object identifier**: an effectively immutable internal object identifier associated with a GQL-object that reifies its identity during execution of a GQL-request.

- **Globally identifiable object**: a GQL-object associated with a global object identifier.
- **Syntactically resolved reference**: a grammatical reference that statically identifies a GQL-object from the reference's syntax and location.
- **Globally resolved reference**: a reference that identifies its referent by a global object identifier.
- **Descriptor**: a coded description of the metadata of a GQL-object.
- **Primary object**: an independently definable GQL-object; every primary object is globally identifiable and has a descriptor.
- **Secondary object**: a GQL-object necessarily defined as a component of another GQL-object.

5.2 Graphs

With the naming layer in place, we can introduce the structure that gives GQL its distinctive character. A graph is built from points and links, and the language treats those parts as objects that can be named, typed, selected, and changed. The first definitions below describe the bare connectivity structure: nodes, edges, endpoints, direction, loops, and multiple edges. The later definitions add labels and properties, which are the features that turn a simple connected structure into a property graph. This section also introduces the idea that a stored graph may either stand without a fixed structural contract or be checked against one. The graph vocabulary introduced here is:

- **Graph**: a primary GQL-object comprising zero or more identifiable points and zero or more identifiable connections between them.
- **Node (vertex)**: an identifiable point in a graph that represents an entity.
- **Edge (relationship)**: an identifiable connection in a graph between two nodes.
- **Graph element**: a node or an edge.
- **Source node**: the node where an oriented link starts.
- **Destination node**: the node where an oriented link ends.
- **Endpoint**: one of the two nodes connected by an edge.
- **Directed edge**: an edge whose endpoints have a source and a destination.
- **Undirected edge**: an edge that may be traversed without choosing a fixed source-to-destination direction.
- **Directionality**: the information saying whether an edge is directed or undirected.
- **Loop**: an edge whose two endpoints are the same node.
- **Multiple edges**: several edges between the same pair of nodes.

- **Multigraph**: a graph that permits multiple distinct edges between the same pair of nodes.
- **Mixed graph**: a graph that can contain both directed and undirected edges.
- **Label**: a name attached to a node or edge to classify it.
- **Label set**: the finite set of labels attached to one node or edge.
- **Edge label**: a label contained in the label set of an edge; by itself it is only a classification mark and does not specify the edge's complete structural contract.
- **Property key**: the name used as the key part in an attribute key-value pair.
- **Property value**: the value stored under such a key.
- **Property**: a key-value pair attached to a node or edge.
- **Property map**: mapping from property keys to property values for one node or edge.
- **Property set**: the group of properties carried by one node or edge.
- **Property graph**: a graph whose nodes and edges can have labels and properties.
- **GQL graph**: the book's structural property-graph model, comprising disjoint finite sets of identifiable nodes and edges together with their labels, properties, edge endpoints, and edge directionality.
- **Constraint**: a rule that a graph must satisfy.
- **Constraint dictionary**: a dictionary associated with a graph that maps zero or more unique names to constraints imposed directly on the graph's elements.
- **Constraining graph type**: an optional globally resolved reference recorded for a graph; its referent supplies the structural and integrity rules that the graph must satisfy.
- **Graph descriptor**: optional constraining graph type together with the graph's constraint dictionary.
- **Open graph**: a graph that has no declared structural contract, so it is not checked against one fixed graph shape.
- **Closed graph**: a graph that has a declared structural contract and must satisfy it.
- **Conformance**: the fact that a value, element, graph, statement, or implementation satisfies the rules that apply to it.
- **Graph-entry conformance**: the requirement that a graph satisfy its directly defined constraints and, when it has a constraining graph type, the structural and constraint requirements supplied by that referent.
- **Graph type violation**: an error reported when a graph does not satisfy the structural requirements or constraints of its constraining graph type.

5.3 Context

Graphs and other GQL-objects are not interpreted in isolation. A statement is evaluated against a current store state, and names are resolved through a catalog, namespaces, and the current working defaults. The terms in this section describe that ambient setting. They explain where named objects live, how a short name is connected to a particular object, and how a session records the context and options that influence later statements. The order starts with the global store and catalog and ends with session-local choices, because session state is meaningful only after the available objects and naming rules are clear. The context terms used in this part are:

- **Catalog**: a persistent, hierarchically organized collection of named GQL-objects.
- **Catalog object**: a GQL-object defined, possibly transitively, in the GQL-catalog.
- **Primary catalog object**: a GQL-object that is both a primary object and a catalog object.
- **GQL-schema**: a persistent dictionary of primary catalog objects.
- **GQL-directory**: a persistent dictionary of GQL-directories and GQL-schemas.
- **Store state**: the current abstract state of the stored graphs, catalog, and other data that statements evaluate against.
- **Execution context**: the runtime state created for an execution and updated as its operations proceed.
- **Catalog entry**: one item in the catalog, such as a stored graph item or a type definition.
- **Graph entry**: a catalog entry that stores a graph together with its optional constraining graph type and its own constraint dictionary.
- **Site**: a place occupied by an instance of a specified data type.
- **Declared type**: the unique data type common to every instance that may occupy a site, or to every result of a specified operation.
- **Assignment**: the operation that initializes or replaces the value at a target site with a source value, applying the relevant type conversions when required.
- **Store assignment**: assignment to a site using the assignment and type-conversion rules defined for stored values and references.
- **Namespace**: a naming area that helps the same short name mean different things in different contexts without conflicts.
- **Name resolution**: the process that decides which catalog object, variable, type, or local binding a name refers to.
- **Working schema reference**: the schema reference used as the default parent for resolving relative catalog-object references.

- **Working graph site**: a site identified by statement syntax and occupied by a reference to the graph used within its scope.
- **Working graph**: the graph identified by the reference occupying the current working graph site.
- **Active context**: the catalog, working schema reference, and working graph site used to interpret names and execute statements.
- **Session context**: the context in which multiple GQL-requests of one session are executed consecutively.
- **Implementation configuration**: the set of implementation-defined choices, such as supported limits and supported reference forms, made explicit in the semantics.
- **Session options**: per-session settings, such as list-index behavior, that affect statement behavior.
- **List-behavior policy**: the session setting that determines offset or ordinal list indexing, negative-index behavior, and out-of-range behavior for list element references; it does not affect list unwinding.
- **Session state**: the session-local information, such as active context and options, that affects how statements are interpreted.

5.4 Values

The previous section located named objects in a store and explained how statements find them. We now need the vocabulary for the items that can be carried through evaluation as data. Some values are scalar, such as numbers or character strings; others are structured, such as lists and records. The section also introduces bags, because GQL keeps multiplicities in many intermediate results rather than silently collapsing duplicates. It ends with binding tables and references, which connect the value layer back to the GQL-objects introduced earlier. The value-level vocabulary is:

- **Base type**: a set of data types that have shared characteristics.
- **Primary base type**: the unique base type associated with a data type.
- **Data type descriptor**: a descriptor that identifies a data type, including its declared primary base type name and associated metadata.
- **Value type**: a data type comprising values.
- **GQL-object type**: a data type whose elements are GQL-objects.
- **Material value**: any value other than the null value.
- **Material data type**: a data type that does not contain the null value.
- **Comparable values**: two values that are capable of being compared.

- **Distinct values**: comparable values that are distinguishable by GQL's distinctness relation; two null values are not distinct, whereas a null value and a material value are distinct.
- **Nullable value type**: a value type that admits the null value in addition to its ordinary material values.
- **Immaterial value type**: the null type, which contains the null value, or the empty type, which contains no values.
- **Supported property value type**: a non-empty value type that an implementation accepts as the type of a property value.
- **Scalar type**: a type for a single simple value, such as an integer, string, Boolean value, date, time, or duration.
- **Numeric type**: a scalar type for numbers, such as integer, decimal, float, or double.
- **Truth value**: a value used by conditions: true, false, or unknown.
- **Boolean type**: a scalar type for truth values.
- **Character string type**: a scalar type for text.
- **Byte string type**: a scalar type for binary data.
- **Temporal type**: a scalar type for dates, times, date-times, or durations.
- **Vector type**: a value type for vector-shaped values where the implementation supports them.
- **Collection type**: a type for values that contain other values.
- **Bag (multiset)**: a collection that can contain the same item multiple times.
- **Multiplicity**: the number of occurrences of an item in a bag.
- **Bag cardinality**: the total number of occurrences in a bag, counting duplicates.
- **List value**: an ordered finite collection of values.
- **List value type**: a type describing list values, including the type of their elements, maximum size, and nullability.
- **Field**: one named component of structured data.
- **Group list value type**: a list type used when one variable contributes several values to one result field.
- **Record**: a finite set of named fields, where each field has a value.
- **Record type**: a type that states which fields a record has and what type each field may contain.
- **Closed record type**: a record type specified with an explicit, possibly empty, set of field types.
- **Unit record type**: the record type with no fields, used as a neutral empty row in some evaluations.

- **Binding table**: a primary GQL-object comprising zero or more, possibly duplicate, records of one record type.
- **Binding table type**: the GQL-object type of binding tables whose records have a specified record type.
- **Column**: a field position shared across all records in a binding table.
- **Ordered binding table**: a binding table whose ordering indication records an order of its record occurrences.
- **Unordered binding table**: a binding table whose ordering indication does not determine an order of its record occurrences.
- **Preferred column name sequence**: binding-table metadata that is either not set or is a permutation of the table's column names.
- **Effective column name sequence**: the preferred column name sequence when it is set, and otherwise the table's column names in ascending order.
- **Binding table descriptor**: the binding table type, an indication of whether the table is ordered or unordered, and a preferred column-name sequence that is either not set or is a permutation of the table's column names.
- **Working record**: the record in the current execution context whose fields provide scalar bindings during execution.
- **Working table**: the binding table in the current execution context whose records drive statement execution.
- **Binding variable**: an execution-context variable associated with an identified field of the working record or with the corresponding fields of records in the working table.
- **Dynamic union type**: a type that can contain values from several component types, useful when different branches of a query can produce different value kinds.
- **Constructed value type**: a value type built from other parts rather than supplied as a single primitive family.
- **Reference value**: a value that describes a globally resolved reference to a GQL-object, such as a graph, node, edge, or binding table.
- **Reference value type**: a type for reference values.
- **Graph reference value**: a reference value that points to a graph object.
- **Node reference value**: a reference value that points to a node.
- **Edge reference value**: a reference value that points to an edge.

5.5 Expressions

With values and references in place, we can define the language forms for operating on them. An expression evaluates to a value or reports an error, while a predicate is

the special case used as a condition. Because missing information is common in graph data, GQL uses three-valued logic, where a condition may be true, false, or unknown. The definitions below also introduce comparison and ordering behavior, runtime errors, and named functions. Function terminology appears here before the later routine section so that the built-in terms can be read without a forward dependency. The expression vocabulary is:

- **Expression**: a piece of statement syntax that computes a value or returns an error.
- **Typing**: the process of assigning and checking types for expressions, statements, properties, and results.
- **Static judgment**: a formal type-checking statement made before runtime evaluation.
- **Evaluation judgment**: a formal statement saying what value or error an expression evaluates to under a store state and the information available during evaluation.
- **Predicate**: an expression used as a Boolean test.
- **Unknown truth value**: the third truth value, represented by the null value, used when a Boolean result is unknown.
- **Three-valued logic**: a logic with true, false, and unknown, used to ensure that the null value has a precise influence on predicates.
- **Connective**: a Boolean operator such as conjunction, disjunction, or negation.
- **Short-circuiting**: the possible non-evaluation of an inessential expression part, whose result cannot affect the enclosing result; if evaluating that part would raise an exception, whether it is evaluated is implementation-dependent.
- **Comparison**: a test such as equality or sort precedence between two values.
- **Weak order**: a total preorder on the values of a base type that determines their relative order for sorting, while allowing distinct values to be equivalent with respect to that order.
- **Totalization**: the act of extending sort behavior so sorting has a defined policy for null values.
- **Null-intolerant predicate**: a predicate that returns unknown when one of its relevant inputs is null.
- **Function**: a named computation that returns a value.
- **Null-tolerant predicate**: a predicate or function that defines a non-null result for at least some null inputs.
- **Strict function**: a function that returns the null value if any of its arguments is the null value.
- **Runtime error**: an error that occurs during evaluation, such as numeric overflow, access through an invalidated reference, or an invalid path representation.
- **Error token**: the formal value used to carry information about an error result.

5.6 Schemas

Values, references, labels, and properties give us the ingredients for stored structures. Schemas describe which combinations of those ingredients are expected or allowed. In GQL, this includes the types of properties, the types of nodes and edges, and the graph types that collect such declarations into a larger contract. An open graph has no constraining graph type; a closed graph must contain only elements of the exact node and edge types specified by its constraining graph type and must satisfy that graph type's constraints. The final terms in the section describe type relationships and the fact that a schema can change over time. The schema terms are:

- **Property type**: a pair made of a property key and the type of values allowed for that key.
- **Node type**: a type whose nodes have exactly its label set and property-type set.
- **Edge type**: a type whose edges have exactly its labels and property types, its directionality, and its endpoint node types.
- **Element type**: either a node type or an edge type.
- **Label test**: a Boolean expression over the labels of a node or edge in a pattern; graph element type membership instead uses an exact label set.
- **Property signature**: the finite mapping from declared property keys to their property types for an element type.
- **Static property signature**: a property signature used during type checking before or independently of runtime values.
- **Node type descriptor**: in the reduced book-level representation, the local type name, label set, and property signature of a node type; the fixed primary-base-type name and preferred-name metadata of the full ISO descriptor are elided.
- **Relationship type**: the alternative standard term for an edge type; it is not an alternative term for an edge label.
- **Endpoint specification**: the ordered source and destination node types of a directed edge type, or the endpoint node-type set of an undirected edge type.
- **Edge type descriptor**: in the reduced book-level representation, the local type name, label set, property signature, directed or undirected kind, and endpoint specification of an edge type; the fixed primary-base-type name and preferred-name metadata of the full ISO descriptor are elided.
- **Structural consistency**: the compatibility requirement for element types that imply one another, covering their element kind, label sets, property names, and compatible property value types.
- **Endpoint structural consistency**: the corresponding structural compatibility requirement for the endpoint node types of edge types that imply one another.

- **Graph type**: a primary GQL-object comprising a node type set, an edge type set, and a constraint set; when named and held by a schema, it is a catalog object.
- **Label vocabulary**: the set of label names used in a given graph, graph type, or type-test context.
- **Key label set**: a set of labels that is an element of a graph type and may be associated with an element type of that graph type.
- **Keyed element type**: an element type associated with a key label set.
- **Graph type descriptor**: the metadata describing a graph type, including its primary-base-type and preferred-name metadata, node- and edge-type descriptor sets, key-label-set dictionaries, and named constraint set dictionary; for a cataloged graph type it is a catalog object descriptor.
- **Subtype**: a more specific type that can be used where a more general type is expected.
- **Subtyping**: the relation that says one type is more specific than another while still satisfying the broader type's expectations.
- **Inheritance**: a possible future schema-language extension for reusing type requirements; it is not part of the graph-type membership rules modeled in this edition.
- **Widening**: a safe type change that accepts all values accepted before, and possibly more.
- **Cast**: a conversion from one type to another type.
- **Implicit cast**: a cast or conversion the system applies automatically only where the GQL rules prescribe one; mere widening does not imply conversion.
- **Absent property**: a property key that is not present on a node or edge.
- **Graph schema**: a proposed future schema-language construct for describing allowed node types, edge types, graph types, properties, identity rules, and validation rules beyond the graph types modeled in this edition.
- **Schema state**: the current schema extracted from the catalog at a point in execution.
- **Schema versioning**: the modeling of schema changes as a sequence of schema states.
- **Schema evolution**: the process by which schema-changing commands produce a new schema from an old one.

5.7 Paths

Once nodes and edges have identities, GQL can talk not only about individual elements but also about routes through them. A path is such a route, represented as an alternating sequence of node and edge references. At the value level, the vocabulary describes how paths are represented, inspected, and constructed. It also requires the

referenced nodes and edges in a path element list to identify a valid route, independently of which graph is currently working. Finally, path modes specify whether nodes or edges may occur more than once along a path. The path vocabulary is:

- **Path in a graph**: an alternating sequence of graph elements where each edge connects the adjacent nodes in the sequence.
- **Path value**: a value representing an alternating sequence of node references and edge references.
- **Path value type**: a constructed value type specified by `PATH`; its descriptor records the primary base type `PATH VALUE DATA` and whether the null value is included.
- **Path element list**: the alternating list of nodes and edges stored inside a path value.
- **Path length**: the number of edge steps in a path value.
- **Path accessor**: an expression or operation that extracts information from a path value, such as its element list, nodes, edges, head, tail, or length.
- **Head of a path**: the first node in a path.
- **Tail of a path**: the last node in a path.
- **Malformed path**: a path representation whose path element list contains null, does not alternate node and edge references, or has an edge that does not connect its adjacent nodes.
- **Path value construction**: the creation of a material path value from a non-null graph-element-reference list that identifies a path.
- **Value-level path concatenation**: the operation that joins two path values when the last node reference of the first and the first node reference of the second have the same node as their referent.
- **Path-value realizability**: the property that a path value's references identify a valid path, independently of the current working graph.
- **Walk**: a path that may repeat nodes and edges.
- **Trail**: a path that may repeat nodes but not edges.
- **Simple path**: a path that does not repeat nodes, except that the first and last node may coincide in the allowed case.
- **Acyclic path**: a path with no repeated nodes at all.
- **Path mode**: the rule that selects whether a path is treated as a walk, trail, simple path, or acyclic path.
- **Mode filter**: the test that rejects candidate paths that do not satisfy the chosen path mode.
- **Computed path**: a path value produced during pattern evaluation rather than stored as a persistent graph object.

- **Stored path**: a persistent path object outside the GQL core, distinct from a computed path value.

5.8 Path Search

The preceding section described what a path is. The next question is how a statement asks the system to find paths that satisfy a description. Path patterns provide that description, and prefixes refine the search by selecting a mode or by choosing only certain paths for each pair of endpoints. Because a graph can contain many possible paths between the same two nodes, the model distinguishes the generation of candidates from the selective stage that keeps only the desired ones. The final terms in this section name the bags and partitions used to express that selection precisely. The search vocabulary is:

- **Path pattern**: a declarative description of admissible paths, optional binding for the whole path, and optional path search behavior.
- **Path variable**: a variable that binds the whole path value found by a path pattern.
- **Path binding**: a sequence, produced for one path pattern, whose entries associate variables or internal pattern markers with graph elements.
- **Anonymous path**: a path found by a pattern without exposing the whole path value as a named binding.
- **Path prefix**: the part of a path pattern that may specify a path mode and search behavior.
- **Search prefix**: the part of a path pattern that chooses which paths to retain for each endpoint pair.
- **Path restrictor**: a path-mode rule viewed as a filter over candidate paths.
- **Admissibility filter**: a filter that decides which candidate paths are acceptable before they become results.
- **Selective path search prefix**: a prefix that reduces candidate paths endpoint-by-endpoint, for example by choosing any path or the shortest paths.
- **Candidate set**: the set of candidate path bindings produced before a selective path-search prefix reduces it.
- **Endpoint partition**: the part of a candidate set containing path bindings with the same start and end nodes.
- **Length group**: the part of an endpoint partition containing paths that all have the same length.

5.9 Path Expressions

Path search needs a compact way to describe which edge steps and intermediate elements are allowed. Regular path expressions fill that role. They are close in spirit to

regular expressions over strings, but their units are traversal steps through a graph rather than characters in a word. The definitions below first name the expression forms and witnesses, then describe predicates and atoms used for single steps, and finally introduce the operators that combine, repeat, reverse, or guard expressions. This ordering prepares the reader for graph-pattern matching, where path expressions become part of larger patterns. Epsilon, inverse-step atoms, and general expression reversal below are auxiliary meta-language constructs used by the book; concrete GQL expresses traversal direction through its directional edge-pattern forms. The expression forms and supporting terms are:

- **Regular path expression**: an expression that describes paths by combining steps, tests, choices, sequencing, and repetition.
- **Path expression**: the concrete syntax form used to write a regular path expression inside a path pattern.
- **Path witness**: a concrete path that proves a regular path expression matched.
- **Element pattern**: a path-expression component that contributes a single node or edge step with optional tests and bindings.
- **Node/Edge predicate**: a Boolean test over nodes or edges.
- **Step predicate**: a test describing one allowed traversal step, including orientation, edge tests, and endpoint tests.
- **Step relation**: the graph relation that says which source node, edge, and destination node triples satisfy a step predicate.
- **Label atom**: a path-expression atom that matches an edge with a chosen label.
- **Predicate atom**: a path-expression atom that matches a step by evaluating a Boolean test.
- **Inverse step**: a path-expression step that follows an edge in the reverse direction.
- **Epsilon path expression**: a path expression that consumes no edge and stays at the same node.
- **Concatenation**: putting path expressions or paths one after another so the end of one connects to the start of the next.
- **Alternation**: a path-expression choice between alternatives.
- **Set alternation**: path-pattern alternation in which equivalent operand results are collapsed to one occurrence.
- **Multiset alternation**: path-pattern alternation in which distinct operand derivations remain distinguishable and may contribute multiple equivalent result occurrences.
- **Quantifier**: a path-expression suffix that controls optionality or repetition.

- **Restrictive path mode**: trail, simple, or acyclic mode; using such a mode for every unbounded quantifier guarantees a finite path-pattern result.
- **Path primary**: an element pattern, a parenthesized path-pattern expression, or a simplified path-pattern expression that forms the operand of path concatenation or quantification.
- **Bounded repetition**: a quantifier that repeats a path expression between a lower and upper bound.
- **Unbounded repetition**: a quantifier that allows arbitrarily many repetitions.
- **Reversal of expressions**: the operation that reverses a regular path expression so it matches the same paths in the opposite direction.
- **Guarded expression**: a path expression filtered by a local Boolean test over the path or its intermediate elements.
- **Well-formedness**: the static rule that an expression, pattern, or statement must obey before it can be evaluated.
- **Finiteness**: the property that evaluation produces a finite bag of results rather than an unbounded one.

5.10 Matching

Path expressions describe local movement, but GQL queries usually ask for larger shapes. A graph pattern combines element patterns and path patterns into a structure that can be matched against a graph. Matching produces assignments for variables, and those assignments must respect scoping rules: some variables are visible outside a component, while others remain local. This section therefore moves from the act of matching to exposure, reference degree, and the staged treatment of selective patterns. It ends with the distinction between matching that permits shared objects and matching that requires selected objects to remain distinct. The matching terms are:

- **Graph pattern**: a pattern that describes graph elements and paths to be matched in a graph.
- **Pattern matching**: the process of finding graph elements and paths that satisfy a graph pattern.
- **Node variable**: a variable declared by a node pattern whose value represents one or more bound nodes.
- **Edge variable**: a variable declared by an edge pattern whose value represents one or more bound edges.
- **Element variable**: a node variable or an edge variable.
- **Subpath variable**: a named pattern marker that is not bound to a value and cannot be referenced; it distinguishes operand derivations in path multiset alternation.

- **Graph pattern variable**: a path variable, subpath variable, or element variable.
- **Multi-path binding**: a tuple containing one path binding from the result of each path pattern in a graph pattern.
- **Match**: a multi-path binding that satisfies all restrictions applicable to its graph pattern.
- **Reduced match**: a match after auxiliary or anonymous parts have been removed or collapsed.
- **Temporary element variable**: an element variable introduced by a syntactic transformation, scoped to its innermost path term and removed before reduced matches are formed.
- **Match mode**: a graph-pattern restriction that applies to all path patterns simply contained in the graph pattern.
- **Repeatable elements match mode**: the match mode that imposes no filter against the same graph edge occurring in different edge-binding positions.
- **Different edges match mode**: the match mode that retains only multi-path bindings in which no graph edge occurs in two different edge-binding positions; matched nodes remain unrestricted.
- **KEEP clause**: a graph-pattern clause whose path-pattern prefix is distributed to every simply contained path pattern before the clause itself is removed.
- **Matching semantics**: the rules that explain how graph patterns produce matches and then binding tables.
- **Boundary variable**: a variable visible at the composition boundary of a pattern component and therefore available when components are combined.
- **Interior variable**: a variable local to a component of a path pattern.
- **Strict interior variable**: an interior variable that is not also a boundary variable.
- **Variable exposure**: the rule that says whether a variable is visible outside the component that introduced it.
- **Unconditional exposure**: the case where every reduced match contains the variable, as a value-bearing binding or as a non-value subpath marker.
- **Conditional exposure**: the case where some reduced matches may not contain the variable.
- **Not-exposed variable**: a variable that cannot be used outside its local component.
- **Graph pattern YIELD clause**: a clause that selects the exposed element and path variables that become output bindings, or specifies `YIELD NO BINDINGS`; a subpath variable cannot be yielded.

- **Graph-pattern binding table**: a syntactic construct consisting of a graph pattern optionally followed by a `YIELD` clause; its semantic operator transforms an incoming binding table into a binding table over the exposed and yielded binding variables.
- **Singleton variable**: a variable that contributes at most one value per reduced match.
- **Group variable**: a variable that can contribute a group of values rather than one value.
- **Effectively bounded group**: a group variable whose maximum number of values is bounded by the syntax.
- **Non-local reference**: a reference from outside the local component where a variable was introduced.
- **Effectively unbounded group**: a group variable whose number of values can grow without a syntactic upper bound; non-local references nevertheless remain restricted to singleton degree.
- **Reference degree**: the classification of a referenceable binding variable as singleton, effectively bounded group, or effectively unbounded group; exposure is classified separately as unconditional, conditional, or not-exposed.
- **Path-variable degree**: the rule that a declared path variable is a singleton inside its declaring path pattern, even if a later optional operation produces a null output value.
- **Propagation rule**: a rule that explains how variable exposure and reference degree change through concatenation, alternation, optionality, and repetition.
- **Questioned path-pattern primary**: a path primary followed by `?`; its zero-occurrence branch makes exposed singleton-variable bindings conditional, while group-variable exposure is preserved.
- **Selective path pattern**: a path pattern using a selective path search prefix.
- **Local generation**: the first matching stage, where candidate path bindings are produced with local filters but before selective search.
- **Selective stage**: the matching stage where a selective path search prefix keeps only selected paths inside each endpoint partition.
- **Global filtering**: the stage where non-local conditions filter the combined bindings.
- **Reduction**: the stage that removes temporary symbols and anonymous structure from raw matches.
- **Homomorphic matching**: an informative graph-theoretic analogy in which different pattern variables may map to the same graph object if the structure still fits; it is not an additional GQL match mode.

- **Isomorphic matching**: an informative graph-theoretic analogy in which selected pattern variables map to distinct graph objects; it is not an additional GQL match mode.
- **Injectivity guard**: an explicit test that prevents selected variables from being mapped to the same graph object.
- **Explicit guard**: a test added by the user or semantics to rule out unwanted matches.

5.11 Algebra

Matching produces bindings, but a query still has to combine, filter, extend, and project them. The algebra in this section treats binding tables as the central intermediate objects. Because GQL follows bag-style behavior in many places, multiplicity is introduced before the table operators. The first terms explain how duplicates are counted or collapsed; the middle terms describe table headers and compatibility; the last terms give the main row-transforming operations, including joins and optional matching. This prepares the ground for full query blocks, which assemble these operators into complete statements. The algebraic terms are:

- **Result multiplicity**: the number of times the same binding appears because several matches or derivations lead to it.
- **Bag semantics**: semantics where duplicates are kept and counted.
- **Set semantics**: semantics where duplicate results are collapsed to one occurrence.
- **Bag union**: an operation that combines bags by adding multiplicities.
- **Bag difference**: an operation that subtracts right-side multiplicities from left-side multiplicities without going below zero.
- **Bag intersection**: an operation that keeps the minimum multiplicity found on both sides.
- **Setification**: the operation that collapses every positive multiplicity to one.
- **Unit table**: the binding table with one empty record, used as a neutral starting input.
- **Empty binding table**: a binding table of cardinality zero, distinct from the unit binding table.
- **Header**: the set of column names and their value types for a binding table.
- **Column name-equal**: a property that several binding tables have the same column names.
- **Column-combinable**: a property that corresponding columns have value types that can be combined safely.
- **Common type**: a combined type to which operands are coerced before union, difference, or intersection behavior is applied.

- **Selection**: an operator that keeps only rows whose test evaluates to true.
- **Projection**: an operator that keeps or computes selected output fields and discards the rest.
- **Extend**: an operator that adds new computed fields to each row.
- **Join**: an operator that combines rows from two binding tables when their shared fields are compatible.
- **Natural join**: a join based on equal values in shared field names.
- **Natural joinability**: the relation between two records whose common fields have equal values under GQL value equality, including two common fields that both contain the null value.
- **Name-compatible join**: a natural join that relies on shared column names agreeing.
- **Theta-join**: a join followed by an explicit Boolean guard.
- **Guarded join**: a join that keeps only row pairs passing an additional test.
- **Left outer join**: a join that keeps left-side rows even when the right side has no match, filling right-side fields with null values.
- **Optional match**: matching that preserves input rows when the optional pattern finds no extension.
- **Correlated optional expansion**: per-input-record optional evaluation that returns its compatible extensions when any exist, and otherwise returns the input record padded with null values for newly introduced columns.
- **Padding**: the insertion of null values for right-side fields when an outer join finds no right-side match.
- **List unnesting**: the `FOR` operation that emits one extended record for each element occurrence of an input list.
- **Semi-join**: a join-like operation that keeps left rows having at least one matching right row.
- **Anti-join**: a join-like operation that keeps left rows having no matching right row.

5.12 Queries

The previous section described the operators available for transforming binding tables. A query block arranges those operators into a unit with its own patterns, filters, and result formation. Subqueries then allow one query to be evaluated inside another, either independently or with access to the current outer row. The environment terms explain what information is available during such evaluation and how local names avoid accidental collisions with outer names. The section ends with composition rules for whole queries, where query parts are combined and duplicate-control choices determine whether repeated rows remain visible. The query-level terms are:

- **Query block**: a unit of query evaluation containing graph patterns, algebraic operators, and a final projection.
- **Subquery**: a query evaluated inside another query.
- **Uncorrelated subquery**: a subquery whose result does not depend on the current outer row.
- **Correlated subquery**: a subquery that uses values from the current outer row.
- **Relational subquery**: a subquery used as a bag of rows.
- **Scalar subquery**: a `VALUE` query whose nested query has one result field and is statically constrained, by `LIMIT 1` or by summarizing all input records into one result without `GROUP BY`, to produce at most one record; an empty result yields null and a singleton result yields its sole value.
- **Row-producing subquery**: a subquery that contributes rows to the surrounding algebra.
- **Environment**: the information available during evaluation, such as the current graph, outer bindings, and callable registries.
- **Static environment**: the compile-time or checking-time mapping from names to their meanings.
- **Dynamic environment**: the runtime information, especially the current outer row, used during correlated evaluation.
- **Row environment**: the mapping from field names in the current row to their values.
- **Global environment**: the mapping for global names such as catalog objects, named graphs, types, and callable routines.
- **Evaluation environment**: the pair of global and row environments used to evaluate expressions.
- **Variable hygiene**: rules that prevent accidental capture or collision of local and outer variable names.
- **Shadowing**: a situation where a local name would hide an outer name.
- **Pipelining**: the passing of one binding table into the next statement or query block.
- **Primitive query statement**: a basic query step such as pattern matching, filtering, or result formation.
- **Linear query statement**: a sequence of query steps evaluated left-to-right over a working table.
- **Whole-query denotation**: the meaning of a complete query obtained by composing all its blocks and operators.
- **Composite query**: a query built by combining operands with set-like operators or fallback behavior.

- **Query part**: one operand inside a composite query.
- **Set operator**: an operator that combines compatible query results using union, difference, or intersection behavior.
- **Left-associative composition**: the rule that a chain of operators is evaluated from the left unless parentheses say otherwise.
- **Fallback query conjunction**: the `OTHERWISE` query conjunction, which selects the first non-empty operand result in a left-associative chain, or the empty combined-type result if every operand result is empty; it is not a set operator and does not provide short-circuit evaluation, because the query-part results are obtained before that selection.
- **Duplicate control**: the rule deciding whether duplicates are preserved or collapsed.
- **Set quantifier**: a duplicate-control rule that decides whether duplicates are preserved or removed.
- **Deduplication**: the removal of duplicate rows by keeping one occurrence.

5.13 Result Shaping

Once a query has produced rows, the language still has to shape what the user actually sees. Ordering adds a presentation order without changing the underlying bag of rows. Exact limiting and offsetting select a contiguous part of that ordered sequence, while approximate limiting may select an implementation-dependent ordered subset, which is useful when a result is too large to return at once. Grouping and aggregation move in a different direction: they partition rows and summarize many input values into fewer output values. The final term ties these operations to the result-forming part of a query. The result-shaping terms are:

- **Ordering**: the act of adding a row order to a binding table result.
- **Ordering metadata**: metadata recording the emission order of rows without changing the underlying bag.
- **Sort key**: an expression used to decide row order.
- **Sort direction**: the choice between ascending and descending order.
- **Nulls policy**: the rule deciding where null values appear in sorted results.
- **Sort specification**: a sort key together with explicit or implicit choices for sort direction and null ordering.
- **Sort specification list**: a non-empty list of sort specifications used for ordering.
- **Collation**: rules for comparing character strings during ordering.
- **Peer ordering**: the implementation-dependent ordering among records that are not distinct with respect to every sort specification, including its null-ordering policy.

- **Slicing**: using `OFFSET` and `LIMIT` to retain selected records from an ordered row sequence; the retained records form a contiguous suffix or prefix only where the applicable exact operation requires it.
- **Exact LIMIT**: limiting that retains the first requested number of records of the ordered input.
- **Approximate LIMIT**: limiting that retains at most the requested number of records chosen by the implementation and need not retain a prefix, while preserving the selected records' relative input order.
- **Pagination**: using ordering and slicing to return only a page or selected portion of a larger result.
- **Grouping**: partitioning rows so each partition can be summarized.
- **Grouping key**: an expression whose value decides which group a row belongs to.
- **Grouping function**: the function that maps rows to grouping keys.
- **Grouping set**: one selected set of grouping keys used to form groups.
- **Aggregate function**: a function that summarizes many input rows or values into one output value.
- **Aggregator**: the semantic object that defines how an aggregate function consumes inputs and produces a result.
- **Global group**: the single group used when aggregation is performed without explicit grouping keys.
- **Post-aggregation filtering**: filtering that happens after grouped summaries have been computed.
- **Primitive result statement**: the part of a query that forms the final visible result, including return items, grouping, distinctness, ordering, and pagination.

5.14 Updates

Once query results are available, GQL moves beyond retrieving graph data and begins modifying the graph itself. An update in GQL still depends on the earlier machinery: it may first use patterns and expressions to find the elements that should change, and only then produce write operations. For that reason, the section begins with data-modification statements and the graph chosen for modification. It then introduces insert and delete forms, followed by the abstract update pipeline that grounds templates into concrete operations. The final terms describe how writes are staged, combined, rejected when they conflict, and eventually represented as primitive mutations. The update vocabulary is:

- **Data-modification statement**: a statement that inserts, deletes, or updates graph data.

- **Data-modification primitive**: a basic write step such as insert, set, remove, delete, or a data-modifying routine call.
- **Linear data-modification statement**: a pipeline that combines graph selection, query steps, and data-modification primitives.
- **Graph-selection header**: an optional front part of a data-modification statement that chooses the graph to modify.
- **Graph expression**: an expression that evaluates to a graph reference.
- **Read-then-update composition**: an update pattern where a query first finds bindings and an update then uses those bindings to change data.
- **Insert graph pattern**: the insert-specific pattern that describes the nodes and edges to create.
- **Insert path pattern**: a path-shaped insert pattern that may create nodes, edges, labels, properties, and bindings together.
- **Insert element pattern**: a node or edge pattern inside an insert graph pattern.
- **Bound insert element pattern**: an insert element pattern that refers to an element already bound instead of creating a new one.
- **System-generated identifier**: an identifier supplied by the system when the user omits a variable name, not retained as a user-visible output column.
- **Identifier allocation**: the creation of fresh global object identifiers for new nodes, edges, or copies.
- **Incident edge**: an edge connected to a node as either source or destination.
- **Plain delete**: deleting an edge directly, or deleting a node only when every edge incident on that node is also designated for deletion by the same statement.
- **Dependent-object error**: an error raised when an operation cannot proceed because another object still depends on the target.
- **Endpoint reattachment**: changing an edge's endpoints, modeled as delete plus recreate rather than an in-place update.
- **Update pipeline**: a read-then-update form where matched bindings are used to produce updates.
- **Read stability**: the rule that the read part of an update is evaluated against a stable view before writes are applied.
- **Update context**: the graph, bindings, stable read view, and options needed to interpret an update.
- **Target reference**: the concrete node or edge selected as the target of an update.
- **Update atom**: one small abstract update action, such as setting one property or adding one label.

- **Update template**: an update atom pattern before variables and expressions have been evaluated for a particular binding.
- **Ground operation**: an update operation whose target and value have already been evaluated.
- **Grounding**: the step that turns an update template into concrete ground operations for one binding.
- **Write set**: the collection of ground operations produced by an update before they are applied.
- **Delta**: the staged change that will transform one graph or store state into another if validation succeeds.
- **Coalescing**: combining several writes into one canonical set of changes when they are compatible.
- **Write conflict**: a situation where multiple writes cannot be combined without ambiguity or rule violation.
- **Multiple property assignments**: the case where one statement produces two or more assignments to the same property of the same element, regardless of whether the values are equal; the implementation either raises 22G0M or chooses one assignment, according to its documented choice.
- **Idempotent label addition**: adding a label in a way where repeated additions have the same final result as one addition.
- **Primitive mutation**: one elementary graph change, such as inserting a node, deleting an edge, setting a property, or removing a label.
- **Set property**: a primitive mutation that assigns a value to a property key.
- **Delete property**: a primitive mutation that removes a property key from an element.
- **Add label**: a primitive mutation that attaches a label to a node or edge.
- **Remove label**: a primitive mutation that removes a label from a node or edge.

5.15 Constraints

Updates can create useful new facts, but they can also break assumptions that the schema is meant to protect. Constraints express those assumptions as rules that a graph must satisfy. The first terms describe the general shape of a constraint: what population it checks, how that population is selected, and what requirement must hold. The middle of the section focuses on keys and uniqueness, because those rules are central to identifying elements. The final terms describe how constraints are stored, induced from catalog declarations, checked after staged changes, and reported when validation fails. The constraint terms are:

- **Integrity constraint**: a constraint used to prevent invalid graph states.
- **Registered subject**: a graph whose permitted characteristics or contents are constrained by a particular integrity constraint.
- **Requirement**: the rule a constraint imposes on every in-scope binding.
- **Constraint specification**: the formal description of what population is checked and what requirement it must satisfy.
- **Constraint body**: the part of a constraint specification that determines its scope and requirement without its constraint name.
- **Constraint normal form**: an implementation-defined, deterministic, idempotent representation of a constraint body; a specification normal form additionally retains the constraint name.
- **Body-equivalence**: the relation between constraint bodies whose deterministic normal forms are equal.
- **Constraint graph pattern**: a graph pattern used to select the elements that a constraint checks.
- **Constraint scope**: the set of graph elements selected for checking by a constraint.
- **Scope predicate**: a predicate that says whether a graph element is in the scope of a constraint or type rule.
- **Key value component**: the main element variable, an endpoint variable of an edge scope, or a property reference rooted at one of those variables and selected as one component for identifying constrained elements.
- **Key value component specification**: the ordered sequence of zero or more key value components used to identify constrained elements.
- **Constraint descriptor**: a descriptor containing a constraint's name, registered subjects, deterministic scope, and key value component specification.
- **Key**: a sequence of zero or more key value components that identifies constrained elements.
- **Key specification**: a deterministic constraint graph pattern together with its key value component specification.
- **Key tuple**: the tuple obtained by evaluating the key value component specification for one in-scope binding, used to compare in-scope elements.
- **Non-distinctness**: the GQL value relation in which two null values are non-distinct.
- **Null-containment**: the recursive property of a value or composite value containing the null value.
- **Key constraint**: a constraint requiring in-scope key values to be recursively not null-containing and requiring elements with non-distinct key tuples to be identical.

- **Uniqueness constraint**: a constraint requiring in-scope elements not to share non-distinct key values.
- **Graph-type conformance**: the requirement that every element of a closed graph is of an element type in its constraining graph type and that the graph satisfies that type's constraints.
- **Direct graph constraint set**: the constraints defined directly in a graph's own constraint dictionary.
- **Active constraint set**: the constraints registered for the active graph, considered collectively, from its own dictionary and its optional constraining graph type.
- **Semantic constraint clause**: one formal clause that a graph must satisfy.
- **Satisfaction relation**: the relation saying that a graph satisfies a constraint or set of constraints.
- **Validation predicate**: the check that a graph after applying a delta still satisfies all relevant constraints.
- **Dynamic constraint validation**: the validation, by a data-modifying statement's General Rules, of the graph's direct constraints after a provisional graph has been produced.
- **Constraint enforcement**: the separate checks of direct graph constraints and graph-type conformance that must succeed before a data-modifying statement succeeds.
- **Integrity constraint violation**: a reported failure raised when graph data breaks an integrity constraint.

5.16 Effects

The update and constraint sections describe what can happen at runtime. An effect system gives a static approximation of that behavior before execution begins. It records what a statement may write, combines such information across larger statements, and checks whether different writes may interfere. This is necessarily conservative: it may prevent an optimizer reordering whose equivalence has not otherwise been established, but it does not introduce an additional rejection or warning rule for GQL programs. The terminology below therefore separates primitive effects, algebraic combination, soundness, and rules that prevent unsafe writes through missing bindings. The effect terms are:

- **Effect**: an abstract description of what a statement may change.
- **Effect system**: the book's static rules that approximate modeled graph-data writes of query and data-modifying statements.
- **Primitive effect**: the effect associated with one primitive update action.
- **Effect algebra**: the algebra used to combine primitive effects and compare them.

- **Statement effect typing**: the static judgment that assigns an effect to a query or data-modifying statement in the book's limited model.
- **Write-purity**: the property of having an empty static write effect; it does not by itself imply independence from evaluation order or context, or result determinism.
- **Effect soundness**: the guarantee that static effects safely over-approximate runtime changes.
- **Static conflict approximation**: a conservative static check for possible conflicts between effects.
- **Optimizer ordering restriction**: a meta-level restriction that prevents an unproved reordering of potentially conflicting writes.
- **No unbound writers**: the rule that an update must not write through variables that may be unbound.
- **Constraint-preserving execution**: a data-modifying execution whose post-state passes mandatory direct-constraint validation and the separate graph-type-conformance check before the statement succeeds.

5.17 Routines

Expressions introduced functions that return values. GQL also has procedures that can be invoked by statements and may expose outputs; named procedures are primary catalog objects, whereas functions are not modeled as such here. The order follows the way such a routine is understood: first the procedure kinds, then their declared interface, then the catalog declaration and body, and finally invocation and resolution. This prepares the transition to transactions, where statements and calls are executed inside larger units of work. The routine vocabulary is:

- **Query procedure**: a procedure whose execution does not cause side effects.
- **Data-modifying procedure**: a procedure whose execution may cause data-modifying side effects only.
- **Catalog-modifying procedure**: a procedure whose execution may cause catalog-modifying side effects, either alone or together with data-modifying side effects.
- **GQL procedure**: a procedure written in GQL.
- **External procedure**: a procedure provided through an implementation-defined mechanism.
- **Inline GQL procedure**: a GQL procedure specified directly at its invocation site rather than as a named catalog object.
- **Signature**: the declared input and output types of a function or procedure.
- **Declaration**: a catalog entry that introduces a named procedure or primary object.
- **Parameter**: an input name and type expected by a function or procedure.

- **Named procedure descriptor**: the mandatory and optional parameters, result type, and side-effect kind of a named external procedure.
- **Procedure body**: the variable definitions and statement block that define an inline GQL procedure.
- **Procedure invocation**: the act of executing a procedure call.
- **Reference resolution**: the process that follows an explicit catalog parent or the implicit current-schema parent and verifies the referenced object's kind.

5.18 Transactions

Statements and routine calls do not always stand alone. A transaction groups work so that a sequence of reads and writes can be completed together or abandoned together. This section starts with commands and characteristics that establish the transaction's mode, then describes successful and unsuccessful completion. It then introduces visibility and the standard's serializable execution model, followed by snapshot and conflict-policy terminology used only for explicitly identified implementation extensions. The final terms distinguish the core isolation guarantee from those extensions. The transaction terms are:

- **Transaction**: a bounded unit of work whose changes complete together or are rolled back together.
- **Transaction model**: the rules for starting, sequencing, committing, rolling back, and isolating transactions.
- **Transaction control command**: a command that starts, commits, or rolls back a transaction.
- **Transaction characteristic**: an option such as access mode or read behavior that affects a transaction.
- **Transaction access mode**: the setting that says whether a transaction is read-only or read-write.
- **Read-only transaction**: a transaction mode that rejects catalog-modifying and data-modifying statements.
- **Read-write transaction**: a transaction mode that permits writes if all other checks also pass.
- **Savepoint**: a point inside a transaction that may support undoing part of the work.
- **Commit**: a successful transaction completion that makes all staged changes part of the committed state.
- **Rollback**: a successful cancellation of the current transaction's changes, leaving none of them committed.

- **Atomicity**: the recovery property ensuring that a successful commit has all of the transaction's effects, while a successfully processed rollback leaves none of them committed.
- **Abort**: a failed transaction outcome accompanied by a rollback attempt; if transaction termination cannot be processed successfully, the catalog and data state is indeterminate.
- **Statement sequencing**: the rule that statements inside a transaction run in program order.
- **Visibility**: the rule that decides which changes a statement or transaction can observe.
- **Provisional change**: a transaction change made before commit and therefore not yet fully visible as committed state.
- **Snapshot**: a fixed view of the graph or store used by a transaction or statement in an implementation extension outside the serializable core model.
- **Transaction snapshot**: a snapshot fixed for a whole transaction by such an implementation extension.
- **Statement snapshot**: a snapshot fixed at a statement boundary by such an implementation extension.
- **Isolation level**: an implementation setting outside the serializable core model that controls concurrent visibility and anomalies.
- **Serializability**: the guarantee that concurrent committed transactions have the same observable effect as some serial order.
- **Write-write conflict**: a conflict caused by concurrent transactions trying to write overlapping entities or fields.
- **First-committer-wins**: a snapshot-isolation policy outside the serializable core model where the first committing transaction succeeds and a conflicting later commit fails.

5.19 Authorization

Transactions determine when changes become visible, but they do not decide who is allowed to see or make those changes. Authorization supplies that control layer. The section begins with the actors that participate in access control, then introduces roles, privileges, and grants. The middle terms explain how direct and role-based grants are combined into effective privileges and how statements produce obligations that must be checked. The final terms describe object availability and authorization-identifier-indexed semantics, which connect authorization back to query results and update execution. The authorization vocabulary is:

- **Security**: the broad concern of controlling who can see or change data and catalog objects.
- **Authorization**: the rules that determine who may reference objects and perform operations.
- **Access control**: authorization applied to actors, objects, and operations.
- **Principal**: a user or other actor identified for authorization purposes.
- **Role**: a named grouping used to organize permissions.
- **Privilege**: permission to perform a kind of operation on an object.
- **Authorization identifier**: an identifier that represents a principal together with the privileges under which associated requests execute; one principal may have more than one such identifier.
- **Privilege kind**: the category of permission, such as read, write, execute, or administer, as chosen by an implementation.
- **Privilege level**: the strength of a privilege within the implementation's privilege ordering.
- **Privilege poset**: the partial order saying when one privilege is strong enough to imply another.
- **Authorization target**: a complete GQL-object to which a privilege applies in the core model, such as a graph, graph type, binding table, or named procedure.
- **Grant**: a fact that gives a privilege on an authorization target to an authorization identifier or role.
- **Grantor**: the authorization identifier that gives a privilege.
- **Grantee**: the authorization identifier or role receiving a privilege.
- **Authorization state**: the set of role and grant facts governing authorization.
- **Effective privilege**: the combined privilege an authorization identifier has on an authorization target after considering direct grants and the grants of its roles.
- **Privilege obligation**: a required privilege produced by type checking or statement analysis.
- **Obligation set**: the set of privilege obligations that must be satisfied before a statement may run.
- **Obligation judgment**: a typing-style judgment that returns both a type and the privileges required by a statement.
- **Authorization check**: the check, before statement evaluation begins, that the effective privileges of the session authorization identifier dominate every privilege obligation of the statement.

- **Object availability**: the implementation-defined relation determining which complete authorization targets an execution context may reference or resolve; availability of a graph does not mean availability of an induced subgraph.
- **Authorization-identifier-indexed semantics**: query semantics relative to the session authorization identifier: ordinary semantics is used only when every referenced object is available and every obligation is satisfied; otherwise exception 42000 is raised.
- **Write check**: an authorization or transaction-mode check performed before a modifying operation takes effect.

5.20 Diagnostics

The last section describes how execution outcomes are reported. Every preceding topic can contribute to such an outcome: expressions may fail, updates may violate constraints, transactions may conflict, and authorization checks may reject an operation. Diagnostics therefore provide a common vocabulary for successful outcomes, warnings, no-data cases, informational conditions, and failures. The section begins with status codes and conditions, then moves to status objects and the result forms carried by execution outcomes. It ends with broad error classes and the rules for choosing a reported status, closing the chapter with the mechanism by which the language communicates success or failure to the user. The diagnostic terms are:

- **Diagnostics**: structured reporting of success, warnings, no-data outcomes, informational conditions, and failures.
- **Status code**: a code used by the diagnostic model to identify an execution situation.
- **Status domain**: the set of all status codes used by the diagnostic model.
- **Condition**: a situation reported by a status code.
- **Status class**: the first part of a status code, identifying a broad family of conditions.
- **Status subclass**: the more specific part of a status code.
- **Completion condition**: a condition indicating normal request completion, a non-fatal issue, or a no-data outcome.
- **Exception condition**: a reported failure condition rather than normal completion.
- **Transaction rollback condition**: a reported failure condition used when a transaction cannot complete successfully.
- **Successful completion**: a completion condition indicating that the request completed without issues.
- **Warning**: a completion condition that reports some issue, but is non-fatal.
- **No-data condition**: a completion condition saying that no data was produced where that matters.

- **Informational condition**: a secondary condition used for extra status information rather than the primary outcome.
- **Diagnostic record**: structured diagnostic information associated with a reported condition.
- **GQL-status object**: an object representing a condition and its diagnostic information, comprising a GQLSTATUS, a status description, a diagnostic-information record when supported, and an optional nested GQL-status object.
- **Regular result**: a value or binding table produced by successful execution of an operation.
- **Omitted result**: a result indicating successful execution of an operation that produced no value.
- **Execution outcome**: the execution-context component comprising a primary GQL-status object, a possibly empty list of additional GQL-status objects, and an optional regular or omitted result.
- **Successful outcome**: an execution outcome that records a result and no exception condition.
- **Failed outcome**: an execution outcome that records an exception condition and no result.
- **Primary status code**: the GQLSTATUS field of the primary GQL-status object attached to an execution outcome.
- **Additional status code**: the GQLSTATUS field of an additional GQL-status object carrying secondary diagnostic information.
- **Result sum type**: the book's meta-level result domain that distinguishes successful results from failures.
- **Short-circuit failure propagation**: the rule that the first failure in a compound statement stops later evaluation and is propagated upward.
- **Standard-defined code**: a status code assigned by the standard.
- **Implementation-defined code**: a status code or subclass chosen by an implementation within allowed ranges.
- **Status precedence**: the rules that decide which condition becomes primary when several conditions could be reported.
- **Error class**: a broad category of errors, such as static, dynamic, concurrency, constraint, or resource errors.
- **Static error**: an error detected before execution, such as an unbound variable or schema mismatch.
- **Dynamic error**: an error detected during execution, such as numeric overflow, access through an invalidated reference, or a malformed path.

- **Concurrency error**: an error caused by isolation, conflicts, or concurrent execution.
- **Constraint error**: an error caused by violating an integrity or graph-type constraint.
- **Resource error**: a standard-defined or implementation-defined error caused by a resource limitation, such as timeout or memory exhaustion.
- **Diagnostic choice**: when several raised exception conditions have the same highest priority, the implementation-dependent choice of which one becomes the status of the current request outcome.

Chapter 5 — Takeaways

- **Purpose:** Established the shared vocabulary used by the semantic chapters before the formal development begins.
- **Introduced:** Core terms for foundations, graphs, context, values, expressions, schemas, paths, patterns, query composition, updates, constraints, transactions, authorization, and diagnostics.
- **Key distinctions:** Conceptual definitions versus full formal treatment; values versus GQL-objects; stored structures versus computed results; static checks versus runtime outcomes.
- **Consequences:** Reduced forward references in later chapters and gave readers a compact map of how the model's components fit together.
- **Bridge:** Prepared the transition from the introductory example to the abstract data model and pattern semantics.

6 Data Model

This chapter fixes the abstract data model that underlies the semantics of GQL and serves as the common reference point for later chapters. It introduces the basic universes of identifiers, types, and values, defines the property-graph structure used for stored graphs, and explains how paths, lists, and records are treated as first-class values. The presentation is independent of any particular physical representation and is parameterized by the conformance options of the GQL standard (for example, with respect to supported property value types).

6.1 Universes and Notation

Let

$\mathbb{A}$ be a universe of identifiers used in GQL, including names of directories, schemas, catalog objects, labels, properties, and columns,

$\mathbb{L} \subseteq \mathbb{A}$ be a universe of label names,

$\mathbb{K} \subseteq \mathbb{A}$ be a universe of property names,

$\mathbb{I}$ be a universe of globally unique identifiers of graph elements,

$\mathbb{T}$ be a universe of types (static, dynamic, and base types),

$\mathbb{V}$ be a universe comprising material values and the distinguished null value null.

The abstract three-valued truth carrier used throughout is $\mathbb{B}_3 = \{\top, \bot, \mathsf{U}\}$; its connection to GQL Boolean values is defined in Section 8.3. Types and their characteristics (material/nullable/immaterial) as well as the naming of types and base types are defined by the GQL specification. Availability of the null and empty immaterial value types and explicit nullability follows the standard's conformance options; the empty type contains no value and therefore contributes no additional element to $\mathbb{V}$. No cardinality assumption is imposed on these ambient universes; by contrast, the node and edge carriers of every concrete graph are finite, as is every binding table used below.

Notational convention. We reserve blackboard-bold symbols such as $\mathbb{V}$ for *universes* (e.g., the universe of values), and we use plain symbols such as V only for *graph-specific carriers* (e.g., the node set of a concrete graph).

A partial function is written $f : A \rightharpoonup B$. The powerset of X is $\mathcal{P}(X)$. Let $\mathbb{N}$ denote the set of natural numbers. A multiset (bag) over X is identified with a multiplicity function $B : X \to \mathbb{N}$. The set of all such multisets is denoted $\mathcal{M}(X)$, and its finite-support

subset is denoted $\mathcal{M}_{\text{fin}}(X)$. We also write $\mathsf{B}(X) := \mathcal{M}_{\text{fin}}(X)$ when a finite bag is used as a meta-level type. Pointwise bag sum is written $B_1 \uplus B_2$, and $\biguplus_i B_i$ denotes its indexed iteration. For bags $B', B \in \mathcal{M}(X)$, B' is a *submultiset* of B, written $B' \sqsubseteq_{\mathsf{bag}} B$, iff $B'(x) \leq B(x)$ for every $x \in X$. For a bag B, $\text{supp}(B) := \{x \mid B(x) > 0\}$ is its support, and $\mathbf{1}[P]$ is 1 when proposition P holds and 0 otherwise.

Finite sequences over X are denoted X^*. *Binding-table records use closed (material) record types:* a record type is a *total* function $\tau_{\text{rec}} : \text{fields}(\tau_{\text{rec}}) \to \mathbb{T}$, where $\text{fields}(\tau_{\text{rec}}) \subseteq \mathbb{A}$ is its declared field set; a record over τ_{rec} is a *total* assignment $r : \text{fields}(\tau_{\text{rec}}) \to \mathbb{V}$ (each declared field has a value; null is a value of an appropriate null-containing type). The set of well-typed records inhabiting τ_{rec} is

$$\text{Rec}_{\tau_{\text{rec}}} := \{\, r : \text{fields}(\tau_{\text{rec}}) \to \mathbb{V} \mid r(n) \text{ has type } \tau_{\text{rec}}(n) \text{ for every } n \in \text{fields}(\tau_{\text{rec}}) \,\}.$$

To retain the metadata of the GQL binding-table descriptor, we model a binding table as

$$T = \langle \tau_{\text{rec}}, B, \mathit{ord}, \mathit{pcs} \rangle, \qquad B \in \mathcal{M}_{\text{fin}}(\text{Rec}_{\tau_{\text{rec}}}),$$

where *ord* is either unordered or determines an ordering of the record occurrences of B, and *pcs* is either notSet or a preferred permutation of the field names of τ_{rec}. The effective column-name sequence is *pcs* when a preferred sequence is set and, otherwise, is the field names of τ_{rec} in ascending order. We write BT for this domain and $\text{bag}(T) = \langle \tau_{\text{rec}}, B \rangle$ for the bag projection used by order-insensitive algebra. The unit binding table is the unordered table with the unit record $\langle\rangle$, an empty record type, and no preferred column-name sequence.

6.2 Property-Graph Structure

The term property graph has been used in the literature for several closely related graph models whose formalizations are not identical (Angles 2012; Angles 2018). Depending on the formalization, differences may concern, among other things, element identities, the cardinality and applicability of labels and properties, edge direction, and the treatment of parallel edges (Angles 2018; Francis et al. 2018; Deutsch et al. 2022). In this book, the term is used in a deliberately broad sense, while the concrete model adopted below is closely aligned with the property-graph data structures defined by the GQL standard (International Organization for Standardization 2024). We refer to an instance of this model as a GQL graph.

The tuple below models only a graph's structural contents; its optional constraining graph type and graph constraint dictionary are descriptor-level information modeled by the graph entry in Section 15.5. A GQL graph is modeled as a 9-tuple

$$G \;=\; \langle V, E, \lambda_V, \rho_V, \lambda_E, \rho_E, \text{src}, \text{dst}, \text{dir} \rangle,$$

where:

$V \subseteq \mathbb{I}$	is a finite set of globally identifiable nodes,
$E \subseteq \mathbb{I}$	is a finite set of globally identifiable edges (a multigraph is permitted),
$V \cap E = \varnothing$	(node and edge identities are disjoint),
$\lambda_V : V \to \mathcal{P}(\mathbb{L})$	assigns to a node its set of labels,
$\rho_V : V \to (\mathbb{K} \rightharpoonup \mathbb{V})$	is the node property map (names of node labels and node properties are in separate namespaces, so a label and a property may share the same identifier on a node),
$\lambda_E : E \to \mathcal{P}(\mathbb{L})$	assigns to an edge its set of labels,
$\rho_E : E \to (\mathbb{K} \rightharpoonup \mathbb{V})$	is the edge property map (names of edge labels and edge properties are in separate namespaces, so a label and a property may share the same identifier on an edge),
$\mathrm{src}, \mathrm{dst} : E \to V$	yield the endpoints of an edge,
$\mathrm{dir} : E \to \{\rightarrow, \leftrightarrow\}$	encodes directedness (directed/undirected).

When $\mathrm{dir}(e) = \leftrightarrow$, $\mathrm{src}(e)$ and $\mathrm{dst}(e)$ are only a representational ordering of the two endpoints; they do not assign source and destination semantics to an undirected edge. Loops are allowed (edges $e \in E$ with $\mathrm{src}(e) = \mathrm{dst}(e)$), as are mixed graphs and multigraphs; each property name is unique within its carrier (node/edge), and property values belong to the supported value types. This abstract structure is instantiated by the library schema and graph instance introduced in Section 4.1.

For example, in that instance $V = \{r, a, c, l, b_1, b_2\}$ and E contains the six displayed edges. For the `OWNS` edge e_{owns}, $\lambda_V(r) = \{\texttt{Reader}\}$, $\rho_V(r)(\texttt{firstname}) = \texttt{'Alice'}$, $\lambda_E(e_{\mathsf{owns}}) = \{\texttt{OWNS}\}$, $\mathrm{src}(e_{\mathsf{owns}}) = r$, $\mathrm{dst}(e_{\mathsf{owns}}) = a$, and $\mathrm{dir}(e_{\mathsf{owns}}) = \rightarrow$; the edge-property map is illustrated by $\rho_E(e_{\mathsf{inc},1})(\texttt{late_fee}) = 0.0$ for the first `INCLUDES` edge.

Because property maps are partial, let $\mathsf{absent} \notin \mathbb{V}$ denote absence of a property, rather than a GQL value, and use the totalized lookup notation

$$\widehat{\rho}_X(x,k) = \begin{cases} \rho_X(x)(k), & k \in \mathrm{dom}(\rho_X(x)), \\ \mathsf{absent}, & k \notin \mathrm{dom}(\rho_X(x)), \end{cases} \qquad X \in \{V, E\}.$$

Evaluation of a property reference is a separate operation: in particular, when its declared type is nullable, a reference to an absent property evaluates to `null`, as specified by GQL.

Note

GQL's syntactic restrictions guarantee a finite graph-pattern result: every unbounded quantifier is guarded by a restrictive path mode, a selective path-search prefix, or the `DIFFERENT EDGES` match mode. In particular, using `TRAIL`, `ACYCLIC`, or `SIMPLE` for all unbounded quantifiers guarantees finiteness (see Section 7.2).

Remark on "openness/closedness". If a graph has an associated *graph type*, it is *closed* (it permits only elements conforming to that type); otherwise it is *open*.

Modeling choices. Although the tuple definition already fixes the types of λ_V, λ_E and ρ_V, ρ_E, we make two conventional but important choices explicit: *(i) Labels as sets.* By typing $\lambda_V : V \to \mathcal{P}(\mathbb{L})$ and $\lambda_E : E \to \mathcal{P}(\mathbb{L})$ we treat labels as *sets* (unordered, without duplicates). Thus, label membership is the primitive notion (e.g., $\ell \in \lambda_V(v)$), and adding/removing a label is idempotent at the semantic level. *(ii) Properties as partial maps.* By typing $\rho_V : V \to (\mathbb{K} \rightharpoonup \mathbb{V})$ and $\rho_E : E \to (\mathbb{K} \rightharpoonup \mathbb{V})$ we model properties as *partial functions* from keys to values. For a particular node or edge, a property key may be absent from its property map, that is, it may lie outside the domain of the corresponding partial function. This is semantically distinct from the property being present with value `null`, when `null` is supported.

Note

These choices deliberately yield a permissive baseline model. Set-valued labels support multi-label classification and align with label tests and index-based access patterns. Partial property maps capture sparse, heterogeneous attribution: different elements may carry different key sets. When tighter control is required, a closed graph's constraining graph type requires every element to have exactly the label and property-type sets of one of its declared element types and to satisfy its key/unique constraints.

6.3 Values and Types

Base universes. We use the single universes $\mathbb{A}, \mathbb{L}, \mathbb{K}, \mathbb{T}$, and $\mathbb{V}$ fixed in Section 6.1; no second value or type universe is introduced here. Let $\sqsubseteq \subseteq \mathbb{T} \times \mathbb{T}$ be a reflexive, transitive subtyping relation, where $\tau_1 \sqsubseteq \tau_2$ is read as "τ_1 is a subtype of τ_2", meaning that every value of type τ_1 is also admissible as a value of type τ_2.

Type constructors.

Scalars: Bool, Int, Real, String, DateTime, $\ldots \in \mathbb{T}$.

Composite: LIST$\langle\tau\rangle$, RECORD$\langle n_i{:}\tau_i\rangle$, PATH $\in \mathbb{T}$.

We write Opt(τ) only as meta-level shorthand for a value type that contains the null value; it is not a GQL type constructor.

Graph element references: NodeRef, EdgeRef $\in \mathbb{T}$ (reference value types whose referents are nodes and edges, respectively).

Conceptually, nullability is represented via the option notation: a site (that is, a typed read/write location such as a property slot or a variable binding; see Chapter 17 for a precise definition) of type Opt(τ) admits the null value null.

Typing contexts. A *type environment* Ξ is a finite map from names to types (for properties, parameters, and binding variables). We write the standard judgment $\Xi \vdash e : \tau$ for expressions. Subsumption: if $\Xi \vdash e : \tau$ and $\tau \sqsubseteq \tau'$, then $\Xi \vdash e : \tau'$.

Expression-evaluation interface. Let Err be an abstract carrier of error or exception-condition tokens. For an expression E and a record r, we reserve $[\![E]\!](r) \in \mathbb{V} \cup \mathsf{Err}$ for evaluation in the row environment induced by r, with the store and global environment fixed by the surrounding execution context. For a Boolean expression φ, $[\![\varphi]\!]^{\mathsf{tv}}(r)$ analogously denotes a member of $\mathbb{B}_3$ or an error. The complete evaluation judgment is given in Chapter 8.

6.4 Basics of Paths as Data

A *path in a graph* G is a finite sequence of graph elements of odd length that starts and ends at a node, i.e., a node–edge–…–edge–node sequence $\langle v_0, e_1, v_1, \ldots, e_n, v_n\rangle$ of length $2n + 1$; the orientation of each edge in the path is determined relative to the order of nodes in the sequence ($\rightarrow$ to the left/right, or undirected). We distinguish the following classes: *trail* (no repeated edges), *simple* (no repeated nodes, except possibly first = last), and *acyclic* (no repeated nodes). *Formally,* $w = \langle v_0, e_1, v_1, \ldots, e_n, v_n\rangle$ is simple iff $v_i \neq v_j$ for all $0 \leq i < j \leq n$ *except possibly* the pair $(i, j) = (0, n)$, allowing $v_0 = v_n$. We write Path(G) for the set of all such paths in G. The full treatment of path values, concatenation, path modes, and result projection is given in Chapter 10.

Material path values and length. We write PathVal for the carrier of material path values. In accordance with the GQL value model, a path value encapsulates a path element list of graph-element reference values; whenever none of those references is invalidated,

their referents identify a path, independently of the current working graph. Concretely, a material path value is written as

$$p = \mathrm{PATH}\Big[\langle n_0, e_1, n_1, \ldots, e_k, n_k\rangle\Big],$$

where $k \geq 0$, each n_i is a node reference value, each e_i is an edge reference value, and the list alternates starting and ending with a node reference (hence has odd length $2k+1$). The *path length* is the number of edge references:

$$\mathrm{PATH_LENGTH}(p) = k.$$

In particular, the length-zero path $\mathrm{PATH}[\langle n_0\rangle]$ is well-formed and satisfies

$$\mathrm{PATH_LENGTH}\Big(\mathrm{PATH}[\langle n_0\rangle]\Big) = 0.$$

Null-propagation. If $p = \mathsf{null}$ (i.e., p is the null value admitted by a nullable path value type), then $\mathrm{PATH_LENGTH}(p) = \mathsf{null}$. Malformed representations (that do not alternate or do not start/end in a node reference) are outside this well-formed path domain; the standard path-value constructions do not produce them. A full formalization of path values, path modes, finiteness and selection, and path accessors is given in Chapter 10.

6.5 Lists and Structural Access

List, record, and path types are constructed value types. They provide the abstract carriers for nested and structured values that occur as property values, as components of binding tables, and as results of expressions (including path-returning queries). By fixing their behavior here, we obtain a uniform basis for the typing and evaluation rules for structural accessors and functions in later chapters.

- *Lists.* A list value is a finite sequence $[v_1, \ldots, v_n]$. Conceptually we describe indexing in ordinal (1-based) terms: $[v_1, \ldots, v_n][i] = v_i$ if $1 \leq i \leq n$; the precise mapping from index expressions to positions (ordinal vs. offset indexing, from-end handling, and out-of-range behavior) follows the session's list-behavior policy. The declared type of list concatenation is determined by GQL's general combination of value types; evaluation then follows the standard null and maximum-cardinality rules.
- *Records.* A record value is a finite mapping from field names to values, written $r.n$ for the value in field n. For records with a closed record type τ_{rec}, a record value r supports field access $r.n$ whenever $n \in \mathrm{dom}(\tau_{\mathrm{rec}})$. If the source record has no field named n, the reference returns `null` when its declared type is nullable; otherwise it raises data exception 22G03. Records model structured values such as rows of binding tables or compound results of expressions (e.g., a `RETURN` item with multiple

named components), so that field access in the language can be defined once in terms of this generic record notion.

- *Paths.* For the path values defined above, we use `nodes`(p), `edges`(p), and `length`(p) as meta-level shorthands derived from the standard accessors ELEMENTS(p) and PATH_LENGTH(p); both standard accessors return null when their path input is null. By treating paths as first-class values with these primitive operations, we can later give uniform typing and evaluation rules for path-returning patterns and path functions, independently of how paths are physically represented in an implementation.

Chapter 6 — Takeaways

- **Purpose:** Established the abstract property-graph data model that all later semantics referenced.
- **Introduced:** A precise notion of a graph state (nodes, edges, endpoints, labels, and properties); universes for identifiers, labels, keys, and values; a value/type discipline including composite values; paths as first-class values.
- **Key distinctions:** Identity (nodes/edges) versus annotation (labels/properties); absent or unknown information versus ordinary values (as later used by expression semantics).
- **Consequences:** Enabled later chapters to define semantics over a fixed carrier with no implicit assumptions; allowed path querying to return structured path values rather than only endpoints; provided hooks needed for typing, constraints, and update effects.
- **Bridge:** Established the foundation required to formalize graph patterns and matching.

7 Patterns and Matching

This chapter formalizes the graph-pattern layer of GQL, which provides the primary mechanisms for selecting, relating, and constraining graph elements. We present the syntax of path and graph patterns, the scoping and safety conditions for variables, and the matching semantics that turn patterns into binding tables over a given graph.

7.1 Graph Patterns

A *graph pattern* is the central declarative object used by GQL to describe which parts of a property graph are to be selected, related, and exposed as bindings. Intuitively, a graph pattern specifies a finite syntactic description of a possibly large set of graph configurations: nodes, edges, paths between nodes, labels, properties, directions, and additional predicates that must be satisfied by the matched graph elements.

In this chapter, we treat a graph pattern as a structured expression whose primary components are *path patterns*. A path pattern describes an admissible path-shaped configuration in the graph, while a graph pattern combines one or more such path patterns and constrains their joint matches. Thus, graph patterns form the bridge between the graph-theoretic layer of the data model and the binding-table layer used by the query algebra. Concrete rigid and conjunctive graph patterns are used in Section 4.4.

Path and multi-path bindings. A *path binding* is a sequence whose entries associate variables or internal pattern markers from one path pattern with graph elements. A *multi-path binding* for a graph pattern is a tuple containing one path binding produced by each of its path-pattern components.

Abstract form. Let PPat denote the set of well-formed path patterns. We write GPat for the set of well-formed graph patterns. Abstractly, a graph pattern $\Gamma \in \mathsf{GPat}$ has the form

$$\Gamma = \langle mm,\, P_1, \ldots, P_n,\, kc,\, \varphi \rangle, \qquad n \geq 1,$$

where:

- mm is an optional *match mode*, controlling whether repeated element bindings or repeated edge bindings are admitted;
- $P_1, \ldots, P_n \in \mathsf{PPat}$ are the path-pattern components of the graph pattern;
- kc is an optional *keep clause*, used to determine which path bindings are retained when path selection is applied at the graph-pattern level;

- φ is an optional graph-pattern search condition, corresponding to a `WHERE` clause.

The following simplified BNF captures the syntactic spine used in this chapter:

```
graph-pattern      ::= [ match-mode ] path-pattern-list
                       [ keep-clause ]
                       [ graph-pattern-where-clause ]

match-mode         ::= "REPEATABLE" element-bindings-or-elements
                     | "DIFFERENT" edge-bindings-or-edges

path-pattern-list ::= path-pattern { "," path-pattern }

keep-clause        ::= "KEEP" path-prefix

graph-pattern-where-clause
                   ::= "WHERE" search-condition
```

The full grammar contains additional syntactic refinements, but this abstraction is sufficient for the semantic development below: a graph pattern is a list of path patterns equipped with optional global controls over repetition, path selection, and filtering.

If a `KEEP` clause is present, its path-pattern prefix is distributed to every path pattern by the syntactic transformation specified for a graph pattern; those path patterns may not already contain a path-search prefix. The `KEEP` clause is then removed. If no match mode is written, the match mode is implementation-defined. `REPEATABLE ELEMENTS` imposes no cross-pattern repetition filter, whereas `DIFFERENT EDGES` retains only multi-path bindings in which no graph edge is bound at two different positions.

Path-pattern components. Each path pattern P_i is evaluated against the current property graph and produces candidate path bindings. These bindings assign graph elements to the variables declared inside the corresponding path pattern. When a graph pattern contains several path patterns, the individual candidate bindings are combined by compatibility on shared variables. Shared variables therefore play the role of joins between path-pattern components.

For example, in a pattern consisting of two path patterns

$$(a) - [e_1] -> (b), \qquad (b) - [e_2] -> (c),$$

the variable b is shared. A graph-pattern match must therefore choose bindings for both path patterns such that the end node of the first component and the start node of the second component are the same graph element. In this sense, graph patterns generalize ordinary edge-pattern matching to joined path-shaped constraints.

Graph-pattern variables. Variables declared in graph patterns are called *graph-pattern variables*. They include element variables, which range over nodes or edges, and path variables, which range over path values. For a graph pattern Γ, we write

$$\mathrm{vars}(\Gamma)$$

for the finite set of graph-pattern variables declared by Γ. We distinguish:

- *node variables*, bound to nodes;
- *edge variables*, bound to edges;
- *path variables*, bound to path values computed during matching;
- *subpath variables*, declared at the head of a parenthesized path pattern but not bound to a value and not referenceable; they distinguish reduced matches for multiset alternation.

Let $\mathrm{vars}_{\mathrm{bind}}(\Gamma)$ be the node, edge, and path variables of Γ, and let

$$\mathrm{vars}^{\mathrm{bind}}_{\mathrm{exp}}(\Gamma) := \mathrm{vars}_{\mathrm{exp}}(\Gamma) \cap \mathrm{vars}_{\mathrm{bind}}(\Gamma),$$

where $\mathrm{vars}_{\mathrm{exp}}(\Gamma)$ is the set of variables exposed by the pattern. Only binding variables (node, edge, and path variables) can contribute columns to the resulting binding table. Anonymous node and edge positions participate in matching without declaring graph-pattern variables; they and temporary symbols are removed during reduction.

Local and global constraints. A graph pattern may contain constraints at two different levels. First, local constraints occur inside node patterns, edge patterns, and parenthesized path patterns. These include label tests, property tests, direction constraints, and locally scoped predicates. Second, a graph pattern may contain a global `WHERE` clause whose search condition is evaluated after the path-pattern components have produced compatible bindings.

We therefore distinguish:

$$\varphi_{\mathrm{loc}} \quad \text{and} \quad \varphi_{\mathrm{glob}},$$

where φ_{loc} denotes constraints evaluated during local path-pattern matching, while φ_{glob} denotes the graph-pattern search condition applied to the combined bindings of the graph pattern. This distinction is important for selective path search: local predicates may reduce the candidate paths before selective search is applied, whereas the global `WHERE` clause filters already combined matches.

Reduced matches. The immediate semantic result of a graph pattern is a set of *reduced matches*. A raw match may contain temporary symbols, anonymous element positions, and auxiliary information introduced by the machinery of path-pattern evaluation. Reduction removes these internal artefacts and keeps only the information that is semantically relevant for subsequent query processing.

We write

$$[\![\Gamma]\!]_G$$

for the reduced matches of a graph pattern Γ over a property graph G. The value-bearing component μ of a reduced match is a finite assignment, possibly structured, from the exposed binding variables of Γ to graph elements, path values, or groups of such values:

$$\mu : \mathrm{vars}^{\mathrm{bind}}_{\mathrm{exp}}(\Gamma) \rightharpoonup \mathbb{V}.$$

Here $\mathbb{V}$ includes reference values for graph elements and constructed path values. A reduced match may additionally retain non-referenceable subpath-marker information used to distinguish the branches of multiset alternation; that information is not a value assignment in μ. We write $\mathsf{REDUCE}(m)$ for the reduction of a raw match m: it removes temporary symbols and anonymous structure and retains the exposed binding information described above. Thus the notation is available before it is used in the staged matching account below.

From graph patterns to binding tables. Graph-pattern matching itself is not the final query result. The reduced matches are subsequently lifted to a binding table whose columns correspond to the exposed binding variables of the pattern, possibly restricted by an explicit `YIELD` clause. At the level of this chapter, we separate these two steps:

$$\Gamma \quad \rightsquigarrow \quad [\![\Gamma]\!]_G \quad \rightsquigarrow \quad \mathrm{BT}_\Gamma.$$

The first step is graph-pattern matching; the second step is binding-table formation. This separation is useful because the same graph-pattern machinery is used by different language constructs: ordinary `MATCH` statements, `OPTIONAL MATCH` statements, nested match blocks, and predicates such as `EXISTS`.

Role in the query pipeline. Within a linear query statement, a graph pattern is evaluated relative to the current working graph and the current working table. Existing bindings may be used to correlate the pattern with earlier query clauses, and newly introduced bindings extend the working table. Thus, graph patterns provide the graph-facing part of the language, while binding-table operations provide the algebraic part: selection, projection, joins, grouping, aggregation, and ordering.

The remainder of this chapter develops this pipeline in stages. We first specify the syntax of path patterns and their path-specific controls. We then define variable exposure and safety conditions, describe the matching pipeline, and finally explain how matches are represented as binding tables and consumed by `MATCH` statements.

7.2 Pattern Syntax

A path pattern is a declarative description of a family of admissible paths between graph elements (nodes and edges) in a graph pattern, together with bindings for their endpoints and, optionally, for the path value itself. It consists of: (1) an optional path variable declaration, (2) an optional prefix specifying the path mode and, optionally, a selective search prefix, and (3) a pattern expression built recursively from element patterns (nodes and edges), with quantification/alternation/concatenation.

Path modes. A path mode determines which repetitions of graph elements are permitted in a path binding. The available modes are: WALK, the default mode, which imposes no restrictions on repeated nodes or edges; TRAIL, which excludes path bindings containing a repeated edge; ACYCLIC, which excludes path bindings containing a repeated node; and SIMPLE, which excludes repeated nodes except when the repeated node is the first and last node of the path, thus permitting a closed path whose interior nodes are distinct.

Using one of the *restrictive* modes TRAIL, ACYCLIC, or SIMPLE for every unbounded repetition guarantees that the result set of graph-pattern matching is finite. The same finiteness guard can instead be supplied by a selective path-search prefix or by the `DIFFERENT EDGES` match mode, as required by the static rules for every unbounded quantifier.

A selective path search prefix (SPSP) further restricts the candidate path bindings produced by a path pattern. It partitions the candidate bindings according to their endpoints, that is, the first and last nodes of the paths, and selects a finite subset separately from each partition. Depending on the SPSP form, the selection may comprise a specified number of arbitrary paths, all shortest paths, a specified number of shortest paths, or all paths belonging to a specified number of the shortest path-length groups. Consequently, the use of an SPSP guarantees a finite result set regardless of the path mode.

The path modes do not form a total order under implication. In particular, SIMPLE and TRAIL are incomparable: in a graph containing undirected edges, a simple path may traverse the same edge twice in opposite directions and therefore need not be a trail.

Availability. The explicit WALK keyword, the restrictive path modes, and the individual SPSP forms are controlled by separate optional features. In ISO/IEC 39075, an optional feature is identified by a Feature ID and defined by its Conformance Requirements

(Subclause 5.3.7); Annex D, *GQL optional feature taxonomy*, collects the optional features informatively.

Stored vs. computed paths. A *computed path* is a path value produced *on the fly* by evaluating the regular path expression of a path pattern over the current graph during matching. Some graph-query formalisms (and some implementations) also admit *stored paths*, i.e., persistent path objects with identity that can be referenced later (for example, as elements of a constructed graph). In this chapter, path patterns are treated as producing *computed* path values by default; when stored paths exist, they matter only if a specific language feature ranges over those stored path objects rather than enumerating paths from the underlying graph connectivity.

Path restrictors as admissibility filters. For a regular path expression R and path mode pm, let $\mathsf{Cand}(R, \mathsf{pm})$ denote abstractly the candidate path bindings that satisfy the local element predicates and the chosen mode. For a candidate family, let $\Pi_{u,v}$ denote restriction to the endpoint pair (u, v), and let SPSP_s denote abstractly the partition-wise selector determined by selective prefix s. Their witness-level definitions and the concrete selector variants are given in Section 11.2. Although the concrete syntax calls them *path modes*, it is helpful to view WALK, TRAIL, ACYCLIC, and SIMPLE as *restrictors*: they act as predicates on the space of candidate paths that a regular path expression would otherwise admit. Concretely, for a fixed regular path expression R and endpoints (u, v), the candidate space for matching is first generated and then filtered by the chosen mode. Only the surviving paths are passed to the selective path-search procedure SPSP_s, where s specifies the applicable path-selection policy; the selected paths are then available for joins and REDUCE.

Selective path search prefixes (SPSP). The (potentially infinite) space of path bindings is partitioned by the pair of endpoints (u, v); we write $\Pi_{u,v}(\cdot)$ for the (u, v)-partition selector. A search-prefix selector (e.g., shortest, all shortest, any, counted shortest) chooses subsets inside each (u, v)-partition independently; we denote the resulting selection operator abstractly by $\mathsf{SPSP}_s(\cdot)$, where s is the chosen SPSP variant. Even under WALK, any *selective* path search prefix (that is, any path-search prefix other than `ALL PATHS`, including the `ANY` and `SHORTEST`-family prefixes) returns a finite set of path bindings; hence the set of reduced matches is finite.

Concrete syntax for path patterns. At the level of concrete syntax, a ⟨*path pattern*⟩ associates an optional path variable and prefix with a regular path expression. Listing 7.1 isolates the spine of path-pattern declarations; the full grammar of `path-expr` is given in the chapter on regular path expressions (see Chapter 11).

Listing 7.1: Simplified grammar of path-pattern declarations

```
path-pattern    ::= [ path-var "=" ] [ path-prefix ] path-expr

path-prefix     ::= path-mode-prefix | search-prefix

path-mode-prefix ::= path-mode [ "PATH" | "PATHS" ]

path-mode       ::= "WALK"
                  | "TRAIL"
                  | "ACYCLIC"
                  | "SIMPLE"

search-prefix   ::= "ALL" [ path-mode ] [ "PATH" | "PATHS" ]
                  | "ANY" [ non-negative-int ] [ path-mode ] [ "PATH" | "
    PATHS" ]
                  | "ALL SHORTEST" [ path-mode ] [ "PATH" | "PATHS" ]
                  | "ANY SHORTEST" [ path-mode ] [ "PATH" | "PATHS" ]
                  | "SHORTEST" non-negative-int [ path-mode ] [ "PATH" | "
    PATHS" ]
                  | ...  /* other GQL search modes */

path-var        ::= identifier
path-expr       ::= ...  /* defined in the regular-path-expression chapter */
```

Intuitively, `path-var` names the binding position for the path value, `path-mode` selects the admissible class of paths (walks, trails, acyclic paths, or simple paths), and `search-prefix` chooses, for each pair of endpoints, which paths (e.g., a single shortest path, all shortest paths, or all paths) are materialized as bindings. If no path mode is specified, `WALK` applies. If no path-search prefix is specified, the path pattern is non-selective; there is no implicit `ANY` selection. Omitting `path-var` yields an anonymous path whose value is not directly exposed as a binding, although its endpoints and other incident variables may still participate in joins and projections.

7.3 Variable Exposure and Safety

We distinguish (for any ⟨path pattern⟩) *boundary variables* (visible at the point of composition and eligible for joins), *interior variables* (scoped to the component), and *strict interior variables* (interior variables that are not boundary).

Named degrees. For any graph-pattern variable x within a component C, exposure has a degree

$$\mathrm{D}_{\mathrm{expo}}(x, C) \in \{\textit{unconditional},\ \textit{conditional},\ \textit{not-exposed}\}.$$

For a referenceable binding variable x, we additionally use

$$\mathrm{D}_{\mathrm{ref}}(x, C) \in \{\textit{singleton},\ \textit{effectively-bounded-group},\ \textit{effectively-unbounded-group}\}.$$

A subpath variable has an exposure degree only for its non-value marker; it is not bound to a value, cannot be referenced, and therefore has no degree of reference.

- **Unconditional vs. conditional exposure.** x is *unconditionally exposed* in C iff every reduced match of C contains x, as a value-bearing binding or, for a subpath variable, as a marker. Otherwise, if some reduced matches of C do not contain x, x is *conditionally exposed*. If x is interior (non-boundary), it is *not-exposed* outside C.
- **Singleton.** x is a *singleton* in C iff, per reduced match of C, x contributes *at most one* binding value.
- **Effectively bounded group.** x is an *effectively-bounded group* in C iff there exists a finite bound B_C determined *syntactically* by C (e.g., finite upper bounds in repetition, finite alternation/branching that does not depend on data cardinalities) such that each reduced match of C contributes at most B_C bindings for x.
- **Effectively unbounded group.** Otherwise x is an *effectively-unbounded group* (e.g., due to unbounded repetition like Kleene $^+$ or * without further syntactic bounds).

For referenceable binding variables, we order reference degrees by

$$\textit{singleton} \ <\ \textit{effectively-bounded-group} \ <\ \textit{effectively-unbounded-group}.$$

Remark (groups and optionality). Referenceable variables with $\mathrm{D}_{\mathrm{ref}}(x, C) \neq \textit{singleton}$ are referred to as *group variables*. A group variable may be bound to an *empty* group in a reduced match (e.g., due to zero repetitions), and such an empty group still counts as a binding for purposes of exposure.

Path-variable degree (local). If a path pattern declares a path variable p (i.e., it has the form `p = ...`), then $\mathrm{D}_{\mathrm{ref}}(p, C) = \textit{singleton}$ and $\mathrm{D}_{\mathrm{expo}}(p, C) = \textit{unconditional for the component } C \textit{ itself}$: every reduced match produced by that component binds p to exactly one path value. *Remark.* Statement-level optionalization (e.g., `OPTIONAL MATCH`) may later turn any newly introduced binding (including a path variable) into a *conditional* output column by filling it with `null` when the optional match fails; this is treated in the statement semantics, not as a path-pattern constructor.

In the GQL grammar, a path primary is an element pattern, a parenthesized path pattern expression, or a simplified path pattern expression.

Propagation rules. Let P, Q be path-pattern primaries; the propagation of exposure and reference degrees over the corresponding pattern constructors is summarized in Table 7.1.

For a subpath variable, only the exposure clauses apply; clauses about reference degree or bound values apply only to binding variables.

Construct	Propagation
Primary (node/edge)	Endpoints are *unconditionally exposed singletons*; no interior variables.
Concatenation $P \cdot Q$	Boundary vars are the endpoints of $P \cdot Q$; interior accumulates non-boundary of P, Q. If a variable is exposed by both operands, both exposures must be unconditional singleton and the result is unconditional singleton; otherwise it is exposed by exactly one operand and retains that operand's degree.
Alternation $P \mid Q$	Exposure: unconditional only if the same variable is unconditionally exposed in both arms; otherwise conditional or not-exposed. Reference degree: $\mathrm{D}_{\mathrm{ref}}(x, P \mid Q) = \max\{\mathrm{D}_{\mathrm{ref}}(x, P), \mathrm{D}_{\mathrm{ref}}(x, Q)\}$ (least upper bound).
Questioned path-pattern primary $P?$	For each variable x: if $\mathrm{D}_{\mathrm{ref}}(x, P) \neq$ *singleton* (i.e., x is a group variable in P), then $\mathrm{D}_{\mathrm{expo}}(x, P?) = \mathrm{D}_{\mathrm{expo}}(x, P)$ and $\mathrm{D}_{\mathrm{ref}}(x, P?) = \mathrm{D}_{\mathrm{ref}}(x, P)$; otherwise (singleton variables), $\mathrm{D}_{\mathrm{ref}}(x, P?) = \mathrm{D}_{\mathrm{ref}}(x, P)$ and $\mathrm{D}_{\mathrm{expo}}(x, P?)$ is *conditional* whenever x is exposed by P (i.e., unconditional exposure in P becomes conditional in $P?$).
Quantification $P^{\{m,n\}}$, P^*, P^+	Endpoints remain boundary; here m and n are, respectively, the lower and upper bounds on the repetition count in the cardinality constraint (with n possibly $= \infty$; for P^* and P^+ this upper bound is understood as ∞). Every element or subpath variable declared by P has group exposure. For a binding variable the group is effectively bounded for a fixed or upper-bounded quantifier, or within a restrictive path mode; otherwise it is effectively unbounded at this syntactic level.

Table 7.1.: Propagation of exposure and reference degrees for variables in path patterns.

Constraints. The following static side-conditions constrain how variables with the degrees from Table 7.1 may be referenced; they are taken as part of the well-formedness conditions for path patterns and aggregates in the remainder of this chapter.

1. **Non-local reference** is permitted *only* for references with *singleton* degree, i.e., when $\mathrm{D}_{\mathrm{ref}}(x, C) =$ *singleton*.
2. In any aggregate, all **group references** must range over the *same* GPV (graph-pattern variable).
3. For **selective** components, joins are permitted *only* on boundary variables.
4. A **selective** ⟨path pattern⟩ (selective path pattern, SPP) *shall not reference* any graph-pattern variable not declared within this SPP.

7.4 Matching Semantics

Sketch of matching semantics. This section does not introduce a new formal semantics; instead, it gives an informal, staged account of how the graph-pattern matching

machinery (path evaluation, selective search, joins, and binding reduction) behaves in our notation. The numbered steps below should be read as an operational pipeline that explains how a graph pattern is turned into a binding table, complementing the more formal binding-table semantics in Sections 7.5 and 7.6.

For a given graph pattern Γ in the context of graph G:

1. **Syntactic normalization.** Distribute a graph-level `KEEP` prefix to every path pattern and remove the `KEEP` clause. Determine the explicit or implementation-defined match mode.
2. **Local generation (with scoped filters).** For each ⟨path pattern⟩ component with regular path expression R and path mode pm, compute the candidate set $\mathsf{Cand}(R, \mathsf{pm})$ of its path bindings that satisfy the local predicates of the element pattern and the chosen path mode. If the component is a parenthesized path pattern with an attached `WHERE` clause, that clause is evaluated here, before any SPSP selection. A non-local reference is permitted only at singleton reference degree; a selective path pattern cannot reference a graph-pattern variable declared outside that selective component. No selective search is applied at this stage.
3. **Selective stage (per component).** If a component is *selective* with selective prefix s, apply its selector independently inside each endpoint partition: for each endpoint pair (u, v), replace the candidate partition $\Pi_{u,v}(\mathsf{Cand}(R, \mathsf{pm}))$ by the selected subset $\mathsf{SPSP}_s(\Pi_{u,v}(\mathsf{Cand}(R, \mathsf{pm})))$. If `DIFFERENT EDGES` applies to the single selective component, discard repeated-edge bindings before selection. The result is a finite set of bindings for that component. Non-selective components keep their full candidate sets $\mathsf{Cand}(R, \mathsf{pm})$ unchanged.
4. **Product and joins.** Form the set-theoretic Cartesian product and enforce natural equijoins on the shared boundary variables that are unconditionally exposed singletons. For selective ⟨path pattern⟩ components, joins are restricted: strict interior variables of a given SPP MUST NOT join with exterior variables or with interior variables of another SPP; only joins on boundary variables are permitted.
5. **Match-mode filtering.** Under `DIFFERENT EDGES`, retain only multi-path bindings in which no graph edge occurs in two different edge-binding positions. Under `REPEATABLE ELEMENTS`, no such filter is applied.
6. **Global filtering.** Apply `WHERE` search conditions (non-local predicates and selections), thereby reducing the set of bindings.
7. **Reduction.** Apply the REDUCE function: remove temporary symbols, reduce anonymous nodes/edges, merge adjacent anonymous nodes, retain subpath-marker distinctions introduced for multiset alternation, and deduplicate results under set semantics at the level of reduced matches.

8. **Binding-table formation.** The final result of matching is the binding table BT_Γ over the columns corresponding to the exposed binding variables and projections from the pattern by appending one record for every reduced match for each occurrence of an incoming working-table record. The reduced-match result itself is a set; duplicate binding-table records may nevertheless arise from duplicate incoming records or from distinct reduced matches whose yielded records are equal. See *Availability* above for the applicable optional-feature mechanism.

Informative homomorphism analogy. In general graph-query formalisms, a common distinction is whether a pattern is matched by *homomorphisms* (structure-preserving mappings that may identify distinct pattern variables) or by *injective matches* (structure-preserving mappings in which distinct pattern variables must be mapped to distinct graph elements, corresponding to subgraph-isomorphism-style matching). Both notions preserve the intended incidence constraints of the pattern; the difference is whether distinct variables are allowed to map to the same graph object. This is a graph-theoretic analogy, not an additional GQL match mode: the standard match modes are `REPEATABLE ELEMENTS` and `DIFFERENT EDGES`, with the semantics given above.

Formally, let P be a pattern with node variables V_P and edge variables E_P, and let G be a graph with node set V_G and edge set E_G. A *match* is a pair of total functions $h_V : V_P \to V_G$ and $h_E : E_P \to E_G$ that preserves the required incidence and typing constraints of P in G (labels/types, directions, endpoint compatibility, and any explicitly stated property tests). We write $h = (h_V, h_E)$.

Homomorphic matching admits all such structure-preserving mappings h. *Injective matching* further requires injectivity on the chosen variable sets:

$$\begin{aligned} \mathsf{Inj}_V(h) &:\iff \forall x \neq y \in V_P.\ h_V(x) \neq h_V(y), \\ \mathsf{Inj}_E(h) &:\iff \forall e \neq f \in E_P.\ h_E(e) \neq h_E(f). \end{aligned}$$

A fully isomorphic regime typically enforces $\mathsf{Inj}_V(h) \wedge \mathsf{Inj}_E(h)$, and, if the model treats nodes and edges as disjoint carriers, optionally also a cross-sort disjointness condition (implementation-dependent).

Example (a concrete match-function pair). For the pattern

$$P_{\mathsf{inc}} = (x_l{:}\texttt{Loan}) - [x_i{:}\texttt{INCLUDES}] \to (x_b{:}\texttt{Book}),$$

one match in the library graph is

$$h = (h_V, h_E), \qquad h_V(x_l) = l, \quad h_V(x_b) = b_1, \quad h_E(x_i) = e_{\mathsf{inc},1}.$$

Reduced matches and binding-table multiplicities. Graph-pattern evaluation produces a set of reduced matches. If two raw matches become equal after REDUCE, they contribute only one reduced match. For each occurrence of an incoming working-table record, graph-pattern binding table formation appends one record per reduced match. Equal output records can nevertheless have multiplicity greater than one when distinct reduced matches have equal yielded fields or when the incoming table contains duplicate records.

Let $\mathsf{Hom}(P, G)$ be the set of all distinct complete homomorphic matches of P in G, including assignments to internal variables and any path witnesses that distinguish matches. Let $\mathsf{bind}(h)$ denote the binding extracted from h, restricted to the variables exported by P, and let $r(h)$ be the reduced match induced by h. For this analogy, a set-valued formulation is:

$$[\![P]\!]_G \;=\; \{\, r(h) \mid h \in \mathsf{Hom}(P, G) \,\}.$$

Changing the admissible raw mappings changes this set, but equal reduced matches are still deduplicated before binding-table formation.

Excluding classes of matches via explicit guards. Rather than baking a single global choice (homomorphism vs. isomorphism) into the core semantics, it is often preferable to keep homomorphic matching as the baseline and let users rule out unwanted matches by adding *guards* that constrain identifications. This captures a spectrum of behaviors between pure homomorphism and full isomorphism.

Let Φ be a predicate over bindings (e.g., conjunctions of inequalities between variables). Define guarded matching by filtering the match set:

$$\mathsf{Hom}_\Phi(P, G) = \{\, h \in \mathsf{Hom}(P, G) \mid \Phi(\mathsf{bind}(h)) \,\},$$
$$[\![P]\!]^\Phi_G = \{\, r(h) \mid h \in \mathsf{Hom}_\Phi(P, G) \,\}.$$

A common family of guards are *injectivity guards* on selected variable subsets. For any $S \subseteq V_P$ define

$$\mathsf{Inj}_S(h) \;:\Longleftrightarrow\; \forall x \neq y \in S.\; h_V(x) \neq h_V(y),$$

and analogously for edge variables. Then "node-isomorphic" matching is obtained by choosing $\Phi := \mathsf{Inj}_{V_P}$, while full isomorphism corresponds to $\Phi := \mathsf{Inj}_{V_P} \wedge \mathsf{Inj}_{E_P}$ (and any additional disjointness conditions required by the underlying data model).

7.5 Binding Tables

A *binding table* is the descriptor-bearing object fixed in Section 6.1: it comprises a finite collection of records of one record type, an ordered/unordered indication, and an optional preferred column-name sequence. Thus, for

$$T = \langle \tau_{\mathrm{rec}}, B, \mathit{ord}, \mathit{pcs} \rangle \in \mathsf{BT}, \qquad B \in \mathcal{M}_{\mathrm{fin}}(\mathrm{Rec}_{\tau_{\mathrm{rec}}}),$$

we use $\mathrm{bag}(T) = \langle \tau_{\mathrm{rec}}, B \rangle$ only as its order-insensitive bag projection; it is not an identification of the complete binding-table object with that pair. There is a distinguished unordered unit table containing the unique unit record $\langle\rangle$. Columns are the fields of the record type; operations such as Cartesian product and column-wise operations rely on the compatibility of field/column names and types. Records of a binding table are not modified once the table is built; new tables arise by construction (e.g., products, selections, projections).

Example (a complete binding-table object). Consider the result of the simple rigid-pattern query in Section 4.4. Let d_1 and d_2 denote the DATE values displayed there as `2025-12-01` and `2025-12-10`, respectively. Its closed record type is

$$\begin{aligned}\tau_{\mathrm{lib}} = \{&\texttt{firstname} \mapsto \mathsf{STRING}, \texttt{lastname} \mapsto \mathsf{STRING}, \texttt{title} \mapsto \mathsf{STRING},\\ &\texttt{due_date} \mapsto \mathsf{DATE}, \texttt{late_fee} \mapsto \mathsf{FLOAT}\}.\end{aligned}$$

The two records shown in the result table are

$$\begin{aligned} r_1 = \{&\texttt{firstname} \mapsto \texttt{'Alice'}, \texttt{lastname} \mapsto \texttt{'Brown'},\\ &\texttt{title} \mapsto \texttt{'Graph DB 101'}, \texttt{due_date} \mapsto d_1, \texttt{late_fee} \mapsto 0.0\},\\ r_2 = \{&\texttt{firstname} \mapsto \texttt{'Alice'}, \texttt{lastname} \mapsto \texttt{'Brown'},\\ &\texttt{title} \mapsto \texttt{'Graph Theory'}, \texttt{due_date} \mapsto d_2, \texttt{late_fee} \mapsto 2.5\}. \end{aligned}$$

Thus the bag component is determined by

$$B_{\mathrm{lib}}(r) = \begin{cases} 1, & \text{if } r = r_1 \text{ or } r = r_2,\\ 0, & \text{otherwise,} \end{cases}$$

and the preferred column-name sequence established by the `RETURN` item order is

$$\mathit{pcs}_{\mathrm{lib}} = \langle \texttt{firstname}, \texttt{lastname}, \texttt{title}, \texttt{due_date}, \texttt{late_fee} \rangle.$$

Because the query contains no `ORDER BY` clause, its complete binding-table object is

$$T_{\text{lib}} = \langle \tau_{\text{lib}}, B_{\text{lib}}, \mathsf{unordered}, pcs_{\text{lib}} \rangle.$$

Consequently, the order in which the two records are printed has no semantic significance. The pair $\text{bag}(T_{\text{lib}}) = \langle \tau_{\text{lib}}, B_{\text{lib}} \rangle$ is only the order-insensitive bag projection, not the complete binding-table object.

Bag-semantics convention. The multiplicity of a record in BT models duplicate table records, not duplicate reduced matches (which are deduplicated as a set). For natural joins, equality on shared fields (natural joinability) is required – formal conditions are given by the specification.

7.6 `MATCH` Statements

A `MATCH` statement is the primitive query statement that turns graph patterns into binding tables and thereby expands the current working table of a linear query statement. In the ISO standard, it appears as a *primitive query statement* inside a *linear query statement*, whose incoming and outgoing working record and table types are threaded left–to–right through the sequence of simple query statements. In this section, we give an abstract syntax for `MATCH`, and a denotational semantics that factors through the graph pattern semantics and the binding–table operators of Chapter 9. Examples of a single rigid `MATCH` and successive `MATCH` statements appear in Section 4.4.

Abstract syntax. Let GPat be the set of well-formed graph patterns, including their optional `WHERE` clauses, and let GPBT be the set of graph pattern binding tables, each consisting of a graph pattern optionally followed by a `YIELD` clause. At the statement level, we distinguish *simple* and *optional* match statements.

We write:

$$\begin{aligned}
\mathsf{MatchStmt} \ni S ::= {} & \mathbf{Match}(\Delta_{\text{gpbt}}) \\
& \mid \mathbf{OptMatch}(O) \\
& \qquad \Delta_{\text{gpbt}} \in \mathsf{GPBT},\ O \in \mathsf{MatchOp}, \\
\mathsf{MatchOp} \ni O ::= {} & \mathbf{Match}(\Delta_{\text{gpbt}}) \\
& \mid \mathcal{Q} \\
& \qquad \Delta_{\text{gpbt}} \in \mathsf{GPBT},\ \mathcal{Q} \in \mathsf{MatchBlock}, \\
\mathsf{MatchBlock} \ni \mathcal{Q} ::= {} & \langle S_1; \ldots; S_n \rangle \\
& \qquad n \geq 1,\ S_i \in \mathsf{MatchStmt}.
\end{aligned}$$

Intuitively:

- **Match**(Δ_{gpbt}) corresponds to a simple match statement `MATCH` Δ_{gpbt}.
- **OptMatch**(**Match**(Δ_{gpbt})) corresponds to `OPTIONAL MATCH` Δ_{gpbt}.
- **OptMatch**($\langle S_1; \ldots; S_n \rangle$) corresponds to an *optional match block* `OPTIONAL {` S_1 `...` S_n `}` or `OPTIONAL (` S_1 `...` S_n `)`, where the inner S_i are themselves match statements (simple or optional).

For concreteness, we use the following simplified BNF, which is a faithful abstraction of the ISO Formats for `MATCH`:

```
match-stmt   ::= "MATCH" graph-pattern-binding-table
               | "OPTIONAL" match-operand

match-operand ::= "MATCH" graph-pattern-binding-table
                | "{" match-block "}"
                | "(" match-block ")"

match-block ::= match-stmt { match-stmt }

graph-pattern-binding-table ::=
    graph-pattern [ graph-pattern-yield-clause ]
```

All the syntactic complexity related to linear query statements, focused versus ambient use of graphs, and the detailed declaration of working record types is pushed into other chapters. Here we focus on the semantic contribution of `MATCH` itself.

Semantic setting. Let BT be the domain of descriptor-bearing binding tables fixed in Section 6.1 and recalled in Section 7.5. Its bag projection is finite and has a common header (a set of column names with value types); selection, projection, and joins are developed in Chapter 9.

For each graph pattern binding table $\Delta_{\text{gpbt}} \in \mathsf{GPBT}$ we assume a *graph pattern binding table* operator

$$\mathsf{GPBT}_{\Delta_{\text{gpbt}}} : \mathsf{BT} \longrightarrow \mathsf{BT}$$

that lifts the reduced matches of the graph pattern contained in Δ_{gpbt} (as defined in Chapter 7) to a binding table, applies its explicit or implicit `YIELD` clause, and may be correlated with an incoming binding table of outer variables. This abstracts the ISO notion of a "⟨*graph pattern binding table*⟩" that combines graph pattern matching with an incoming working table.

For a correlated binding-table transformer $\mathcal{F}$, let $\mathsf{OptExpand}_{\mathcal{F}}(T)$ denote the per-record optional expansion of T: every input record occurrence is extended by every row produced by $\mathcal{F}$ for that occurrence, or once with nulls in all statically declared new columns

if $\mathcal{F}$ produces no row. The bag formula for this operator is given in Section 9.2; this declaration fixes the operator before its uses below.

For a binding table $T \in \mathsf{BT}$ we write:

- $\mathrm{cols}(T)$ for the finite set of column names of T;
- $T(X)$ for the restriction of T to a subset $X \subseteq \mathrm{cols}(T)$ of columns (projection);
- $T \bowtie U$ for the natural join (bag-valued) of T and U on common column names, as in Chapter 9;
- $T \triangleleft U$ for the *left outer join* of T and U, reusing the left-outer-join operator from Chapter 9 (in particular, abbreviating the guarded operator $T \triangleleft_\theta U$ where θ is the standard equality guard on shared columns).

In a linear query statement, a match statement S acts on the *current working table* T and produces a new working table T' (that is, the outgoing working table obtained from T by applying the semantics of S, so $T' = [\![S]\!]_{\mathsf{match}}(T)$). At our level of abstraction, we therefore model $\mathsf{MatchStmt}$ by a semantic function

$$[\![-]\!]_{\mathsf{match}} \ : \ \mathsf{MatchStmt} \longrightarrow (\mathsf{BT} \to \mathsf{BT}).$$

Simple `MATCH`. Let $S = \mathbf{Match}(\Delta_{\mathrm{gpbt}})$ be a simple match statement and let $T \in \mathsf{BT}$ be the incoming working table.

The ISO semantics for a simple match statement proceeds by constructing a graph pattern binding table Δ_{gpbt} (using the current working table as input), and then setting the current working table to (a copy of) that result, taking care not to duplicate columns already present in the incoming working types. In the ISO standard, compatibility between incoming bindings and newly produced bindings is enforced inside the construction of the graph pattern binding table.

We capture this behavior at our level of abstraction by taking the output of the graph-pattern binding table operator as the denotation of `MATCH`. Let

$$T_b \ = \ \mathsf{GPBT}_{\Delta_{\mathrm{gpbt}}}(T).$$

We define:

$$[\![\mathbf{Match}(\Delta_{\mathrm{gpbt}})]\!]_{\mathsf{match}}(T) \ = \ T_b.$$

In a query block that begins with a single `MATCH` and whose incoming working table is the singleton table with one empty record, this definition reduces to the binding table semantics for the graph pattern:

$$[\![\mathbf{Match}(\Delta_{\mathrm{gpbt}})]\!]_{\mathsf{match}}(\mathrm{BT}_{\mathrm{unit}}) \ = \ \mathsf{GPBT}_{\Delta_{\mathrm{gpbt}}}(\mathrm{BT}_{\mathrm{unit}}),$$

where $\mathrm{BT}_{\mathrm{unit}}$ is the distinguished unit table (see Section 7.5), with empty header (no columns) and a single empty record $\langle\rangle$.

Optional `MATCH` with a single pattern. Consider an optional match statement of the form $S = \mathbf{OptMatch}(\mathbf{Match}(\Delta_{\mathrm{gpbt}}))$. The ISO semantics does not define an independent denotation for this construct, but instead *desugars* it into an `OPTIONAL CALL` with an explicit list of input binding variables (taken from the incoming working record and table types) and an explicit list of output binding variables (the *new* bindings supplied by the graph pattern binding table). The semantics of `OPTIONAL CALL` then implements a left outer join between the incoming bindings and the results of the call.

At the level of pure binding-table semantics this is the correlated optional expansion from Section 9.2. Let $T \in \mathsf{BT}$ be the incoming working table and let $\mathcal{F}_{\Delta_{\mathrm{gpbt}}}(U) = \mathsf{GPBT}_{\Delta_{\mathrm{gpbt}}}(U)$. We define:

$$[\![\mathbf{OptMatch}(\mathbf{Match}(\Delta_{\mathrm{gpbt}}))]\!]_{\mathsf{match}}(T) \;=\; \mathsf{OptExpand}_{\mathcal{F}_{\Delta_{\mathrm{gpbt}}}}(T).$$

The correlated expansion:

- preserves every row of T;
- when there is at least one match produced by Δ_{gpbt} compatible with a given row of T, duplicates that row for each such match and fills the newly introduced columns according to the match; but
- when there is no such match, preserves the row once and fills the newly introduced columns with the null value (or the appropriate representation of "no binding").

This is exactly the behavior of optional pattern matching familiar from property-graph practice and from Section 9.2.

Example (an unsuccessful optional match).

```
USE libraryGraph
MATCH (r:Reader)
OPTIONAL MATCH (r)-[:USES]->(c:LibraryCard)
  WHERE c.num = 'CARD-9999'
RETURN r.firstname AS firstname, c.num AS card_number
```

For the current library instance, the result contains the record

$$\{\texttt{firstname} \mapsto \texttt{'Alice'},\ \texttt{card_number} \mapsto \mathsf{null}\},$$

because the incoming reader record is retained when the optional pattern has no compatible match.

Optional `MATCH` blocks. A match block $\mathcal{Q} = \langle S_1; \ldots; S_n \rangle$ represents a non-empty sequence of match statements. Its semantics as a block (ignoring optionality) is just functional composition:

$$[\![\mathcal{Q}]\!]_{\mathsf{match}}(T) \;=\; [\![S_n]\!]_{\mathsf{match}}(\ldots [\![S_2]\!]_{\mathsf{match}}([\![S_1]\!]_{\mathsf{match}}(T))\ldots).$$

An optional match statement with a block, $S = \mathbf{OptMatch}(\mathcal{Q})$, is then interpreted as a correlated left outer join between each incoming occurrence and the overall effect of the block evaluated with that occurrence as input. Let $\mathcal{F}_{\mathcal{Q}}(U) = [\![\mathcal{Q}]\!]_{\mathsf{match}}(U)$. Then:

$$[\![\mathbf{OptMatch}(\mathcal{Q})]\!]_{\mathsf{match}}(T) \;=\; \mathsf{OptExpand}_{\mathcal{F}_{\mathcal{Q}}}(T).$$

Informally, the block $\mathcal{Q}$ is evaluated as if it were a sequence of ordinary `MATCH` statements, producing extended rows that may introduce fresh binding variables. The optional wrapper then ensures that every input row is preserved: rows that admit at least one successful extension through $\mathcal{Q}$ are duplicated for each such extension (with the new bindings filled in), while rows that admit no extension through $\mathcal{Q}$ are preserved once, with the new columns filled with nulls. This is equivalent to the standard's translation of `OPTIONAL MATCH` blocks into `OPTIONAL CALL` statements with explicit input and output binding variable lists.

This same notion of a match block is also reused by the ISO semantics of the `EXISTS` predicate, which rewrites `EXISTS { MATCH ...}` or `EXISTS { ...match-block ...}` into a nested query whose body consists of the match block followed by a trivial `RETURN` of a Boolean column.

Interaction with linear query statements. In a full linear query statement,

$$\texttt{USE}\ g\ \texttt{MATCH}\ \Gamma_1\ \texttt{MATCH}\ \Gamma_2\ \ldots\ \texttt{FILTER}\ \varphi\ \texttt{RETURN}\ \Pi,$$

the match statements occur as primitive query statements in the sequence governed by the linear query statement syntax. Let

$$S_1, \ldots, S_k$$

be the `MATCH` or `OPTIONAL MATCH` statements in the order in which they appear. At the level of binding tables, their combined effect on an initial working table T_0 is the composition

$$T_k \;=\; [\![S_k]\!]_{\mathsf{match}}(\ldots [\![S_2]\!]_{\mathsf{match}}([\![S_1]\!]_{\mathsf{match}}(T_0))\ldots),$$

and it is T_k that is subsequently consumed by the selection, projection, aggregation and ordering operators that make up the rest of the query block (Chapter 9). In the "whole-

query denotation" of a query block, `MATCH` thus serves as the bridge between pattern-based access to the graph and the relational-style algebra over binding tables.

7.7 Key Scopes and Finiteness

Let $T \in \mathsf{BT}$ be a binding table with $\mathrm{bag}(T) = \langle \tau_{\mathrm{rec}}, B \rangle$ (see Section 7.5). In this section, displayed pairs denote bag projections; descriptor metadata are propagated by the applicable table operator. The abbreviation $[\![E]\!](r)$ means evaluation of E in the row environment induced by r, as defined formally in Chapter 8. We work throughout with *bag semantics*: the multiset component B records the multiplicity of each record, and each operator below acts on B so that it propagates and combines multiplicities in the standard way (for example, selection preserves the multiplicities of surviving rows; projection does not perform duplicate elimination and its output multiplicities are obtained by summing multiplicities of preimages; unions add multiplicities; joins multiply them). Unless explicitly stated otherwise (in particular for `DISTINCT` and `UNION` without `ALL`), the operators defined in this section are multiset-preserving in this sense.

Extend. The extension with projection for expressions $E_1, \ldots, E_k$ and fresh names $n_1, \ldots, n_k$ is typed by $\Xi_{\tau_{\mathrm{rec}}} \vdash E_i : \tau_i$ for $1 \le i \le k$, where $\Xi_{\tau_{\mathrm{rec}}}$ is the type environment induced by τ_{rec}. The extension operator is defined by

$$\mathrm{EXT}_{E_1 \mapsto n_1, \ldots, E_k \mapsto n_k}(\langle \tau_{\mathrm{rec}}, B \rangle) \;=\; \Big\langle\, \tau_{\mathrm{rec}} \cup \{n_i \mapsto \tau_i\}_{i=1}^{k},\; B' \Big\rangle,$$

where the output multiplicity function B' is defined by

$$B'(r') \;:=\; \sum_{r \in \mathrm{Rec}_{\tau_{\mathrm{rec}}}} \mathbf{1}\Big[r' = r \cup \{n_i \mapsto [\![E_i]\!](r)\}_{i=1}^{k}\Big] \cdot B(r).$$

Equivalently (bag-sum form),

$$B' \;=\; \biguplus_{r \in \mathrm{supp}(B)} B(r) \cdot \Big\{\, r \cup \{n_i \mapsto [\![E_i]\!](r)\}_{i=1}^{k} \,\Big\}.$$

Projection. Projection π_X onto a set (or list) X of field names is the bag-valued push-forward along record restriction:

$$\pi_X(\langle \tau_{\mathrm{rec}}, B \rangle) \;=\; \langle \tau_{\mathrm{rec}} \restriction_X,\; B_X \rangle, \qquad B_X(r_X) \;:=\; \sum_{r:\; r \restriction_X = r_X} B(r).$$

In particular, projection $\pi_{n_1,\ldots,n_k}$ onto the field names $n_1, \ldots, n_k$ is the special case that keeps exactly these fields and discards all others.

Selection. $\sigma_\varphi(\langle \tau_{\mathrm{rec}}, B\rangle)$ keeps those r for which $[\![\varphi]\!]^{\mathsf{tv}}(r) = \top$; $\bot$ and U are rejected, i.e.,

$$\sigma_\varphi(\langle \tau_{\mathrm{rec}}, B\rangle) \;=\; \langle \tau_{\mathrm{rec}}, B_\sigma\rangle, \qquad B_\sigma(r) \;:=\; \begin{cases} B(r) & \text{if } [\![\varphi]\!]^{\mathsf{tv}}(r) = \top, \\ 0 & \text{otherwise.} \end{cases}$$

Name-compatible join (natural join). Let $\mathrm{bag}(T_i) = \langle \tau_i, B_i\rangle$ for $i \in \{1,2\}$, and let $S = \mathrm{dom}(\tau_1) \cap \mathrm{dom}(\tau_2)$. Records r_1, r_2 are *naturally joinable* iff, for every $n \in S$, their fields have equal values in the value-level sense used by the GQL definition of natural joinability. This is not an evaluation of the three-valued GQL comparison predicate; in particular, two fields containing the null value have equal values. Define

$$\begin{aligned} \mathrm{bag}(T_1 \bowtie T_2) &= \langle \tau_1 \cup \tau_2, B_\bowtie\rangle, \\ B_\bowtie(r) &:= \sum_{\substack{r_1,r_2 \text{ naturally joinable} \\ r = r_1 \cup r_2}} B_1(r_1) B_2(r_2). \end{aligned}$$

Theta-join $\bowtie_\theta$ is $\bowtie$ followed by σ_θ.

Semi/anti join. The semi-join $\mathrm{BT}_1 \ltimes_\theta \mathrm{BT}_2$ keeps those rows of BT_1 that have at least one join partner in BT_2 satisfying θ. The anti-join $\mathrm{BT}_1 \not\ltimes_\theta \mathrm{BT}_2$ keeps exactly those rows of BT_1 that do not have any join partner in BT_2 satisfying θ.

Union and duplicate control. If $\mathrm{bag}(T_i) = \langle \tau, B_i\rangle$ for $i \in \{1,2\}$, UNION ALL is a bag union:

$$\mathrm{bag}(T_1 \text{ UNION ALL } T_2) = \langle \tau,\ B_1 \uplus B_2\rangle$$

on a common type τ, after any permitted coercion to that common type. UNION (distinct) is set union over records (multiplicity truncated to 0/1).

Distinct (deduplication). DISTINCT(BT) maps multiplicities $m \geq 1$ to 1.

Ordering and slicing. ORDER $\mathrm{BY}_{\vec{\sigma}}$(BT) enriches BT with an order descriptor (it does not change the multiset). For integers $k, \ell \geq 0$, SKIP k and LIMIT ℓ are applied to the emission of results in that order: SKIP k discards the first k rows of the ordered sequence, and LIMIT ℓ keeps at most ℓ subsequent rows.

Constraint (primitive result statement). In a `RETURN` result with `ORDER BY`, `OFFSET`, or `LIMIT`, (i) sort keys MUST NOT contain subqueries; (ii) order-by references are restricted to the statement's admissible *ORDER_REFS* (as determined by preceding `DISTINCT`/`GROUP BY` and the projection); (iii) availability of `OFFSET`/`LIMIT` is governed by conformance options (e.g., `ORDER BY` and page statement: `OFFSET` clause and `ORDER BY` and page statement: `LIMIT` clause).

Unwind. For a list-valued field L of declared type $\mathsf{LIST}\langle\tau_{\text{elem}}\rangle$ and target name n,

$$\text{bag}(\text{UNWIND}_{L\mapsto n}(T)) = \left\langle \tau_{\text{rec}} \cup \{n \mapsto \tau_{\text{elem}}\},\ B' \right\rangle$$

where each record with $L = [v_1, \ldots, v_m]$ yields m records with $n := v_i$. If $L = \mathsf{null}$, that incoming record produces no output record.

Note

`UNWIND` corresponds to the `FOR` statement in the standard; availability of `WITH ORDINALITY` / `WITH OFFSET` bindings and iteration over binding tables is controlled by conformance options (e.g., `FOR` statement: `WITH ORDINALITY`, `FOR` statement: binding table support, and `FOR` statement: `WITH OFFSET`).

Chapter 7 — Takeaways

- **Purpose:** Formalized the pattern layer of GQL as the primary mechanism for selecting and relating graph elements.
- **Introduced:** Graph- and path-pattern syntax; the notion of a match together with extracted variable bindings; variable exposure and safety rules; binding-table formation for pattern results.
- **Key distinctions:** Exposed versus non-exposed variables (participating in joins but not necessarily projected); finite versus potentially unbounded match spaces and the role of finiteness conditions for path patterns.
- **Consequences:** Determined result multiplicities at the semantic level; ensured well-defined, finite outputs under stated restrictions; provided the exact intermediate representation consumed by the expression and algebra layers.
- **Bridge:** Fed directly into expression evaluation, which interpreted filters and computed derived values over bindings.

Part III

Expressions and Algebra

8 Expressions and Evaluation

This chapter specifies how GQL expressions are typed and evaluated with respect to the current database state (store state) and the relevant environments. It makes explicit the interaction between value domains, implicit casts, errors, and three-valued logic, and clarifies how nulls affect the outcome of expression evaluation.

8.1 Expression Domains

Let Ω denote the *store state*. A *row environment* for evaluating expressions in the context of a record r is a partial mapping

$$\eta_r : \mathbb{A} \rightharpoonup \mathbb{V} \quad \text{with} \quad \eta_r(n) = r(n) \text{ for all } n \in \mathrm{dom}(r),$$

and undefined for $n \notin \mathrm{dom}(r)$. A *global environment* η_g binds constants (catalog objects, procedures, named graphs, types, etc.). An *evaluation environment* is the pair $\langle \eta_g, \eta_r \rangle$.

8.2 Typing and Evaluation

Typing and casting. Let $\vdash E : \tau$ denote the statically derived typing judgment that expression E has type $\tau \in \mathbb{T}$. A (partial) implicit cast relation $\rightsquigarrow \subseteq \mathbb{T} \times \mathbb{T}$ determines where a value of type τ may be coerced to τ' at use sites (e.g., integer $\rightsquigarrow$ decimal). We write $v : \tau \rightsquigarrow \tau'$ when the coerced value exists.

Evaluation judgment. Using the abstract error carrier Err declared in Chapter 6, whose tokens include, for example, division-by-zero, type-mismatch, and overflow, we write

$$\langle \Omega, \eta_g, \eta_r \rangle \ \vdash \ E \ \Downarrow \ u$$

where $u \in \mathbb{V} \cup \mathsf{Err}$. Evaluation is total into u but may yield errors (note that $\mathsf{null} \in \mathbb{V}$). When r is a record, η_r is the row environment induced by r, and Ω, η_g are fixed by the surrounding context, we use the abbreviation

$$[\![E]\!](r) = u \quad \Longleftrightarrow \quad \langle \Omega, \eta_g, \eta_r \rangle \vdash E \Downarrow u.$$

For a Boolean expression φ, $[\![\varphi]\!]^{\mathsf{tv}}(r)$ denotes the corresponding member of $\mathbb{B}_3$, or an element of Err, under the same convention.

Evaluation order. ISO/IEC 39075 specifies expression results through the General Rules but does not classify built-in functions as pure, stable, or volatile. Unless precedence, parentheses, or a specific General Rule determines otherwise, effective evaluation is generally from left to right; whether expressions are actually evaluated in that order is implementation-dependent, particularly when an operand can raise a condition or when the result can be obtained without evaluating every part. Consequently, no stronger repeatability or evaluation-order guarantee is assumed here than the guarantee stated for the particular expression in the standard.

8.3 Truth Values and `NULL`

Three-valued logic. Boolean contexts range over the three-valued domain fixed in Section 6.1,

$$\mathbb{B}_3 = \{\top, \bot, \mathsf{U}\},$$

read as *true*, *false*, and *unknown*, respectively. The connective tables follow Kleene's strong three-valued logic; for example:

$$\neg\mathsf{U} = \mathsf{U},$$
$$\top \wedge \mathsf{U} = \mathsf{U},$$
$$\bot \wedge \mathsf{U} = \bot,$$
$$\text{etc.}$$

In concrete GQL, the abstract truth values $\top$ and $\bot$ correspond to the Boolean constants `TRUE` and `FALSE`, and U is represented by the Boolean *null* value, as specified in the standard; here, $\mathbb{B}_3$ is used as an abstract carrier for the three-valued semantics. A comparison $x\ \theta\ y$ ($\theta \in \{=, \neq, <, \leq, >, \geq\}$) evaluates to U if either operand is null. Predicates such as `IS NULL` and `IS NOT NULL` are two-valued (they never yield U).

Example (a library evaluation judgment). Let $er_{\mathsf{inc},2} \in \mathbb{V}$ denote an edge reference value whose referent is the second `INCLUDES` edge $e_{\mathsf{inc},2}$. Let the current row bind that reference:

$$\eta_r(\mathtt{i}) = er_{\mathsf{inc},2}, \qquad \rho_E(e_{\mathsf{inc},2})(\mathtt{late_fee}) = 2.5.$$

For the library store state Ω_{lib},

$$\langle \Omega_{\mathsf{lib}}, \eta_g, \eta_r \rangle \vdash \mathtt{i.late_fee > 0.0} \Downarrow \top.$$

Ordering and sorting. For each base type whose values admit ordering, let $\preceq$ denote the corresponding weak order. Values of different types are compared only when their types are comparable, possibly after applying permitted implicit casts to a common comparable type. If universal comparison is supported, values that are not otherwise comparable may

instead be ordered according to an implementation-defined total order that remains stable throughout the execution of a GQL request.

The ordering used by `ORDER BY` additionally determines the placement of null through either `NULLS FIRST` or `NULLS LAST`. This placement rule is specific to sorting: comparisons of null with a material value using $<$ or $>$ yield U, rather than treating null as smaller or greater than that value.

Domains. Let Var denote the (countable) universe of graph-pattern and scalar variables. A (variable) *mapping* is a partial function

$$\mu : \mathsf{Var} \rightharpoonup \mathbb{V}.$$

We thus distinguish two forms of absence: (i) *unbound* variables, i.e., those $x \in \mathsf{Var} \setminus \mathrm{dom}(\mu)$ for which $\mu(x)$ is undefined, and (ii) the distinguished constant $\mathsf{null} \in \mathbb{V}$ among scalar values. Expressions may evaluate to a value $v \in \mathbb{V}$ (including null), or they may raise a runtime *error* err.

Truth values and evaluation. A predicate (Boolean) expression ϕ is evaluated under a mapping μ and graph G as

$$[\![\phi]\!]^{\mathsf{tv}}_{G,\mu} \in \mathbb{B}_3 \cup \{\mathsf{err}\}.$$

Atomic comparisons use SQL-style three-valued logic:

$$[\![e_1{=}e_2]\!]^{\mathsf{tv}}_{G,\mu} = \begin{cases} \top & \text{if the equality rules establish that the values are equal,} \\ \bot & \text{if the equality rules establish that the values are unequal,} \\ \mathsf{U} & \text{otherwise, because a required comparison is unknown} \end{cases}$$

(for essentially comparable operands).

In particular, the result is U if either operand is null. Equality of lists and records is determined recursively, so a material list or record containing a null value can also produce U: an unequal component decides $\bot$, all components equal decide $\top$, and an otherwise undecided component comparison gives U. The other comparison operators follow their standard transformations and ordering rules; non-comparable operands raise the specified data exception. If evaluation of a required sub-expression raises err, evaluation of the comparison raises the corresponding dynamic exception. The treatment of an error in an inessential part of a containing expression follows the implementation-dependent rule described below.

Connectives. We use Kleene's strong truth tables:

$\wedge$	$\top$	U	$\bot$
$\top$	$\top$	U	$\bot$
U	U	U	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\vee$	$\top$	U	$\bot$
$\top$	$\top$	$\top$	$\top$
U	$\top$	U	U
$\bot$	$\top$	U	$\bot$

$$\neg\top = \bot,\ \neg\bot = \top,\ \neg\mathsf{U} = \mathsf{U}.$$

These tables determine the result but do not impose operational short-circuiting. For example, after $\bot$ in a conjunction or $\top$ in a disjunction, the other operand is inessential to the truth value. If evaluating such an inessential part would raise an exception, ISO/IEC 39075 leaves it implementation-dependent whether that exception is raised.

Null-intolerant and null-tolerant predicates. The null behavior of each built-in is determined by its rules in the standard. As derived terminology in this formalization, a function f is *strict* if any null argument yields null. A predicate p is *null-intolerant* if any null argument yields U. Non-strict/total built-ins may map null to a concrete value (e.g., coalesce).

Null predicates for composites (conformance). For list and record values the null check also includes `IS [NOT] CONTAINING NULL`, which inspects elements/fields for the presence of null; availability of this predicate follows the corresponding conformance feature in the GQL standard.

`SAME` predicate. The optional `SAME` predicate (Feature G114) takes at least two permitted arguments; generalized arguments require Feature G117. If an argument is null, it raises 22004, and if two arguments are not comparable, it raises 22G04. Otherwise, its result is $\top$ exactly when every pair of arguments compares equal, and $\bot$ otherwise.

Filtering. For a bag $\mathcal{B}$ of mappings, the `WHERE` filter ϕ is

$$\mathrm{filter}_{\phi}(\mathcal{B}) \;:=\; \biguplus_{\mu\in\mathcal{B}} \begin{cases} \mu & \text{if } [\![\phi]\!]^{\mathsf{tv}}_{G,\mu} = \top, \\ (\text{omit}) & \text{if } [\![\phi]\!]^{\mathsf{tv}}_{G,\mu} \in \{\bot, \mathsf{U}\} \end{cases}$$

and is *undefined* (signals a dynamic exception) if $[\![\phi]\!]^{\mathsf{tv}}_{G,\mu} = \mathsf{err}$ for any μ.

Chapter 8 — Takeaways

- **Purpose:** Specified how expressions were typed and evaluated relative to a graph store state and variable environments.
- **Introduced:** A static typing discipline for expressions; evaluation rules for arithmetic, comparisons, and predicates; a three-valued truth model driven by unknown values.
- **Key distinctions:** Static typing versus dynamic evaluation; ordinary Boolean reasoning versus three-valued reasoning in the presence of unknowns.
- **Consequences:** Made filtering semantics precise under unknowns and errors; identified the specified implementation-defined and implementation-dependent evaluation choices; provided reusable evaluation machinery for grouping, ordering, and updates.
- **Bridge:** Enabled the algebraic layer to treat queries as compositions of operators over binding tables.

9 Algebra of Binding Tables

This chapter introduces the algebra of binding tables, which forms the intermediate value domain on which query blocks and statements operate. We treat binding tables as bags of records and present selection, projection, joins, unwinding, and subquery composition as operations that transform them. The algebra is formulated with explicit bag semantics, three-valued logic, and ordering metadata, in a manner that aligns with the GQL standard's treatment of optional pattern matching, correlation, and primitive result statements.

Classical bag-query algebras provide background on duplicates, aggregation, nesting, and expressive power (Grumbach and Milo 1996; Libkin and Wong 1997). They serve here only as mathematical background; binding-table descriptors, ordering metadata, preferred column names, and the operator rules remain governed by GQL.

9.1 Selection and Projection

Let $T \in \mathsf{BT}$ be a binding table with $\mathrm{bag}(T) = \langle \tau_{\mathrm{rec}}, B \rangle$. The formulas below act on this bag projection; each operator separately preserves or updates the ordered/unordered and preferred-column metadata of the complete binding-table descriptor. Projecting actions are modeled as functions on multisets of records:

- *Selection:* $\sigma_{\varphi}(\mathrm{BT})$ — the submultiset of records satisfying predicate φ (evaluated in GQL types, with null handling according to the comparison semantics and the three-valued truth tables from Chapter 8).
- *Projection:* $\pi_{E_1 \mapsto n_1, \ldots, E_k \mapsto n_k}(\mathrm{BT})$ — each input record occurrence contributes once to a new record r' with fields named n_i taking values $[\![E_i]\!](r)$; multiplicities of equal projected records are summed.
- *Natural join:* $\mathrm{BT}_1 \bowtie \mathrm{BT}_2$ — join based on name compatibility and equal values of shared fields; the bag multiplicity is consistent with the product of multiplicities of the paired records.
- *Ordering:* ordering metadata may be included in the descriptor of a binding table, distinguishing ordered from unordered tables; it does not affect the underlying bag semantics but determines the order in which records are emitted.[2]

[2] Paging operators such as `LIMIT` and `SKIP` (cf. the *order by and page* statement in the GQL standard) are modeled separately as slice operators that may change the underlying bag by discarding rows.

For example, let T_{fees} have the two rows

$$r_1 = \langle \texttt{title} \mapsto \texttt{'Graph DB 101'}, \texttt{late_fee} \mapsto 0.0 \rangle,$$
$$r_2 = \langle \texttt{title} \mapsto \texttt{'Graph Theory'}, \texttt{late_fee} \mapsto 2.5 \rangle.$$

Then $\sigma_{\texttt{late_fee}>0}(T_{\text{fees}})$ retains only r_2, and projecting it onto `title` yields the singleton row $\langle \texttt{title} \mapsto \texttt{'Graph Theory'} \rangle$.

9.2 Joins and `OPTIONAL` (Left-Outer Join)

Compatibility and merge. Mappings μ, ν are *naturally compatible*, written $\mu \sim \nu$, iff their fields with the same name have equal values in the value-level sense used by the GQL definition of natural joinability. This is not an evaluation of the three-valued comparison predicate; in particular, two common fields that both contain the null value have equal values. Their *merge* $\mu \sqcup \nu$ is the union partial map after any required common-type coercion and is well-defined when $\mu \sim \nu$.

Binding-table semantics of `OPTIONAL MATCH`. Let BT be the current table, let Γ' be a pattern, and let Δ'_{gpbt} be its graph pattern binding table with an explicit or implicit `YIELD` clause. In ISO/IEC 39075, `OPTIONAL MATCH` is specified via a translation to an `OPTIONAL CALL` whose `RETURN` list contains only the *new* output bindings. The call is evaluated separately for every occurrence of an incoming record, so the correlation columns must not be projected away before the optional expansion.

More generally, let $\mathcal{F}$ be a correlated binding-table transformer and let C_{new} be the columns in its declared output record type that are not columns of BT. For every occurrence of μ in BT, put

$$\mathsf{Out}_{\mathcal{F}}(\mu) := \mathcal{F}(\{\{\mu\}\})(C_{\text{new}}),$$

where the singleton is an incoming table occurrence and the final parentheses denote projection. Define the correlated optional expansion by

$$\mathsf{OptExpand}_{\mathcal{F}}(\mathrm{BT}) := \biguplus_{\mu \in \mathrm{BT}} \begin{cases} \biguplus_{\nu \in \mathsf{Out}_{\mathcal{F}}(\mu)} \{\mu \sqcup \nu\} & \text{if } \mathsf{Out}_{\mathcal{F}}(\mu) \neq \varnothing, \\ \{\mu \sqcup \{c \mapsto \mathsf{null} \mid c \in C_{\text{new}}\}\} & \text{otherwise.} \end{cases}$$

Thus, with $\mathcal{F} = \mathsf{GPBT}_{\Delta'_{\text{gpbt}}}$, $\mathrm{OPT_MATCH}_{\Gamma'}(\mathrm{BT}) = \mathsf{OptExpand}_{\mathcal{F}}(\mathrm{BT})$. This preserves correlation with each incoming record and fills all statically declared new fields with null exactly when that record has no matching extension.

Bag join. For bags $\mathcal{B}_1, \mathcal{B}_2$ of mappings, define the (natural) bag join

$$\mathcal{B}_1 \bowtie \mathcal{B}_2 \;:=\; \biguplus_{\mu\in\mathcal{B}_1}\; \biguplus_{\nu\in\mathcal{B}_2} \begin{cases} \{\mu \sqcup \nu\} & \text{if } \mu \sim \nu, \\ \varnothing & \text{otherwise.} \end{cases}$$

Guarded join (ON-condition). Given a Boolean guard θ, the guarded join is

$$\mathcal{B}_1 \bowtie_\theta \mathcal{B}_2 \;:=\; \mathrm{filter}_\theta(\mathcal{B}_1 \bowtie \mathcal{B}_2),$$

where θ is evaluated over $\mu \sqcup \nu$. Note that U or $\bot$ in θ discards the pair (no match), $\top$ keeps it. If θ evaluates to err for some pair, the join is undefined (signals a dynamic exception).

Left outer join. Let $\mathrm{hdr}(\mathcal{B})$ be the set of variables in the declared record type of $\mathcal{B}$, including when $\mathcal{B}$ is empty. Let π_{L} project a merged mapping to the left variables $\mathrm{hdr}(\mathcal{B}_1)$, let $\ominus$ denote bag difference on these left projections, and let $\mathsf{pad}_{\mathrm{R}}$ map each remaining left mapping μ to $\mu \sqcup \eta$, where η binds every right-only variable in $\mathrm{hdr}(\mathcal{B}_2) \setminus \mathrm{hdr}(\mathcal{B}_1)$ to null. For $\mathcal{B}_1, \mathcal{B}_2$ and guard θ, the left outer join is the bag

$$\mathcal{B}_1 \;\triangleleft_\theta\; \mathcal{B}_2 \;:=\; (\mathcal{B}_1 \bowtie_\theta \mathcal{B}_2) \;\uplus\; \mathsf{pad}_{\mathrm{R}}\Big(\mathcal{B}_1 \ominus \pi_{\mathrm{L}}\Big(\mathcal{B}_1 \bowtie_\theta \mathcal{B}_2\Big)\Big)\,,$$

Denotational semantics of `OPTIONAL MATCH`. For a basic pattern P and an optional pattern Q with internal guard θ (e.g., a `WHERE`-predicate inside the optional clause), denotation is

$$[\![P \texttt{ OPTIONAL } (Q)]\!]_G \;=\; \mathsf{OptExpand}_{\mathcal{F}_Q}([\![P]\!]_G)\,,$$

where $\mathcal{F}_Q(\{\{\mu\}\}) = [\![Q]\!]_{G,\mu}$ includes the internal guard θ. A separate `FILTER WHERE` ϕ after `OPTIONAL` is applied by filter_ϕ to the outer-join result (hence null-sensitive); a `WHERE` belonging to the optional graph pattern remains inside $\mathcal{F}_Q$.

`OPTIONAL` Block Denotation For a block consisting of a basic pattern P, an optional pattern Q with guard θ, a separate trailing filter ϕ, and a projection $\vec{s}$, let $\mathrm{proj}_{\vec{s}}$ denote the corresponding record projection, including evaluation of its expressions with the null-and-error semantics of Chapter 8 and preservation of bag multiplicities. Then

$$\begin{aligned}&[\![\texttt{MATCH } P \texttt{ OPTIONAL } (Q) \texttt{ FILTER WHERE } \phi \texttt{ RETURN } \vec{s}]\!]_G \\ &\quad=\; \mathrm{proj}_{\vec{s}}\Big(\mathrm{filter}_\phi\Big(\mathsf{OptExpand}_{\mathcal{F}_Q}([\![P]\!]_G)\Big)\Big)\,,\end{aligned}$$

Note

`OPTIONAL` is evaluated per current row; if the optional part yields no rows, a single all-null row is produced on its side before composing with the current row by cartesian product, matching the effect of the standard transformation via `OPTIONAL CALL`.

9.3 `FOR` and List Unnesting

`FOR` on a list is modeled by $\mathrm{unnest}_{n:L}(\mathrm{BT})$. For each occurrence of a record $r \in \mathrm{BT}$, if the field L contains the list $(v_1, \ldots, v_m)$, the operator produces the m records

$$r[n \mapsto v_1], \ldots, r[n \mapsto v_m].$$

Under bag semantics, each generated record inherits the multiplicity of r; thus, multiplicities are multiplied when identical output records arise from different input occurrences or list positions. If L evaluates to null, no output record is generated for r.

9.4 Subqueries and Scope

A *query block* is a triple $\langle \Gamma, \Phi, \Pi \rangle$:

- Γ: (possibly empty) conjunction of graph/path patterns with a `WHERE`-predicate producing a base binding table as in Section 7.1.
- Φ: a sequence of algebraic operators over binding tables (select, extend, join, unwind, group, distinct, order, slice).
- Π: a final projection (RETURN-style) specifying visible columns and expressions to emit.

The grouping, ordering, and slicing stages are defined in Chapters 12 and 13. For a store state Ω, let $\mathrm{BT}_\Gamma(\Omega)$ denote the base binding table obtained by evaluating Γ in the working graph and correlation environment of Ω; for empty Γ, it is the unit binding table. The denotation of a block is

$$[\![\langle \Gamma, \Phi, \Pi \rangle]\!](\Omega) \;=\; \Pi\Big(\,\Phi(\mathrm{BT}_\Gamma(\Omega))\,\Big).$$

Example (query-block decomposition). For the second query in Section 4.4, which computes `total_late_fee`, let Γ_{fee} denote its two path patterns together with the two ISBN conditions in their `WHERE` clause. In the abstraction above,

$$\begin{aligned}
\Phi_{\mathrm{fee}} &= \sigma_{\texttt{total_late_fee}>0.0} \circ \mathrm{EXT}_{(\texttt{i1.late_fee}+\texttt{i2.late_fee}) \mapsto \texttt{total_late_fee}},\\
\Pi_{\mathrm{fee}} &= \pi_{\substack{\texttt{l.loanid} \mapsto \texttt{loanid},\\ \texttt{total_late_fee} \mapsto \texttt{totalLateFee}}},\\
\mathsf{QB}_{\mathrm{fee}} &= \langle \Gamma_{\mathrm{fee}}, \Phi_{\mathrm{fee}}, \Pi_{\mathrm{fee}} \rangle.
\end{aligned}$$

Thus the graph-pattern `WHERE` belongs to Γ_{fee}; `LET` and the subsequent `FILTER WHERE` determine Φ_{fee}, while `RETURN` determines Π_{fee}.

Linear-query pipelining. ISO/IEC 39075 does not use a `WITH` clause as a query-block boundary. Instead, the incoming and outgoing working record and working table types are threaded from each simple query statement to the next in a linear query statement. Projection and renaming are performed by the applicable result statement, and only the columns of its outgoing working table type are visible to a following statement.

Subqueries. A subquery evaluates to a binding table in the current Ω. When the subquery is correlated, references to names from the enclosing query are resolved in the current outer-row environment. The resulting binding table is consumed according to the surrounding evaluation context, as specified below for relational, scalar, and row-producing subqueries.

Environments and scopes. A *static environment* $\mathcal{E}$ maps names (variables, parameters, functions, and relevant catalog objects) to denotations. A *dynamic environment* Dyn provides the current graph G and the current outer-row mapping μ_o during correlated evaluation.

Uncorrelated subquery. A subquery Q that contains no reference to an outer binding is independent of the outer row. Its result is consumed according to its surrounding context; the additional static restrictions for a value query expression are stated below under scalar subqueries.

Correlated subquery. A subquery referencing outer attributes is a function of the outer row:

$$[\![Q]\!](\mu_o) \;=\; [\![Q]\!]_{\mathcal{E},\mathsf{Dyn}=(G,\mu_o)}.$$

The `EXISTS` context consumes the resulting bag as follows:

$$\text{EXISTS}(Q)(\mu_o) = \begin{cases} \top & \text{if } [\![Q]\!](\mu_o) \neq \varnothing, \\ \bot & \text{otherwise.} \end{cases}$$

Row-producing subqueries. In the `FROM`/pattern position, a subquery Q contributes its bag of rows to the surrounding algebra just like a base pattern.

Dynamic-context notation. In the formulas below, (G, μ_o) abbreviates the corresponding components of Dyn; function and procedure registries continue to be resolved through the static environment $\mathcal{E}$. The outer mapping μ_o may be empty.

Variable hygiene. Each subquery has its own local variable namespace V_{loc} disjoint from outer variables by α-renaming, except for explicitly correlated references to outer names, which are resolved in μ_o. Shadowing of names by aliases is disallowed unless explicitly stated.

Relational subqueries. A subquery Q used by an `EXISTS` predicate is evaluated as a bag $\mathcal{B}_Q = [\![Q]\!]_{G,\mu_o}$ in environment (G, μ_o). Here μ_o is the current left-row mapping.

$$[\![\text{EXISTS}(Q)]\!]^{\mathsf{tv}}_{G,\mu_o} = \begin{cases} \top & \text{if } \mathcal{B}_Q \neq \varnothing, \\ \bot & \text{if } \mathcal{B}_Q = \varnothing \end{cases}$$

(never U); errors from inside the subquery propagate to the enclosing expression.

Scalar subqueries. A scalar subquery is a value query expression `VALUE` Q. Its nested query has a single result column, and its emitting `RETURN` is statically required either to use `LIMIT 1` or to contain an aggregate without `GROUP BY`; hence its result has at most one record. Formally, with $\mathcal{B}_Q = [\![Q]\!]_{G,\mu_o}$:

$$[\![\texttt{VALUE } Q]\!]_{G,\mu_o} = \begin{cases} \mathsf{null} & \text{if } \mathcal{B}_Q = \varnothing, \\ v & \text{if the sole field of the sole record of } \mathcal{B}_Q \text{ has value } v. \end{cases}$$

Subquery in `OPTIONAL`. When Q is correlated and appears on the right of `OPTIONAL`, the correlation uses the left-row mapping μ at the point of join; each μ yields $[\![Q]\!]_{G,\mu}$ independently, composing via $\triangleleft$ as above.

9.5 Whole-Query Denotation

A full read-only query is a pipeline of blocks

$$Q = \langle B_1;\ B_2;\ \ldots;\ B_m \rangle.$$

Each block is evaluated with respect to the fixed store state Ω and transforms the binding table produced by its predecessor. Let $\text{BT}_0 = \text{BT}_{\text{unit}}$, and, for $1 \leq i \leq m$, define

$$\text{BT}_i := [\![B_i]\!](\Omega; \text{BT}_{i-1}).$$

The denotation of the complete query is therefore

$$[\![Q]\!](\Omega) := \text{BT}_m.$$

Equivalently,

$$[\![Q]\!](\Omega) = [\![B_m]\!]\Big(\Omega; [\![B_{m-1}]\!](\Omega; \cdots [\![B_1]\!](\Omega; \mathrm{BT}_{\mathrm{unit}}) \cdots)\Big).$$

Example (linear-query pipeline). The linear-composition query in Section 4.4 instantiates this recurrence with four successive transformations. Let $\Delta_{\mathrm{gpbt},1}$ denote the first `MATCH` graph pattern together with its last-name `WHERE` predicate and its implicit `YIELD` clause, let $\Delta_{\mathrm{gpbt},2}$ denote the corresponding graph pattern binding table for the second `MATCH`, and let

$$\varphi \equiv \texttt{i.due_date < DATE '2025-12-31'}$$

be the predicate of the subsequent `FILTER WHERE`. At the level of the abstraction above, $Q_{\mathrm{lin}} = \langle B_1; B_2; B_3; B_4 \rangle$, where

$$\begin{aligned}
\mathrm{BT}_1 &= \mathsf{GPBT}_{\Delta_{\mathrm{gpbt},1}}(\mathrm{BT}_{\mathrm{unit}}),\\
\mathrm{BT}_2 &= \mathsf{GPBT}_{\Delta_{\mathrm{gpbt},2}}(\mathrm{BT}_1),\\
\mathrm{BT}_3 &= \sigma_{\varphi}(\mathrm{BT}_2),\\
\mathrm{BT}_4 &= \mathrm{order}_{\texttt{due_date}\uparrow}\left(\pi_{\substack{\texttt{l.loanid}\mapsto\texttt{loanid},\texttt{b.title}\mapsto\texttt{title},\\ \texttt{i.due_date}\mapsto\texttt{due_date},\texttt{i.late_fee}\mapsto\texttt{late_fee}}}(\mathrm{BT}_3)\right).
\end{aligned}$$

Here $\mathrm{order}_{\texttt{due_date}\uparrow}$ denotes ascending ordering by the projected `due_date` field. Consequently, $[\![Q_{\mathrm{lin}}]\!](\Omega) = \mathrm{BT}_4$.

An updating statement is modeled separately because its evaluation may change the store state. Given an input binding table BT, its denotation has the form

$$[\![U]\!](\Omega; \mathrm{BT}) = \langle \Omega', R \rangle,$$

where Ω' is the resulting store state and R is either a result binding table or an omitted result. When no preceding query blocks supply bindings to the update, BT is typically the unit binding table.

Soundness checks and algebraic laws (selected) Let $\equiv$ denote bag-equivalence of binding tables (same records with same multiplicities, ignoring order). The following laws are intended only when every relevant expression evaluation yields the same value and raises no dynamic exception:

- $\sigma_{\varphi_1}\Big(\sigma_{\varphi_2}(\mathrm{BT})\Big) \equiv \sigma_{\varphi_1 \wedge \varphi_2}(\mathrm{BT})$.
- $\pi_X\Big(\sigma_{\varphi}(\mathrm{BT})\Big) \equiv \sigma_{\varphi|_X}\Big(\pi_X(\mathrm{BT})\Big)$ when φ mentions only X.
- $\bowtie$ is associative and commutative up to field renaming and multiplicity algebra.
- $\mathrm{DISTINCT}(\mathrm{DISTINCT}(\mathrm{BT})) \equiv \mathrm{DISTINCT}(\mathrm{BT})$.

9.6 Composite Queries and Set Operators

A *composite query* combines the results of one or more linear query pipelines using set-like operators `UNION`, `EXCEPT`, `INTERSECT` and the fallback operator `OTHERWISE`. At the semantic level, a composite query expression denotes a binding table obtained by composing the binding tables produced by its constituent linear queries.

Syntactic shape. The grammar fragment

```
composite-query ::= query-part { query-conj query-part }
```

introduces a left-associative composition of `query-parts`. Each `query-part` is a `linear-query`. A `query-conj` is one of `UNION`, `EXCEPT`, `INTERSECT`, or `OTHERWISE`, with the set operators optionally qualified by a set quantifier `ALL` or `DISTINCT`.

In line with the GQL standard, the following constraints are imposed:

- If a set operator (`UNION`, `EXCEPT`, `INTERSECT`) is specified without an explicit set quantifier, the quantifier `DISTINCT` is implicit.
- Within a single `composite-query`, all occurrences of `query-conj` must be the same query conjunction: the same operator and, for a set operator, the same `ALL`/`DISTINCT` choice after applying the implicit `DISTINCT`. Thus neither mixed operators such as `UNION ... EXCEPT ...` nor mixed quantifiers such as `UNION ALL ... UNION DISTINCT ...` are permitted.

Thus, each well-formed composite query expression has the form

$$Q_0 \ \mathbf{op} \ Q_1 \ \mathbf{op} \ \ldots \ \mathbf{op} \ Q_n$$

for a fixed query conjunction **op**, including its fixed set quantifier when applicable, and $n \geq 0$, and is interpreted as left-associative:

$$(((Q_0 \ \mathbf{op} \ Q_1) \ \mathbf{op} \ Q_2) \ \ldots \ \mathbf{op} \ Q_n).$$

Typing and column combination. Each `linear-query` is a (possibly focused) linear query statement in the sense of the standard. We assume that evaluation of a linear query L produces a binding table $T_L = [\![L]\!] \in \mathsf{BT}$ whose bag projection is

$$\mathrm{bag}(T_L) = \langle \tau_L, B_L \rangle,$$

where τ_L is a record type describing the columns and their value types, and B_L is a finite multiset of records of type τ_L.

Let $Q_0, \ldots, Q_n$ be the query parts of a composite query, $T_i = [\![Q_i]\!]$, and $\mathrm{bag}(T_i) = \langle \tau_i, B_i \rangle$ (these are the results of the immediately contained linear queries). Typing of the composite query is governed by the following constraints, mirroring Subclause 14.2:

- All τ_i must be *column name-equal*: they have the same set of column names.
- The τ_i must be *column-combinable*, in the sense of the type-combination relation: for each column name, the corresponding value types in τ_i admit a combined type, possibly a dynamic union type.
- Let τ be the combined record type obtained from $\tau_0, \ldots, \tau_n$. The declared type of the composite query is the material binding table type with column type τ.

Operationally, each operand $T_i = [\![Q_i]\!]$ can be coerced to the common type τ before applying the set operator, using the same coercion and dynamic-union rules as for column-combinable record types. When the operator is one of `UNION DISTINCT`, `EXCEPT ALL`, `EXCEPT DISTINCT`, `INTERSECT ALL`, `INTERSECT DISTINCT`, the standard additionally requires that no column in τ has a dynamic union type that mixes incomparable value types; otherwise, the composite query fails static validation. In these cases, every column acts as a grouping key, and the general grouping rules apply to the composite query.

Bag-based semantics of set operators. We now describe the algebraic semantics in terms of the binding-table algebra. Recall that a binding table is the descriptor-bearing object in BT fixed in Section 6.1. The equations below determine its bag projection $\langle \tau, B \rangle$; the ordered/unordered indicator and preferred column-name sequence remain components of the complete result descriptor and are determined by the corresponding operation. Bag union is written $B_1 \uplus B_2$, with multiplicities added element-wise.

For a binary composite expression of the form Q_L $\mathbf{op}_q$ Q_R, where Q_L and Q_R denote, respectively, the left and right query part, with set operator

$$\mathbf{op} \in \{\texttt{UNION}, \texttt{EXCEPT}, \texttt{INTERSECT}\}$$

and quantifier $q \in \{\texttt{ALL}, \texttt{DISTINCT}\}$ (with `DISTINCT` as the default), let $T_L = [\![Q_L]\!]$ and $T_R = [\![Q_R]\!]$, with $\mathrm{bag}(T_L) = \langle \tau, B_L \rangle$ and $\mathrm{bag}(T_R) = \langle \tau, B_R \rangle$ after coercion to the common type τ. For any record r of type τ, write $M = B_L(r)$ and $N = B_R(r)$ for its multiplicities in the left and right operand (with $M, N \geq 0$).

UNION ALL.

The result is the bag union

$$\mathrm{bag}([\![Q_L \ UNION \ ALL \ Q_R]\!]) = \langle \tau, B \rangle \quad \text{with} \quad B(r) = M + N.$$

This is the *bag* analog of GQL `UNION ALL`: duplicates from both operands are preserved, and their multiplicities added.

UNION (distinct).

The result is the set-union of the supports of the two bags:

$$B(r) = \begin{cases} 1 & \text{if } M + N > 0, \\ 0 & \text{otherwise.} \end{cases}$$

Equivalently, $[\![Q_L \; UNION \; Q_R]\!]$ is obtained by applying the DISTINCT operator to $[\![Q_L \; UNION \; ALL \; Q_R]\!]$.

EXCEPT ALL.

The result is bag difference:

$$B(r) = \max(M - N, 0).$$

Intuitively, duplicates of r in the right operand cancel duplicates in the left operand, but never below zero.

EXCEPT (distinct).

The result is set difference:

$$B(r) = \begin{cases} 1 & \text{if } M > 0 \text{ and } N = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Equivalently, one can first apply `DISTINCT` to both operands and then form their `EXCEPT ALL` bag difference.

INTERSECT ALL.

The result is bag intersection:

$$B(r) = \min(M, N).$$

Each duplicate appears as many times as it is present in both operands.

INTERSECT (distinct).

The result is set intersection:

$$B(r) = \begin{cases} 1 & \text{if } M > 0 \text{ and } N > 0, \\ 0 & \text{otherwise.} \end{cases}$$

Again, this can be seen as `INTERSECT ALL` followed by `DISTINCT`.

For any of the distinct variants, and for `EXCEPT ALL` and `INTERSECT ALL`, if a record of the left operand and a record of the right operand are not comparable values, evaluation raises the data exception “values not comparable” (22G04) instead of producing a binding table.

Fallback semantics of `OTHERWISE`. The operator `OTHERWISE` is a fallback query conjunction rather than a set operator. For a binary composite

$$Q_L \text{ OTHERWISE } Q_R,$$

where Q_L and Q_R denote the left and right operands, respectively, let $T_L = [\![Q_L]\!]$ and $T_R = [\![Q_R]\!]$, with

$$\mathrm{bag}(T_L) = \langle \tau, B_L \rangle \qquad \text{and} \qquad \mathrm{bag}(T_R) = \langle \tau, B_R \rangle,$$

where τ is their common combined type. Then

$$[\![Q_L \text{ OTHERWISE } Q_R]\!] = \begin{cases} T_L & \text{if } B_L \neq \varnothing, \\ T_R & \text{otherwise.} \end{cases}$$

For a longer chain

$$Q_0 \text{ OTHERWISE } Q_1 \text{ OTHERWISE } \cdots \text{ OTHERWISE } Q_n,$$

the result is the binding table produced by the leftmost query operand whose result is non-empty, or the empty binding table of the combined type if all operands produce empty results. The same column-combination discipline as for the set operators applies; no duplicate elimination or bag arithmetic is performed. This describes result selection, not short-circuit evaluation: the General Rules for a composite query expression obtain the results of its query parts before choosing between them, so an operand is not skipped merely because an earlier result is non-empty.

Folding over multiple query parts. Let a composite query be written, after enforcing the uniform-operator rule, as

$$Q_0 \ \mathbf{op}\ Q_1 \ \mathbf{op}\ \ldots\ \mathbf{op}\ Q_n,$$

and let $\mathsf{Comb_op}, q$ be the binary operation on binding tables induced by the chosen operator and quantifier (one of the six set operations, or the `OTHERWISE` fallback). Then the denotation of the composite query is the left fold

$$[\![Q_0 \ \mathbf{op}\ \ldots\ \mathbf{op}\ Q_n]\!] = \mathsf{Comb_op}, q\Big(\ldots \mathsf{Comb_op}, q([\![Q_0]\!], [\![Q_1]\!]), \ldots, [\![Q_n]\!]\Big).$$

This algebraic account is extensionally equivalent to the stepwise binding table construction described for the *composite query expression* in the GQL standard and

provides a compact description of how composite queries act on the intermediate binding tables produced by linear query statements.

Chapter 9 — Takeaways

- **Purpose:** Defined the compositional algebra over binding tables as the intermediate semantic domain of queries.
- **Introduced:** Binding tables as bags of records with a fixed header; core operators such as selection, projection, joins, optional matching, unwinding/construction; a clause-by-clause denotation of query blocks.
- **Key distinctions:** Inner join versus optional (left-outer) join and the resulting padding with unknowns; bag (multiset) behavior versus set behavior and where duplicates were preserved or eliminated.
- **Consequences:** Made queries compositional and mechanically translatable into algebra; explained how optional matching changed both shapes and multiplicities of results; established a reusable semantic engine for subqueries and set operators.
- **Bridge:** Prepared the path chapters by fixing how path results were represented and combined with other bindings.

Part IV

Paths and Regular Path Expressions

10 Path Semantics

This chapter formalizes paths as first-class values in the GQL data model and explains how different path modes constrain the shape of walks through a graph. It describes the interaction between path modes, selective search prefixes, and accessors, and clarifies how path values are constructed, combined, and projected back into binding tables.

10.1 Modes: Walk, Trail, Simple, Acyclic

Paths in a graph. Given a graph

$$G = \langle V, E, \lambda_V, \rho_V, \lambda_E, \rho_E, \mathrm{src}, \mathrm{dst}, \mathrm{dir} \rangle,$$

a *path in* G is a finite alternating sequence

$$w = \langle v_0, e_1, v_1, \ldots, e_n, v_n \rangle \in \mathsf{Path}(G) \quad (n \geq 0)$$

such that for every $i \in \{1, \ldots, n\}$, the edge e_i connects v_{i-1} and v_i. More explicitly, if e_i is directed, either $(\mathrm{src}(e_i), \mathrm{dst}(e_i)) = (v_{i-1}, v_i)$ or $(\mathrm{src}(e_i), \mathrm{dst}(e_i)) = (v_i, v_{i-1})$; its orientation in the path is respectively left-to-right or right-to-left. If e_i is undirected, its endpoints are v_{i-1} and v_i, without a directed orientation. Which of these possibilities satisfies a particular edge pattern is determined by that pattern's directionality constraint. We write $\mathrm{head}(w) = v_0$, $\mathrm{tail}(w) = v_n$, $|w| = n$ (number of edges), $\mathrm{Edges}(w) = \langle e_1, \ldots, e_n \rangle$ as a list, and $\mathrm{Nodes}(w) = \langle v_0, \ldots, v_n \rangle$ as a list.

Walk, trail, simple, acyclic.

- w is a *walk* always.
- w is a *trail* iff all e_i are pairwise distinct.
- w is *simple* iff the node sequence has no repetitions *except possibly* that the first and last node may coincide; i.e., $v_i \neq v_j$ for all $0 \leq i < j \leq n$ with the single allowed case $i = 0$, $j = n$ (so $v_0 = v_n$ is permitted).
- w is *acyclic* iff the node sequence has no repetitions at all (in particular $v_0 \neq v_n$ unless $n = 0$).

For $M \in \{\mathsf{WALK}, \mathsf{TRAIL}, \mathsf{SIMPLE}, \mathsf{ACYCLIC}\}$, define the mode predicate $\mathsf{ok}_M : \mathsf{Path}(G) \to \{\top, \bot\}$ by $\mathsf{ok}_M(w) = \top$ iff w obeys the restriction of M. **Default:** unless explicitly qualified otherwise, the path mode is WALK.

For example, the library-instance path $\langle r, e_{\mathsf{owns}}, a, e_{\mathsf{check}}, l, e_{\mathsf{inc},1}, b_1 \rangle$, where e_{check} is the `CHECK_OUT` edge directed from l to a, is traversed from the reader through the account and loan to the first book. This path has no repeated node or edge. It is therefore a walk, trail, simple path, and acyclic path.

Concatenation of walks. For two paths

$$w = \langle v_0, e_1, \ldots, e_n, v_n \rangle \quad \text{and}$$
$$w' = \langle v'_0, e'_1, \ldots, e'_m, v'_m \rangle$$

with $v_n = v'_0$, their *walk concatenation* is the path

$$w \cdot w' \;=\; \langle v_0, e_1, \ldots, e_n, \underbrace{v_n = v'_0}_{\text{glued}}, e'_1, \ldots, e'_m, v'_m \rangle.$$

This is a purely combinatorial operation on the walk domain and will be used when defining the semantics of regular path expressions.

10.2 Finiteness and Selection

Notation. For a set X, let $\mathcal{P}(X)$ denote its power set and $\mathcal{P}_{\text{fin}}(X)$ its finite subsets.

For a path mode $M \in \{\mathsf{WALK}, \mathsf{TRAIL}, \mathsf{SIMPLE}, \mathsf{ACYCLIC}\}$ and a regular path expression R, we use the preview notation

$$\mathcal{P}_M^R(u, v) \in \mathcal{P}(\mathsf{Path}(G))$$

for the path projection, at endpoints (u, v), of the RPE witness denotation defined in Section 11.1. Thus its members are exactly the paths that satisfy R under direction tests and the mode predicate M. Selective path search prefixes (shortest, all shortest, any, etc.) are maps

$$\mathsf{Sel} : \; \mathcal{P}(\mathsf{Path}(G)) \;\to\; \mathcal{P}_{\text{fin}}(\mathsf{Path}(G))$$

applied independently per (u, v)-partition. The well-formedness rules require every unbounded quantifier to be bounded in effect by a restrictive path mode, a selective path-search prefix, or the `DIFFERENT EDGES` match mode; accordingly, graph-pattern evaluation returns a finite set of reduced matches.

10.3 Path Accessors and Results

Accessors on path values. Let $p \in \mathsf{PathVal} \cup \{\mathsf{null}\}$. We write $\mathsf{PEL}(p)$ for the *path element list* and $\mathsf{len}(p)$ for the *path length*. Formally,

$$\mathsf{PEL}(p) \;:=\; \text{ELEMENTS}(p) \qquad \text{and} \qquad \mathsf{len}(p) \;:=\; \text{PATH_LENGTH}(p).$$

Both accessors are *null-propagating*: if $p = \mathsf{null}$ then $\mathsf{PEL}(p) = \mathsf{null}$ and $\mathsf{len}(p) = \mathsf{null}$.

For a material path value $p = \mathrm{PATH}[\langle n_0, e_1, n_1, \ldots, e_k, n_k\rangle]$, the element list is exactly that alternating list and the length is the number of edges:

$$\mathsf{PEL}(p) = \langle n_0, e_1, n_1, \ldots, e_k, n_k\rangle \qquad \text{and} \qquad \mathsf{len}(p) = k.$$

Malformed representations (non-alternating lists or lists not starting/ending in a node reference) are outside the well-formed path domain; if encountered at runtime, they raise a *malformed path* data exception (rather than yielding null).

From $\mathsf{PEL}(p)$ we derive the node and edge projections as lists:

$$\begin{aligned}\mathsf{nodes}(p) &:= \langle\, x \in \mathsf{PEL}(p) \mid x \text{ is a node reference} \,\rangle,\\ \mathsf{edges}(p) &:= \langle\, x \in \mathsf{PEL}(p) \mid x \text{ is an edge reference} \,\rangle.\end{aligned}$$

When p is material, the following cardinality equalities hold (including the case $\mathsf{len}(p) = 0$):

$$\begin{aligned}\#\,\mathsf{nodes}(p) &= \mathsf{len}(p) + 1,\\ \#\,\mathsf{edges}(p) &= \mathsf{len}(p),\\ \#\,\mathsf{PEL}(p) &= 2 \cdot \mathsf{len}(p) + 1.\end{aligned}$$

We abbreviate the endpoints by

$$\begin{aligned}\mathsf{head}(p) &:= \text{ the first element of } \mathsf{nodes}(p),\\ \mathsf{tail}(p) &:= \text{ the last element of } \mathsf{nodes}(p),\end{aligned}$$

with the understanding that if $p = \mathsf{null}$ (or $\mathsf{len}(p) = \mathsf{null}$) then $\mathsf{head}(p) = \mathsf{tail}(p) = \mathsf{null}$.

Path value construction and concatenation. For node/edge references n_i, e_i forming a path element list $\langle n_0, e_1, n_1, \ldots, e_k, n_k\rangle$,

$$\mathrm{PATH}\Big[\langle n_0, e_1, n_1, \ldots, e_k, n_k\rangle\Big]$$

denotes the corresponding (material) path value.

List helpers. For any finite list $L = \langle x_1, \ldots, x_m\rangle$ with $m \geq 1$, let

$$\mathrm{dropLast}(L) := \langle x_1, \ldots, x_{m-1}\rangle \qquad \text{and} \qquad \mathrm{dropFirst}(L) := \langle x_2, \ldots, x_m\rangle,$$

and let $L +\!\!+ L'$ denote list concatenation.

Given path values p and q, their *value-level concatenation* $p \circ q$ is defined when the last and first elements, respectively, are node reference values whose referent is the same

node, and yields the path value obtained by gluing these endpoints so that the shared node occurs *once*. Otherwise, a *malformed path* exception is raised.

Formally, if either operand is null, then $p \circ q = \mathsf{null}$. Otherwise assume

$$\mathsf{PEL}(p) = \langle n_0, e_1, n_1, \ldots, e_k, n_k \rangle \quad \text{and} \quad \mathsf{PEL}(q) = \langle n'_0, e'_1, n'_1, \ldots, e'_\ell, n'_\ell \rangle,$$

with $k, \ell \geq 0$. If n_k and n'_0 are not node reference values whose referent is the same node, a malformed-path exception is raised. If their referent is the same node but

$$\mathrm{card}(\mathrm{dropLast}(\mathsf{PEL}(p))) + \mathrm{card}(\mathsf{PEL}(q))$$

exceeds the implementation-defined maximum path-element-list cardinality (IL015), data exception 22G10 is raised. Otherwise:

$$\mathsf{PEL}(p \circ q) \;:=\; \mathrm{dropLast}(\mathsf{PEL}(p)) \;+\!\!+\; \mathsf{PEL}(q).$$

Equivalently (expanding the list and showing the glued node once):

$$\mathsf{PEL}(p \circ q) \;=\; \langle n_0, e_1, n_1, \ldots, e_k, \underbrace{n'_0}_{\text{same referent as } n_k}, e'_1, n'_1, \ldots, e'_\ell, n'_\ell \rangle.$$

In particular, whenever both p and q are material and $p \circ q$ is defined,

$$\mathsf{len}(p \circ q) = \mathsf{len}(p) + \mathsf{len}(q).$$

When both p and q are material and correspond to walks w, w', the operation $p \circ q$ mirrors the walk concatenation $w \cdot w'$ defined above.

Result projection (from matches to values). Let a graph pattern be evaluated to a finite set of *reduced matches* (multi-path bindings). Projection of binding variables into a working record proceeds per variable X as follows:

- If X is a *node variable* of singleton degree and is bound to a node v, the projected field value is the node reference value for v. If X is conditional singleton and unbound, the value is null. If X has group degree, the projected value is the list of node reference values in the left-to-right binding order (duplicates retained).
- If X is an *edge variable*, the cases are analogous with edge reference values.
- If X is a *path variable* declared by some path pattern of the graph pattern, the projected field value is the path value p whose $\mathsf{PEL}(p)$ is the left-to-right list of graph element references in the *extracted path* of that binding. If the surrounding statement semantics preserves rows in which that declaring component contributes no match

(e.g., `OPTIONAL MATCH`), then the path variable is projected as null for those rows, analogously to other newly introduced binding variables.

A reference to a path variable always has *singleton* reference degree. Within reduced matches produced by its declaring path pattern component it is *unconditionally* bound; however, statement-level optionalization may yield null in the output binding table for unmatched input rows.

Result typing. Let type(X) denote the declared value type of the projected field for variable X in the current query: *Element variables.* Let X be an element variable. At singleton degree, type(X) is an implementation-determined supertype. It contains the corresponding node/edge reference value type. At conditional singleton degree, the type additionally contains null. At group degree, type(X) is a *group list value type* of node (resp. edge) reference values.

Path variables. If X is a path variable and is unconditionally exposed in the output, then type(X) is the (material) *path value type*. If X may be null due to statement-level optionalization, then type(X) additionally contains null (i.e., it is an Opt($Path$)-like type in the sense of Chapter 6).

Interaction with selection and modes. For each selective path pattern, selection is applied to the candidate path bindings after evaluating any contained `WHERE` conditions and enforcing restrictive path modes (e.g., TRAIL, SIMPLE, ACYCLIC). Variants such as ANY SHORTEST, ALL SHORTEST, SHORTEST N PATHS, and SHORTEST N GROUPS operate on the candidates as specified and yield a finite set of retained bindings; projection then proceeds as above.

Chapter 10 — Takeaways

- **Purpose:** Defined paths as first-class values and fixed how path modes constrained admissible traversals.
- **Introduced:** Path modes such as walk, trail, acyclic, and simple; admissibility conditions for each mode; path accessors that exposed length and component sequences.
- **Key distinctions:** Allowing repetition of edges or nodes versus forbidding it under stricter modes; binding endpoints only versus binding the entire path object.

- **Consequences:** Determined exactly what a path match could return and what the user could inspect; constrained evaluation to avoid ill-defined or explosive results where required; enabled downstream clauses to treat paths as structured values.
- **Bridge:** Set the stage for regular path expressions as the declarative mechanism that generated these path values.

11 Regular Path Expressions

Regular path expressions (RPEs) describe constrained walks through a graph and form the declarative core of GQL's path-matching capabilities. In this chapter, we fix a precise syntax for RPEs, use an auxiliary witness semantics that can record explicit multiset alternation, and connect this semantics to the family of selective shortest-path operators. Graph-pattern evaluation itself returns a set of reduced matches, as specified in Chapter 7.

11.1 Syntax and Denotation

Syntax. Fix a graph $G = \langle V, E, \ldots \rangle$ and a finite alphabet $\Sigma_L \subseteq \mathbb{L}$ of edge-label names. RPEs are generated by:

$$r ::= \varepsilon \mid \ell \mid \ell^{-1} \mid [\nu] \mid \{\eta\} \mid \{\eta\}^{-1} \mid (r \cdot r) \mid (r \cup r) \mid (r \cup_{\mathsf{ms}} r) \mid r^* \mid r^{m..n}$$

where $\ell \in \Sigma_L$, $m, n \in \mathbb{N} \cup \{\infty\}$ with $m \leq n$, $[\nu]$ is a node test to be satisfied at the *current node*, and $\{\eta\}$ and $\{\eta\}^{-1}$ are edge tests to be satisfied by the next directed edge consumed in forward and backward orientation, respectively (see below). Here $\nu : V \to \{\top, \bot\}$ is a two-valued node predicate and $\eta : E \to \{\top, \bot\}$ is a two-valued edge predicate.

Concrete syntax for path expressions. At the concrete-syntax level, regular path expressions appear inside ⟨path pattern⟩ constructs and are written using infix alternation, concatenation by juxtaposition, and postfix quantifiers. Parenthesized subexpressions may carry an attached `WHERE` clause whose search condition is local to the component. The following simplified BNF captures the backbone of this concrete syntax. Here `element-pattern` is an abstract interface for a node or edge pattern; its role in path patterns is described in Section 7.2.

```
path-expr    ::= path-term
                 | path-term "|" path-term ( "|" path-term )*
                 | path-term "|+|" path-term ( "|+|" path-term )*
path-term    ::= path-factor+          /* concatenation */
path-factor  ::= path-primary [ quantifier ]
path-primary ::= element-pattern
                 | "(" [ subpath-var "=" ] path-expr
                       [ where-clause ] ")"
quantifier   ::= "?"
                 | "*"
                 | "+"
```

```
                | "{" int "}"
                | "{" [ int ] "," [ int ] "}"
where-clause ::= "WHERE" search-condition
subpath-var  ::= identifier
element-pattern  ::= node-pattern | edge-pattern
node-pattern     ::= ...  /* node test and optional binding */
edge-pattern     ::= ...  /* edge step, tests, and optional binding */
search-condition ::= ...  /* boolean expressions */
```

Informally, `path-expr` corresponds to the abstract r-grammar above, with `"|"` realizing set alternation, the GQL multiset-alternation token `"|+|"` realizing $\cup_{\mathsf{ms}}$, juxtaposed `path-factor`s realizing concatenation, and the `quantifier` forms mapping to r^*, $r^{m..n}$, and their common shorthands. A subpath variable is a graph-pattern variable used to distinguish reduced matches for multiset alternation; it is not bound to a value and cannot be referenced.

For example, the expression $\texttt{OWNS} \cdot \texttt{CHECK_OUT}^{-1} \cdot \texttt{INCLUDES}$ accepts the library-instance path from the reader through the account and loan to either book.

For every quantified or questioned path primary, the simply contained primary shall have minimum path length greater than zero. In particular, ε^* and any other repetition of a zero-edge primary belong to the auxiliary calculus but are not well-formed concrete GQL path patterns. This condition prevents a single zero-edge witness from acquiring infinitely many derivations through different repetition counts.

Important At the *level of path witnesses alone* (i.e., when one forgets how variables are exposed by element patterns), the postfix quantifier `"?"` behaves like the bounded repetition $\{0, 1\}$: it admits either zero or one occurrence of the subexpression. However, when `"?"` is applied to a `path-primary` that contains `element-pattern`s with bindings, GQL assigns it the status of a *questioned path-pattern primary* in the sense of Chapter 7. In particular, the propagation of *variable exposure degrees* for `"?"` is *not* in general identical to treating it as a mere abbreviation for $\{0, 1\}$: singleton-variable exposure becomes conditional due to the zero-iteration branch, while group-variable exposure is preserved as specified in Table 7.1. Thus, in this book we treat `"?"` as syntactic sugar for $\{0, 1\}$ only in the purely language-theoretic sense of accepted path witnesses, not as an identity of the variable-exposure meta-theory.

An `element-pattern` contributes a single step (possibly with tests on nodes and edges), while a parenthesized `path-expr` with an optional `WHERE` clause behaves as a grouped subpattern whose search condition is evaluated prior to any selective path-search prefix. A non-selective component may use a non-local reference of singleton degree; a selective path pattern may not reference a graph-pattern variable declared outside it.

Step predicates and step relations (for element patterns). To connect the abstract RPE syntax above with the concrete `element-pattern` constructs, we use a generalized notion of a *step predicate*.

Step predicates. A step predicate δ is a quadruple

$$\delta = (\mathsf{orient},\ \Phi_e,\ \Phi_{\mathrm{src}},\ \Phi_{\mathrm{dst}})$$

where $\mathsf{orient} \in \{\rightarrow, \leftarrow, \leftrightarrow\}$ selects forward/backward/undirected traversal (query-time), *distinct* from the graph attribute $\mathrm{dir}(e) \in \{\rightarrow, \leftrightarrow\}$ (data-time directedness); Φ_e is a Boolean predicate over $(e, \lambda_E(e), \rho_E(e))$; and $\Phi_{\mathrm{src}}, \Phi_{\mathrm{dst}}$ are Boolean predicates over endpoint vertices (their labels/properties). Predicates evaluate in the three-valued logic of Section 8.3; U counts as failure for step selection. Reading an absent property yields `null` when its declared property type is nullable; for a material property type it raises 22G03. A surrounding predicate may then evaluate to U as a consequence of a null value.

Step relation. The graph-induced ternary relation $S_\delta \subseteq V \times E \times V$ is defined as

$$\begin{aligned}(v,e,z) \in S_\delta \iff \Big(& [\mathsf{orient} = \rightarrow \ \wedge\ \mathrm{dir}(e) = \rightarrow \ \wedge\ \mathrm{src}(e) = v \ \wedge\ \mathrm{dst}(e) = z] \ \vee \\ & [\mathsf{orient} = \leftarrow \ \wedge\ \mathrm{dir}(e) = \rightarrow \ \wedge\ \mathrm{src}(e) = z \ \wedge\ \mathrm{dst}(e) = v] \ \vee \\ & [\mathsf{orient} = \leftrightarrow \ \wedge\ \mathrm{dir}(e) = \leftrightarrow \ \wedge\ \{v,z\} = \{\mathrm{src}(e), \mathrm{dst}(e)\}] \Big) \\ \wedge\ \big(\Phi_e(e) = \top\big) & \wedge \big(\Phi_{\mathrm{src}}(v) = \top\big) \wedge \big(\Phi_{\mathrm{dst}}(z) = \top\big).\end{aligned}$$

Intuition: forward and backward steps match directed edges in the indicated orientation, while the undirected step $\leftrightarrow$ matches an undirected edge in either traversal direction. Edge-pattern variants that admit more than one orientation use the union of the corresponding cases.

Example (one library step). For the library instance, let

$$\delta_{\mathsf{inc}} = (\rightarrow, \Phi_{\mathsf{inc}}, \top, \top), \qquad \Phi_{\mathsf{inc}}(e) = \top \iff \texttt{INCLUDES} \in \lambda_E(e).$$

Since $e_{\mathsf{inc},1}$ is directed from l to b_1 and has this label,

$$(l, e_{\mathsf{inc},1}, b_1) \in S_{\delta_{\mathsf{inc}}}.$$

Label and predicate atoms as special cases. The abstract atoms ℓ, ℓ^{-1}, and $\{\,\eta\,\}$ can be seen as notational shorthands for specific step predicates:

- $\{\,\eta\,\}$ corresponds to the forward step predicate $\delta_\eta = (\rightarrow, \Phi_e, \top, \top)$ with $\Phi_e(e) = \top$ iff $\eta(e) = \top$.
- $\{\,\eta\,\}^{-1}$ corresponds to the backward step predicate $\delta_{\eta^{-1}} = (\leftarrow, \Phi_e, \top, \top)$ with the same edge test.
- ℓ corresponds to the forward label-step predicate $\delta_\ell = (\rightarrow, \Phi_e, \top, \top)$ with $\Phi_e(e) = \top$ iff $\ell \in \lambda_E(e)$.
- ℓ^{-1} corresponds to the backward label-step predicate $\delta_{\ell^{-1}} = (\leftarrow, \Phi_e, \top, \top)$ with the same label test.

We keep the explicit equations for these atoms below to make the endpoint orientation transparent; undirected steps are represented by the $\mathsf{orient} = \leftrightarrow$ case of S_δ.

Ternary denotation with bag semantics. For the fixed graph G and mode mode, the denotation of r is a (possibly infinite) bag of endpoint-labeled path witnesses:

$$[\![r]\!]_G^{\mathsf{mode}} \;\in\; \mathcal{M}(V \times \mathsf{Path}(G) \times V).$$

Its support consists of triples (u, w, v) such that $\mathrm{head}(w) = u$, $\mathrm{tail}(w) = v$, $\mathsf{ok}_{\mathsf{mode}}(w) = \top$, and w matches r by the rules below.

Important (bag meaning). The multiplicity $[\![r]\!]_G^{\mathsf{mode}}(u, w, v)$ counts the number of *derivations* (proof trees) of the judgment "w matches r from u to v" induced by the inductive clauses (alternation may add multiplicities; concatenation multiplies and sums over intermediate endpoints; repetition sums over iteration depth). Consequently, the same triple (u, w, v) may occur with multiplicity > 1 when the expression admits multiple decompositions producing the same witness w (e.g., due to explicit multiset duplication such as $r \cup_{\mathsf{ms}} r$, or due to different decompositions in iterated constructs). This derivation count is auxiliary bookkeeping. Ordinary alternation and the final graph-pattern result are set-valued; only distinctions introduced by explicit multiset alternation are intended to survive as distinct reduced matches.

Conformance note. GQL distinguishes the ordinary vertical-bar alternation, which has set semantics, from a multiset-sensitive alternation form. Accordingly, the unsubscripted abstract constructor $r_1 \cup r_2$ denotes set alternation; $\cup_{\mathsf{ms}}$ is used only for the explicit multiset-alternation form.

For a bag B, define its setification by

$$\mathrm{setify}(B)(x) := \begin{cases} 1 & \text{if } B(x) > 0, \\ 0 & \text{otherwise.} \end{cases}$$

If one needs *set* semantics ("distinct path witnesses") for the overall result of an RPE, it can be obtained either by setification setify or, equivalently, by taking the support $\mathrm{supp}([\![r]\!]_G^{\mathsf{mode}})$ as a post-processing step.

Base cases. (All bags below assign multiplicity 1 to each listed triple.)

$$
\begin{aligned}
[\![\varepsilon]\!]_G^{\mathsf{mode}} &= \{\, (v, \langle v\rangle, v) \mid v \in V \,\} \\
[\![[\nu]]\!]_G^{\mathsf{mode}} &= \{\, (v, \langle v\rangle, v) \mid \nu(v) = \top \,\} \\
[\![\ell]\!]_G^{\mathsf{mode}} &= \left\{ (u, \langle u, e, v\rangle, v) \,\middle|\, \begin{smallmatrix} e\in E,\, \ell\in\lambda_E(e),\, \mathrm{dir}(e)=\rightarrow, \\ \mathrm{src}(e)=u,\, \mathrm{dst}(e)=v, \\ \mathsf{ok}_{\mathsf{mode}}(\langle u,e,v\rangle)=\top \end{smallmatrix} \right\} \\
[\![\ell^{-1}]\!]_G^{\mathsf{mode}} &= \left\{ (v, \langle v, e, u\rangle, u) \,\middle|\, \begin{smallmatrix} e\in E,\, \ell\in\lambda_E(e),\, \mathrm{dir}(e)=\rightarrow, \\ \mathrm{src}(e)=u,\, \mathrm{dst}(e)=v, \\ \mathsf{ok}_{\mathsf{mode}}(\langle v,e,u\rangle)=\top \end{smallmatrix} \right\} \\
[\![\{\,\eta\,\}]\!]_G^{\mathsf{mode}} &= \left\{ (u, \langle u, e, v\rangle, v) \,\middle|\, \begin{smallmatrix} e\in E,\, \eta(e)=\top,\, \mathrm{dir}(e)=\rightarrow, \\ \mathrm{src}(e)=u,\, \mathrm{dst}(e)=v, \\ \mathsf{ok}_{\mathsf{mode}}(\langle u,e,v\rangle)=\top \end{smallmatrix} \right\} \\
[\![\{\,\eta\,\}^{-1}]\!]_G^{\mathsf{mode}} &= \left\{ (v, \langle v, e, u\rangle, u) \,\middle|\, \begin{smallmatrix} e\in E,\, \eta(e)=\top,\, \mathrm{dir}(e)=\rightarrow, \\ \mathrm{src}(e)=u,\, \mathrm{dst}(e)=v, \\ \mathsf{ok}_{\mathsf{mode}}(\langle v,e,u\rangle)=\top \end{smallmatrix} \right\}.
\end{aligned}
$$

Inductive cases. Here and in the sequel, $\biguplus_k B_k$ denotes the iterated bag union of bags B_k, obtained by pointwise addition of multiplicities.

We write bag-join on the middle endpoint with path concatenation as $\bowtie_{\bowtie}$, where $\bowtie$ combines two compatible triples by concatenating their path witnesses:

$$(u, w_1, x) \;\bowtie\; (x, w_2, v) \;=\; (u, w_1 \cdot w_2, v).$$

Formally, for bags $B_1, B_2 \in \mathcal{M}(V \times \mathsf{Path}(G) \times V)$, define

$$(B_1 \bowtie_{\bowtie} B_2)(u, w, v) \;:=\; \sum_{x\in V} \sum_{\substack{w_1,w_2 \\ w=w_1\cdot w_2}} B_1(u, w_1, x) \cdot B_2(x, w_2, v).$$

We also use the mode filter, which rejects witnesses not admissible under the chosen path mode:

$$\mathrm{filter}_{\mathsf{mode}}(B)(u, w, v) \;:=\; \begin{cases} B(u, w, v) & \text{if } \mathsf{ok}_{\mathsf{mode}}(w) = \top, \\ 0 & \text{otherwise,} \end{cases}$$

Alternation. We distinguish explicitly:

$$
\begin{aligned}
[\![r_1 \cup_{\mathsf{ms}} r_2]\!]_G^{\mathsf{mode}} &:= [\![r_1]\!]_G^{\mathsf{mode}} \uplus [\![r_2]\!]_G^{\mathsf{mode}} \qquad \text{(multiset alternation)}, \\
[\![r_1 \cup_{\mathsf{set}} r_2]\!]_G^{\mathsf{mode}} &:= \mathrm{setify}\Big([\![r_1]\!]_G^{\mathsf{mode}} \uplus [\![r_2]\!]_G^{\mathsf{mode}} \Big) \qquad \text{(set alternation)}.
\end{aligned}
$$

Unless stated otherwise, we interpret the abstract constructor $(r_1 \cup r_2)$ as $\cup_{\mathsf{set}}$, matching concrete `"|"`. Only the distinct multiset-alternation token is interpreted as $\cup_{\mathsf{ms}}$.

Concatenation and repetition. Concatenation composes the witnesses produced by its two operands. A witness of r_1 ending at some node x may be combined with a witness of r_2 starting at the same node x. The resulting witness is obtained by concatenating the two paths, while its multiplicity is the product of the multiplicities of the component witnesses. Contributions obtained through different intermediate nodes or different decompositions of the same path are then added by the bag join defined above.

Although each operand has already been evaluated under the selected path mode, their concatenation need not itself satisfy that mode. For example, two individually admissible path witnesses may, after concatenation, repeat an edge or a node forbidden by the selected mode. The mode filter is therefore applied again to the composed witnesses:

$$[\![r_1 \cdot r_2]\!]_G^{\mathsf{mode}} = \mathrm{filter}_{\mathsf{mode}}\Big([\![r_1]\!]_G^{\mathsf{mode}} \bowtie_{\bowtie} [\![r_2]\!]_G^{\mathsf{mode}}\Big).$$

To define repetition, we first define the k-fold concatenation of an expression r. The zeroth power denotes the empty path at each node and is therefore identified with the semantics of ε:

$$[\![r]\!]_G^{\mathsf{mode},0} = [\![\varepsilon]\!]_G^{\mathsf{mode}}.$$

For $k \geq 0$, the $(k+1)$-st power is obtained by concatenating one additional witness of r with a witness of its k-th power. As in ordinary concatenation, the resulting path is filtered according to the selected path mode:

$$[\![r]\!]_G^{\mathsf{mode},k+1} = \mathrm{filter}_{\mathsf{mode}}\Big([\![r]\!]_G^{\mathsf{mode}} \bowtie_{\bowtie} [\![r]\!]_G^{\mathsf{mode},k}\Big).$$

Consequently, $[\![r]\!]_G^{\mathsf{mode},k}$ contains the path witnesses formed from exactly k successive occurrences of r, with their bag multiplicities determined by all admissible decompositions into k component witnesses.

For a well-formed concrete GQL quantified primary, every occurrence of r consumes at least one edge. Hence a witness of length L can arise only from repetition counts $k \leq L$, and its multiplicity cannot become infinite by summing over repetition depths. Expressions that violate the positive minimum-path-length condition, such as ε^*, are excluded from the concrete language.

The Kleene-star expression collects the witnesses obtained for every finite number of repetitions:

$$[\![r^*]\!]_G^{\mathsf{mode}} = \biguplus_{k=0}^{\infty} [\![r]\!]_G^{\mathsf{mode},k}.$$

Thus, the result includes the empty-path witnesses contributed by $k = 0$, as well as witnesses formed by one or more occurrences of r. Because the outer operation is bag union, the multiplicity of a witness records the total number of derivations of that witness across all repetition counts and all admissible decompositions.

Finally, for bounds $m \in \mathbb{N}$ and $n \in \mathbb{N} \cup \{\infty\}$ with $m \le n$, bounded repetition retains only the powers whose repetition counts lie in the specified interval:

$$[\![r^{m..n}]\!]_G^{\mathsf{mode}} = \biguplus_{k=m}^{n} [\![r]\!]_G^{\mathsf{mode},k}, \qquad (n = \infty \text{ allowed}).$$

When $n = \infty$, the expression denotes lower-bounded repetition; when $m = n$, it denotes exactly m repetitions. In particular, $r^{0..\infty}$ coincides with r^*.

Reversal of expressions. Define the *reversal* $r^\top$ recursively by

$$\begin{aligned}
\varepsilon^\top &= \varepsilon, \\
[\nu]^\top &= [\nu], \\
\{\,\eta\,\}^\top &= \{\,\eta\,\}^{-1}, \\
(\{\,\eta\,\}^{-1})^\top &= \{\,\eta\,\}, \\
\ell^\top &= \ell^{-1}, \\
(\ell^{-1})^\top &= \ell, \\
(r \cdot s)^\top &= s^\top \cdot r^\top, \\
(r \cup s)^\top &= r^\top \cup s^\top, \\
(r \cup_{\mathsf{ms}} s)^\top &= r^\top \cup_{\mathsf{ms}} s^\top, \\
(r^*)^\top &= (r^\top)^*, \\
(r^{m..n})^\top &= (r^\top)^{m..n}.
\end{aligned}$$

This witnesses symmetry of undirected matching and is used in bidirectional expansions.

Guards over intermediates. Let ψ be a Boolean predicate on paths, with type

$$\psi : \mathsf{Path}(G) \to \{\top, \bot\}.$$

It is built from node/edge predicates with quantifiers $\forall, \exists$ over position indices. For $B \in \mathcal{M}(V \times \mathsf{Path}(G) \times V)$, define

$$\mathrm{filter}_\psi(B)(u, w, v) := \begin{cases} B(u, w, v) & \text{if } \psi(w) = \top, \\ 0 & \text{otherwise.} \end{cases}$$

and define a *guarded* expression $r \;\&\; \psi$ by

$$[\![r \;\&\; \psi]\!]_G^{\mathsf{mode}} \;=\; \mathrm{filter}_\psi\Big([\![r]\!]_G^{\mathsf{mode}}\Big).$$

Intent: $r \;\&\; \psi$ is a semantic shorthand for a parenthesized path pattern with an attached `WHERE` clause that uses only the component's local variables.

11.2 Shortest-Path Semantics

Candidates and endpoint partitions. Fix a graph G and a path mode pm such that

$$\mathsf{pm} \in \{\mathsf{WALK}, \mathsf{TRAIL}, \mathsf{SIMPLE}, \mathsf{ACYCLIC}\}.$$

For a regular path expression R, let $\mathsf{Cand}(R, \mathsf{pm})$ be the set of candidate path matches admitted by the auxiliary denotation and the chosen path mode. More precisely, let $\mathsf{Der}(R, \mathsf{pm})$ be the set of successful derivation trees. For $d \in \mathsf{Der}(R, \mathsf{pm})$, write (u_d, w_d, v_d) for its endpoint/path triple and $\chi_{\mathrm{sp}}(d)$ for its trace of subpath markers introduced by explicit multiset alternation. Define

$$\begin{aligned} d \equiv_{\mathrm{sp}} d' \quad &\Longleftrightarrow \quad (u_d, w_d, v_d) = (u_{d'}, w_{d'}, v_{d'}) \\ &\qquad \wedge \;\; \chi_{\mathrm{sp}}(d) = \chi_{\mathrm{sp}}(d'), \\ \mathsf{Cand}(R, \mathsf{pm}) &:= \mathsf{Der}(R, \mathsf{pm}) / \equiv_{\mathrm{sp}} . \end{aligned}$$

For a candidate class $c = [d]_{\equiv_{\mathrm{sp}}}$, define

$$\mathrm{path}(c) := w_d.$$

This is well-defined because equivalent derivations have the same endpoint/path triple. Thus ordinary derivational duplicates are identified, while branch distinctions introduced by explicit multiset alternation remain distinct, as represented in the standard by subpath markers. For $u, v \in V$, write the endpoint partition

$$\Pi_{u,v}(\mathsf{Cand}(R, \mathsf{pm})) := \{c \in \mathsf{Cand}(R, \mathsf{pm}) \mid \mathrm{head}(\mathrm{path}(c)) = u \;\wedge\; \mathrm{tail}(\mathrm{path}(c)) = v\}.$$

Below, $\Pi_{u,v}$ abbreviates this partition for the fixed R and pm. Every path-search selector acts endpoint-locally on these sets and returns the set union of its per-partition results.

Example (one endpoint partition).

For the library instance, take $R_{\mathsf{inc}} = \texttt{INCLUDES}$ and $\mathsf{pm} = \mathsf{WALK}$, and let

$$w_{\mathsf{inc},1} = \langle l, e_{\mathsf{inc},1}, b_1 \rangle.$$

If $c_{\mathsf{inc},1}$ is the candidate represented by the successful derivation whose endpoint/path triple is $(l, w_{\mathsf{inc},1}, b_1)$, then

$$c_{\mathsf{inc},1} \in \Pi_{l,b_1}\Big(\mathsf{Cand}(R_{\mathsf{inc}}, \mathsf{WALK})\Big).$$

Length and groups. Let $\mathrm{len}(w)$ denote the number of edge-steps in w. For a fixed partition $\Pi_{u,v}$, define its length-groups by

$$\mathcal{G}_{u,v}(k) := \{c \in \Pi_{u,v} \mid \mathrm{len}(\mathrm{path}(c)) = k\}, \qquad (k \in \mathbb{N}).$$

Groups are ordered by the key k in increasing order. We write

$$\mathrm{minlen}(u, v) = \min\{\, \mathrm{len}(\mathrm{path}(c)) \mid c \in \Pi_{u,v} \,\}$$

when $\Pi_{u,v} \neq \varnothing$.

Selective path-search operators. A path-search prefix in this abstraction is one of:

$$\begin{aligned}&\mathsf{ALL\ PATHS},\\ &\mathsf{ANY}_N\ \mathsf{PATHS},\\ &\mathsf{SHORTEST}_N\ \mathsf{PATHS},\\ &\mathsf{SHORTEST}_N\ \mathsf{GROUPS},\end{aligned}$$

where $N \in \mathbb{Z}_{>0}$. Except for $\mathsf{ALL\ PATHS}$, these are selective prefixes. Their denotation is a partition-wise selection operator $\mathsf{SPSP}_s(\cdot)$ over $\mathsf{Cand}(R, \mathsf{pm})$, defined as follows.

All paths (non-selective).

$$\mathsf{SPSP}_{\mathsf{ALL\ PATHS}}(\mathsf{Cand}(R, \mathsf{pm})) \;=\; \mathsf{Cand}(R, \mathsf{pm}).$$

Any N paths per endpoints. For each (u, v), if $|\Pi_{u,v}| \leq N$, keep $\Pi_{u,v}$; otherwise choose an implementation-dependent subset $\mathrm{pick}_N(\Pi_{u,v}) \subseteq \Pi_{u,v}$ of cardinality N.

$$\mathsf{SPSP}_{\mathsf{ANY}_N\ \mathsf{PATHS}}(\mathsf{Cand}(R, \mathsf{pm})) \;=\; \bigcup_{u,v} \begin{cases} \Pi_{u,v} & \text{if } |\Pi_{u,v}| \leq N, \\ \mathrm{pick}_N(\Pi_{u,v}) & \text{otherwise.} \end{cases}$$

Counted shortest N paths per endpoints. Within each $\Pi_{u,v}$, order candidates by nondecreasing path length; ties may be resolved in an implementation-dependent manner. Return up to N candidates from the front of this order.

$$\mathsf{SPSP}_{\mathsf{SHORTEST}_N\ \mathsf{PATHS}}(\mathsf{Cand}(R, \mathsf{pm})) \;=\; \bigcup_{u,v} \mathrm{takeShortest}_N(\Pi_{u,v}),$$

where $\text{takeShortest}_N(\Pi_{u,v}) \subseteq \Pi_{u,v}$ and $|\text{takeShortest}_N(\Pi_{u,v})| = \min\{N, |\Pi_{u,v}|\}$.

Counted shortest N groups per endpoints. Within each $\Pi_{u,v}$, take the groups $\mathcal{G}_{u,v}(k)$ in increasing k. Return the set union of the first N nonempty groups (or all of them if fewer exist):

$$\mathsf{SPSP}_{\mathsf{SHORTEST}\ N\ \mathsf{GROUPS}}(\mathsf{Cand}(R, \mathsf{pm})) \;=\; \bigcup_{u,v} \bigcup_{\substack{k \text{ among the} \\ \text{first } N \text{ nonempty groups}}} \mathcal{G}_{u,v}(k).$$

Named shorthands and defaults. The following equivalences and defaults hold:

$$\begin{aligned} \mathsf{ANY\ SHORTEST\ PATH} &\equiv \mathsf{SHORTEST}_1\ \mathsf{PATHS}, \\ \mathsf{ALL\ SHORTEST\ PATHS} &\equiv \mathsf{SHORTEST}_1\ \mathsf{GROUPS}, \end{aligned}$$

if N is omitted then $N := 1$, if pm is omitted then $\mathsf{pm} := \mathsf{WALK}$.

Well-formedness and finiteness. If a selective prefix declares $N \leq 0$ it is an error. Every quantified or questioned path-pattern primary has positive minimum path length. For each unbounded quantifier, a restrictive mode TRAIL, ACYCLIC, or SIMPLE, a selective prefix, or the graph pattern's `DIFFERENT EDGES` match mode is required. A restrictive mode or `DIFFERENT EDGES` makes the admissible match set finite; a selective prefix may start from an infinite candidate set but returns a finite selected set.

Chapter 11 — Takeaways

- **Purpose:** Established the syntax and semantics of regular path expressions as the primary declarative mechanism for path matching.
- **Introduced:** Regular path-expression constructs for alternation, concatenation, and repetition; forward and inverse navigation steps; a shortest-path evaluation mode with explicit selection behavior.
- **Key distinctions:** General path enumeration versus shortest-path selection; forward navigation versus inverse navigation along relationships.
- **Consequences:** Made expressive path queries portable and comparable across implementations; clarified how explicit multiset alternation preserves distinct matches; explained where semantics deliberately restricted outputs to control explosion and improve predictability.
- **Bridge:** Connected path results back to the pipeline model by enabling grouping and aggregation over bindings that included paths.

Part V

Aggregation and Ordering

12 Grouping and Aggregation

This chapter explains how GQL transforms bags of bindings into grouped summaries using the `GROUP BY` clause and aggregate functions. We formalize grouping as a partitioning of binding tables by key expressions, specify the behavior of standard and collection-valued aggregates (including their treatment of null and empty inputs), and describe the evaluation pipeline that combines `WHERE`, `GROUP BY`, `HAVING`, `DISTINCT`, and ordering into a single coherent semantics.

12.1 `GROUP BY` and Aggregates

A concrete use of `GROUP BY` and `SUM` appears in Section 4.4.

Grouping function. Let $T \in \mathsf{BT}$ be an input binding table with $\mathrm{bag}(T) = \langle \tau, B \rangle$, and $K = \langle k_1, \ldots, k_s \rangle$ be an ordered sequence of grouping keys (names or expressions). We define the key extractor

$$\kappa_K(r) \;=\; \langle [\![k_1]\!](r), \ldots, [\![k_s]\!](r) \rangle.$$

Write $u \approx_{\mathrm{nd}} v$ when u and v are not distinct under GQL's recursive distinctness rules. The grouping induces a finite index set I (one index per distinctness class of κ_K-values) and a family of pairwise disjoint sub-bags $(B_i)_{i \in I}$ with $B_i \sqsubseteq_{\mathsf{bag}} B$ such that

$$B \;=\; \biguplus_{i \in I} B_i \quad \text{and} \quad \forall i \in I \, \forall r, s \in B_i : \; \kappa_K(r) \approx_{\mathrm{nd}} \kappa_K(s).$$

Here $B_i \sqsubseteq_{\mathsf{bag}} B$ means $B_i(r) \leq B(r)$ for every record r. If distinctness cannot be determined because two values are not comparable, grouping raises data exception 22G04. For the empty grouping sequence $K = \langle\rangle$, take $I = \{*\}$ and $B_* = B$, including when B is empty. Thus global aggregation has one group and, on empty input, evaluates aggregates over the empty multiset.

Aggregators. An *aggregator* is a partial function

$$F : \mathcal{M}_{\mathrm{fin}}(\mathbb{V}) \rightharpoonup \mathbb{V} \cup \{\mathsf{null}\}$$

equipped with a null-handling policy and an input-type requirement appropriate to the particular aggregate operation. For a value expression E, the notation $F(E)$ denotes the application of F to the multiset

$$\{\{ [\![E]\!](r) \mid r \in B_i \}\}$$

obtained by evaluating E for every record of a group B_i.

Except for COUNT(∗), aggregate functions eliminate null inputs before computing their result. If no set quantifier is specified, ALL is implicit, so duplicate values and their multiplicities are preserved. With DISTINCT, duplicate non-null values are eliminated before the aggregate operation is applied.

The principal aggregators and their default empty-input behavior are as follows:

- COUNT(E): returns the number of records in the group for which $[\![E]\!](r) \neq$ null. Under ALL, repeated values are counted repeatedly; under DISTINCT, each distinct non-null value is counted once. If no non-null value remains, the result is 0.
- COUNT* ≡ COUNT(∗): returns the number of records in the group, independently of the values contained in those records. In particular, records containing only null values are still counted. On an empty group, the result is 0.
- SUM(E): returns the arithmetic sum of the non-null numeric values of E, with multiplicities preserved under ALL. If no non-null value remains, the default result is null.
- AVG(E): if at least one non-null numeric value of E remains, returns their arithmetic mean, that is, their sum divided by their number. If no non-null value remains, the result is null.
- MIN(E): returns the least non-null value of E, according to the applicable GQL ordering rules. If no non-null value remains, the result is null.
- MAX(E): returns the greatest non-null value of E, according to the applicable GQL ordering rules. If no non-null value remains, the result is null.
- PRODUCT(E), if supported: returns the arithmetic product of the non-null numeric values of E, with multiplicities preserved under ALL. If no non-null value remains, the default result is null.
- COLLECT_LIST(E), when optional Feature GF10 is supported: returns a list containing the non-null values of E. Under ALL, multiplicities are preserved; under DISTINCT, each distinct value occurs once. The list follows the order induced by the input group when such an order is defined; no application-independent order is assumed for an unordered group. If no non-null value remains, the result is the empty list.

When the corresponding optional feature is supported, GQL permits an explicit empty-aggregate behavior for SUM and PRODUCT. For example,

$$\text{SUM}(E, 0 \text{ ON EMPTY}) \text{ returns } 0, \text{ and } \text{PRODUCT}(E, 1 \text{ ON EMPTY}) \text{ returns } 1,$$

when no non-null input value remains.

Empty-aggregate behavior. By default, aggregates over an empty input multiset yield null, except that COUNT(∗) and COUNT(E) yield 0, and COLLECT_LIST yields the empty list. When the optional feature F443 (empty aggregate behavior) is supported, the aggregate functions SUM and PRODUCT admit an `ON EMPTY` clause:

$$\text{SUM}(E, 0 \texttt{ ON EMPTY}) \quad \text{and} \quad \text{PRODUCT}(E, 1 \texttt{ ON EMPTY}),$$

which override the default null result on empty input (as specified in GQL). Using `NULL ON EMPTY` is permitted but semantically redundant with the default behavior.

Grouped extend. For expressions E_j used inside aggregators and a per-group bag projection

$$\pi^*_{E_j}(B_i) \;=\; \{\{ [\![E_j]\!](r) \mid r \in B_i \}\} \quad (i \in I),$$

(a multiset, possibly containing null values), let r_i be the record consisting of the key fields determined by partition i and the aggregated fields $n_j := F_j(\pi^*_{E_j}(B_i))$. Let τ_K be the record type contributed by the grouping keys and τ_j the result type of the j-th aggregate. The grouped result of applying `GROUP BY` with aggregation is

$$\text{bag}(\text{GROUP BY } K \text{ AGG } \{F_j(E_j) \mapsto n_j\}_j(T)) \;=\; \Big\langle\; \tau_K \cup \{n_j \mapsto \tau_j\}_j,\;\; \biguplus_{i \in I} \{\, r_i \,\} \;\Big\rangle,$$

with the complete output descriptor carrying the ordering and preferred-column metadata prescribed for the result statement.

`HAVING` and post-aggregation filtering. In a `SELECT` statement, the `HAVING` clause filters groups after grouping and aggregate evaluation. In contrast to `WHERE`, which filters individual input records before grouping, `HAVING` retains only those grouped result records for which its search condition evaluates to $\top$.

Formally, a `HAVING` clause is modeled as a selection σ_φ applied *after* grouping, where the predicate φ may refer to grouping keys and aggregate outputs. If no `HAVING` clause is present, the corresponding selection is the identity, equivalently with predicate $\varphi \equiv \top$.

Concrete syntax for `GROUP BY` and `HAVING`. At the concrete-syntax level, grouping and post-aggregation filtering are available in a `SELECT` statement as an optional `GROUP BY`

clause followed by an optional `HAVING` clause. A `RETURN` statement may contain `GROUP BY`, but not `HAVING`. The following simplified BNF summarizes the relevant fragments, leaving the surrounding statement productions to the GQL standard:

```
group-by-clause ::= "GROUP" "BY"
                    grouping-element-list

grouping-element-list ::= grouping-element
                          { "," grouping-element }
                        | "()"  /* empty grouping set */

grouping-element ::= binding-variable-reference

having-clause ::= "HAVING" search-condition

search-condition ::= ...  /* boolean expression */
```

Intuitively, each `binding-variable-reference` identifies one grouping key, while `"()"` denotes the empty grouping-element list (global aggregation). The `having-clause` attaches a `search-condition` that is evaluated per group over the grouping keys and aggregate outputs, matching the post-aggregation filtering semantics described above. The nonterminal `search-condition` ranges over boolean expressions in the three-valued logic.

Example (aggregate results and `HAVING`). For the two `INCLUDES` edges associated with Alice in the library instance, the multiset of late fees is

$$M_{\mathsf{Alice}} = \{\{0.0, 2.5\}\}.$$

Consequently,

$$\mathrm{COUNT}(*) = 2, \qquad \mathrm{AVG}(\texttt{i.late_fee}) = 1.25,$$
$$\mathrm{MIN}(\texttt{i.late_fee}) = 0.0, \qquad \mathrm{MAX}(\texttt{i.late_fee}) = 2.5.$$

When optional Feature GF10 is supported, one possible value is

$$\mathrm{COLLECT_LIST}(\texttt{i.late_fee}) = [0.0, 2.5],$$

but the reverse order is also permitted because the input group is unordered. In a `SELECT` statement, the group is retained by the condition

```
HAVING AVG(i.late_fee) > 1.0,
```

because its average is 1.25.

12.2 HAVING and DISTINCT

Grouped projection and pagination. For a bag B, we write $\gamma_{\vec{g};\vec{h}}(B)$ for the combined grouping and aggregate-projection operator. It partitions the records of B according to the grouping expressions $\vec{g}$ and produces one output record for each group by evaluating the expressions $\vec{h}$, which may include grouping expressions and aggregate expressions, over that group. This is a meta-level algebraic operator, not concrete GQL syntax.

For a sequence $S = \langle s_1, \ldots, s_n \rangle$, let $\mathrm{OFFSET}_o(S)$ denote the sequence obtained by discarding its first o elements, or the empty sequence if $o \geq n$. Similarly, $\mathrm{LIMIT}_\ell(S)$ denotes the sequence containing at most the first ℓ elements of S. When the corresponding clause is absent, the respective operation is the identity. For an unordered bag B, let $\mathrm{Enum}_{\mathrm{impl}}(B)$ denote an implementation-dependent enumeration of its record occurrences.

Let φ and θ be the `WHERE` and `HAVING` predicates, $\vec{g}$ and $\vec{h}$ the grouping and output-expression vectors, $\vec{o}$ the sort-specification list, and o, ℓ non-negative integer offset and limit values, respectively. Given an input bag B_0 (from a pattern match or subquery), we define the following canonical `SELECT` evaluation pipeline:

$$B_1 = \mathrm{filter}_{\varphi}(B_0)$$

(three-valued filter: keep only $\top$)

$$B_2 = \begin{cases} \gamma_{\vec{g};\vec{h}}(B_1) & \text{if a } \texttt{GROUP BY} \text{ or implicit global grouping is present} \\ B_1 & \text{otherwise} \end{cases}$$

$$B_3 = \begin{cases} \sigma_{\theta}(B_2) & \text{if a } \texttt{HAVING} \text{ clause is present} \\ B_2 & \text{otherwise} \end{cases}$$

$$B_4 = \begin{cases} \mathrm{DISTINCT}(B_3) & \text{if } \texttt{DISTINCT} \text{ is requested} \\ B_3 & \text{otherwise} \end{cases}$$

$$S_0 = \begin{cases} \mathrm{ORDER\ BY}_{\vec{o}}(B_4) & \text{if an } \texttt{ORDER BY} \text{ clause is present} \\ \mathrm{Enum}_{\mathrm{impl}}(B_4) & \text{otherwise} \end{cases}$$

$$S_1 = \begin{cases} \mathrm{OFFSET}_o(S_0) & \text{if an } \texttt{OFFSET} \text{ clause is present} \\ S_0 & \text{otherwise} \end{cases}$$

$$S' = \begin{cases} \mathrm{LIMIT}_\ell(S_1) & \text{if a } \texttt{LIMIT} \text{ clause is present} \\ S_1 & \text{otherwise.} \end{cases}$$

Consequently, the particular records retained by `OFFSET` or `LIMIT` are not determined unless an explicit `ORDER BY` clause establishes the relevant order.

The final result is S', represented here as a sequence. In the absence of a `WHERE` clause, we take $\varphi \equiv \top$, so that $B_1 = B_0$. In the common case of aggregates without an explicit `GROUP BY`, $\vec{g}$ is empty and $\vec{h}$ contains aggregate expressions. All input records then form a single global group, which may contain no records when B_1 is empty.

If an update or side-effecting clause is present, it consumes B_1 or a later stage explicitly specified by the statement and executes according to the transactional semantics of Chapter 19. Ordering affects the selection performed by `OFFSET` and `LIMIT`; otherwise, it affects only the presentation of the query result unless the semantics of a particular statement explicitly depend on order.

12.3 Primitive Result Statements

Position in the language. In a linear query or data-modifying statement, the tail of the statement is a *primitive result statement* that turns the working binding table produced by the preceding clauses into the final result sequence. Concretely, this is the part of the statement that contains the `RETURN` clause, an optional set quantifier and grouping, and optional ordering and pagination, corresponding to the *primitive result statement*, *return statement*, and *order by and page statement* of the GQL standard. Concrete primitive result statements, including aggregation and pagination, are shown in Section 4.4.

Link to the canonical pipeline. The primitive result pipeline is the `RETURN` portion of the account in Section 12.2: projection and optional `GROUP BY`, followed by `DISTINCT` when requested, and then `ORDER BY`, `OFFSET`, and `LIMIT`. Unlike `SELECT`, a `RETURN` statement has no `HAVING` clause. For reference, we collect here a condensed, *informative* BNF for the concrete syntax that drives this pipeline. The normative concrete grammar remains that of ISO/IEC 39075.

Simplified BNF. Using the bracket conventions of this book (`[ ... ]` for optionality and `{ ... }` for repetition), we write:

```
primitive-result-stmt ::= return-stmt [ order-by-and-page-stmt ]
                        | "FINISH"

return-stmt ::= "RETURN" return-body

return-body ::= [ set-quant ]
                ( "*" | return-item-list )
                [ group-by-clause ]

return-item-list ::= return-item { "," return-item }

```

```
return-item ::= expr [ "AS" alias ]

set-quant   ::= "DISTINCT" | "ALL"

expr   ::= ...   /* value or aggregating expression */
alias  ::= identifier
```

This summary follows the productions for *primitive result statement*, *return statement*, and *return statement body* in the GQL standard. All alternatives and non-terminals used here are either directly taken from that grammar, or are compact names for groups of its productions. The main points of note are:

- If the `set-quant` is omitted, the default is `ALL`, that is, multiset semantics apply and duplicate rows in the result are preserved.
- The `"*"` alternative abbreviates "all visible bindings" in the current working table, that is, one projected column for each visible name, as prescribed by the standard and by our projection operators in Chapter 9.
- The non-terminal `group-by-clause` stands for an optional `GROUP BY` with its grouping keys. `HAVING` is available in a `SELECT` statement, not in this `RETURN` production.
- The non-terminal `order-by-and-page-stmt` is expanded in Chapter 13; it permits `ORDER BY` with optional pagination, `OFFSET` with optional `LIMIT`, or `LIMIT` alone. Its denotation is the composition of ordering and slicing operators described there.
- The metavariable `expr` ranges over value and aggregating expressions as defined in Part III (including aggregate function calls and scalar expressions); in the standard, this corresponds to the use of aggregating value expressions in the return statement body.
- The alternative `"FINISH"` reflects the optional `FINISH` statement in the standard; it is present only when the corresponding conformance feature is supported. In linear data-modifying statements, it is used for "update-only" programs whose observable effect is confined to changes of the graph state.
- The form `RETURN NO BINDINGS` is an internal specification device rather than syntax available to GQL users. It is equivalent to `RETURN ALL` with an empty return-item list and without a `GROUP BY` clause. Consequently, it produces one unit record $\langle\rangle$ for each input record, preserving input multiplicities while exporting no binding columns. The standard uses this form in syntactic transformations, notably when an `OPTIONAL MATCH` supplies no new output bindings. Since the simplified grammar above describes user-visible syntax, this internal form is intentionally omitted from its productions.

Chapter 12 — Takeaways

- **Purpose:** Defined how GQL transforms bags of bindings into grouped summaries using grouping keys and aggregate functions.
- **Introduced:** Grouping as partitioning of bindings by key; aggregate evaluation over each group; post-group filtering via `HAVING`; duplicate control via `DISTINCT` at the appropriate scope.
- **Key distinctions:** Pre-aggregation filtering versus post-aggregation filtering; bag-preserving aggregation versus duplicate elimination in aggregate inputs.
- **Consequences:** Fixed how unknown values and empty groups affected aggregates; made grouped outputs compositional within the binding-table algebra; enabled predictable analytic patterns on graph-shaped data.
- **Bridge:** Prepared the final result-shaping layer: ordering and pagination of (possibly aggregated) results.

13 Ordering and Pagination

This chapter formalizes how GQL turns unordered bags of bindings into ordered result sequences and how it supports pagination over such sequences. We treat `ORDER BY` as an ordering operation parameterized by direction, nulls policy, and collation, and `LIMIT`/`OFFSET` as slicing operators that produce ordered binding tables.

Concrete syntax for ordering and pagination. In linear query and data-modifying statements, ordering and paging are collected in the *order by and page statement* that follows the primitive result body (cf. Chapter 12). The following simplified BNF summarizes the relevant concrete syntax; it is aligned with the corresponding productions in ISO/IEC 39075 but omits catalog- and feature-related details:

```
order-by-and-page-stmt ::=
    order-by-clause
    [ offset-clause ]
    [ limit-clause ]
  | offset-clause [ limit-clause ]
  | limit-clause
order-by-clause ::= "ORDER" "BY" sort-spec-list
sort-spec-list ::= sort-spec { "," sort-spec }
sort-spec      ::= value-expr [ "ASC" | "DESC" ]
                   [ null-ordering ]
null-ordering  ::= "NULLS" ( "FIRST" | "LAST" )
offset-clause  ::= ( "OFFSET" | "SKIP" ) non-negative-int
limit-clause   ::= "LIMIT" [ "EXACT" | "APPROX" | "APPROXIMATE" ]
                   non-negative-int
value-expr ::= ...
```

When present, `ORDER BY` orders on one or more `value-expr` sort keys (each with optional direction and nulls policy). It may be followed by `OFFSET` (or its standard synonym `SKIP`) and `LIMIT`; an `OFFSET` with optional `LIMIT`, or a `LIMIT` alone, is also permitted without `ORDER BY`. The semantics of these operators, including peer ordering and approximation options, are given in the sections below.

13.1 `ORDER BY` and Ordered Tables

Sort specification list and defaults. An `ORDER BY` clause applies a *sort specification list* to the current working table. Each sort specification consists of a sort key (an aggregating

value expression), an optional ordering direction (`ASC`/`DESC`), and an optional nulls policy (`NULLS FIRST`/`NULLS LAST`). If the direction is omitted, ascending order is implied. If the nulls policy is omitted, an implementation-defined default is implied that must not depend on context outside the sort list. The result is an *ordered* binding table. Ordering by a due date and by an aggregate result is illustrated in Section 4.4.

Comparison semantics, nulls, and peers. Let P, Q be records and $K_1, \ldots, K_n$ the sort keys. The relative position of P and Q is determined lexicographically by comparing $K_i(P)$ and $K_i(Q)$ using $<$ for ascending or $>$ for descending, with special treatment of null per the chosen nulls policy. If both compared values are null, they are considered equal. If one side is null, the comparison outcome is fixed by `NULLS FIRST`/`NULLS LAST`. If neither side is null and the comparison yields U, the relative ordering is implementation-dependent. Records that are not distinct with respect to all sort specifications are *peers*; the ordering among peers is implementation-dependent.

Collation for character strings. Ordering of character strings uses a determined collation. The implementation defines exactly one default collation (UCS_BASIC, UNICODE per UTS#10 (The Unicode Consortium 2025), or an implementation-defined custom collation). This collation is used when comparing strings in ordering operations.

Type comparability of sort keys. Without the optional feature *Universal comparison* (GA04), the declared types of operands used in ordering operations (including sort keys) shall be comparable value types; otherwise, the program is non-conforming.

13.2 Determinism and Peer Ordering

An `ORDER BY` yields an ordered binding table, but the ordering of peers and of comparisons that evaluate to U is implementation-dependent. Full cross-implementation determinism requires (i) sort-key tuples that distinguish all output records and yield known comparisons, (ii) the same null ordering, and (iii) the same determined collation. An explicit nulls policy is the portable way to establish (ii), and relying on different implementations' default collations does not establish cross-implementation determinism. (Without `ORDER BY`, only bag semantics apply, see below.)

13.3 `LIMIT`/`OFFSET` (Slice)

`LIMIT` clause (`EXACT` vs. `APPROXIMATE`). `LIMIT` retains only a limited number of records from the current working table and produces an *ordered* binding table. If the input is unordered, the system first imposes an implementation-dependent order and then limits the records. A concrete `ORDER BY`–`LIMIT` composition appears in Section 4.4. `LIMIT` may be `EXACT` or `APPROX`/`APPROXIMATE` (optional feature GQ28). If the approximation

keyword is omitted, then EXACT is implicit, except in the special case where a linear query statement that directly contains the LIMIT clause also directly contains a value expression whose declared type is a vector type; in that case it is implementation-defined whether EXACT or an approximation synonym is implicit. Let V_{lim} be the non-negative integer value computed for the LIMIT clause. For EXACT, the first V_{lim} records are returned. For APPROXIMATE, the implementation first chooses an actual result cardinality $V'_{\text{lim}} \leq V_{\text{lim}}$, and then selects V'_{lim} records from the ordered input in an implementation-dependent manner. The selected records retain their relative input order, but they need not form a prefix of the input; in particular, they need not be the first V'_{lim} records.

OFFSET clause. Let V_{off} be the non-negative integer value computed for the OFFSET clause. It discards the first V_{off} records and returns the remainder as an *ordered* binding table. If the input is unordered, an implementation-dependent order is imposed prior to discarding. The clause requires optional Feature GQ12.

Example (OFFSET on a determined order).

```
USE libraryGraph
MATCH (l:Loan)-[i:INCLUDES]->(b:Book)
RETURN b.title AS title, i.due_date AS due_date
ORDER BY due_date
OFFSET 1
```

The ordered input contains Graph DB 101 first and Graph Theory second, so the result contains only the latter record.

Composition and evaluation order. When ORDER BY, OFFSET, and LIMIT appear together, evaluation proceeds *in this order*:

$$\texttt{ORDER BY} \rightarrow \texttt{OFFSET} \rightarrow \texttt{LIMIT}.$$

This composition is specified by the *order by and page statement*, which applies the sort first, then discards the initial V_{off} rows, and finally retains at most V_{lim} rows, where V_{off} and V_{lim} are the values specified by the OFFSET and LIMIT clauses, respectively.

Interaction with result formation. The ORDER BY and page statement is evaluated after its associated RETURN body. The standard's syntax transformations can introduce hidden aliases needed to evaluate admissible sort keys and remove them from the final output. Without Feature GA07, *Ordering by discarded binding variables*, a sort key cannot directly reference a binding variable discarded by that projection, except through the alias cases admitted by the applicable RETURN or SELECT syntax rules.

13.4 Determinism Guarantees

Totality of `GROUP BY`. For any finite bag B and finite expression vectors $\vec{g}, \vec{h}$ whose aggregators are defined on each group bag B_i induced by $\vec{g}$ (with expected null-handling),

$$\text{GROUP BY } \vec{g} \text{ YIELD } \vec{h}\ (B)$$

is a finite bag.

Optionals do not invent non-null data. Let A and B be binding tables, and consider the left outer join $A \triangleleft_\theta B$ defined in Section 9.2. For a binding table T with record type τ_T, write $\mathrm{hdr}(T) = \mathrm{dom}(\tau_T)$. For any output record, every attribute whose name occurs only in the schema of B (is in $\mathrm{hdr}(B) \setminus \mathrm{hdr}(A)$) is either material (copied from a matching record in B, if such a match exists) or null (if no match exists). No other values for these attributes are introduced.

Determinism of `ORDER BY`. Given the applicable fixed weak orders $\preceq$ from Section 8.3, their strict parts $\prec$, a collation, and a null policy, and a sort-specification list $\vec{o}$, $\text{ORDER BY}_{\vec{o}}(B)$ determines the order between non-peer records whose key comparisons are known. The relative order of peers, and of non-null key values whose comparison is U, remains implementation-dependent; full determinism therefore additionally requires sort keys that distinguish every output record and yield known comparisons. Without `ORDER BY`, only bag semantics are defined.

Chapter 13 — Takeaways

- **Purpose:** Formalized how unordered query results were converted into ordered sequences and how pagination operated over those sequences.
- **Introduced:** Ordering driven by sort keys with explicit null placement and collation; tie behavior and peer ordering; pagination as slicing with offset and limit.
- **Key distinctions:** Ordered results versus unordered bag results; deterministic versus implementation-dependent pagination when peers or unknown comparisons remain.
- **Consequences:** Explained when users could rely on repeatable page boundaries; clarified the semantics of null placement in ordering; integrated ordering and pagination into the overall clause pipeline.
- **Bridge:** Transitioned from read-only semantics to data modification and graph updates.

Part VI

Data Modification

14 Data Modification

This chapter formalizes how GQL data-modification statements transform the underlying property-graph store, treating updates as structured deltas that map one graph state to another. We give a declarative description of the insertion and deletion of graph elements and the in-place modification of labels and properties, and explain how these operations compose with pattern-based reads, identifier allocation, and constraint checking. The model makes implementation-defined details such as reference value types explicit, but is otherwise independent of any particular execution strategy.

Semantic interfaces used in this chapter. Let **GTypeDesc** denote the domain of graph-type descriptors. A graph entry has the abstract form

$$\mathbf{GraphEntry} = \langle G, \mathsf{GT}^{?}, \mathcal{I}_G \rangle,$$

where G is its graph, $\mathsf{GT}^{?}$ is either a reference to a descriptor in **GTypeDesc** or none, and $\mathcal{I}_G$ is the dictionary of constraints defined directly on the graph. The current working graph site has a declared graph-reference type and is occupied by a graph reference value. That reference identifies the graph denoted here by G_{cur}; its corresponding graph entry or graph descriptor supplies the optional constraining graph-type reference and constraint dictionary. Write $\mathsf{GE}_{G_{\mathrm{cur}}}$ for that entry or descriptor. When the graph is written G, GE denotes the corresponding active entry or descriptor. Let $\mathbb{AID}$ denote the domain of session authorization identifiers and let $u \in \mathbb{AID}$ denote the identifier under which a statement is executed. Finally, let $\mathbb{STATUS}$ denote the domain of five-character GQLSTATUS codes. To raise a code $\kappa \in \mathbb{STATUS}$ is to produce a failed outcome, which has no statement result. Let $\mathsf{omitted}$ denote an omitted statement result, and write $\mathsf{success}(R)$ for a successful outcome whose result is R, abstracting over its status objects; this is the projection of the $\mathsf{Ok}(R, \zeta, \overline{\zeta})$ outcome defined in Chapter 21. The later chapters on types, constraints, catalogs, authorization, and diagnostics refine these interfaces.

Implementation configuration.

We make the implementation-defined choices used below explicit by an *implementation configuration* $\mathcal{K}$. Formally, let

$$\mathcal{K} = \begin{array}{l} \langle \mathsf{RefTy}_V,\ \mathsf{RefTy}_E,\ \mathsf{LabMin}_V,\ \mathsf{LabMax}_V,\ \mathsf{LabMin}_E,\ \mathsf{LabMax}_E, \\ \ \mathsf{PropMax}_V,\ \mathsf{PropMax}_E \rangle \end{array}$$

where:

- $\mathsf{RefTy}_V, \mathsf{RefTy}_E$ are implementation-defined reference value types for nodes and edges;
- $\mathsf{LabMin}_V, \mathsf{LabMax}_V \in \mathbb{N} \cup \{\infty\}$ and $\mathsf{LabMin}_E, \mathsf{LabMax}_E \in \mathbb{N} \cup \{\infty\}$ are the supported minimum and maximum cardinalities of node and edge label sets, respectively;
- $\mathsf{PropMax}_V, \mathsf{PropMax}_E \in \mathbb{N} \cup \{\infty\}$ are the supported maximum numbers of node and edge properties.

For the current working graph G_{cur}, the constraint dictionary $\mathcal{I}_G$ in $\mathsf{GE}_{G_{\text{cur}}}$ contains the constraints defined directly for that graph. After a data-modifying operation, those constraints are dynamically validated against the modified graph. The subsequent graph-type conformance check, when a constraining graph type is present, also accounts for the constraints of that graph type. Constraints registered for unrelated graph or graph-type subjects are not consulted.

Whenever we write "implementation-defined" in this chapter, the corresponding choice is assumed to be represented in $\mathcal{K}$.

Syntactic frame for data modification. At the concrete-syntax level, a *linear data-modification statement* is a pipeline consisting of an optional graph-selection header followed by simple data-accessing steps, at least one of which is data-modifying, and an optional primitive result statement at the end. The following highly simplified BNF captures this spine; detailed productions for each kind of query and DML primitive are given in later sections.

```
linear-dm-stmt ::= [ "USE" graph-expr ] dm-body [ result-step ]
dm-body        ::= { data-access-step } dm-primitive
                   { data-access-step }
data-access-step ::= query-step | dm-primitive
dm-primitive   ::= insert-stmt
              | set-stmt
              | remove-stmt
              | delete-stmt
              | "CALL" dm-procedure  /* data-modifying procedure */
query-step     ::= ... /* simple query statement */
result-step    ::= ... /* FINISH, RETURN, or SELECT; final only */
graph-expr     ::= ... /* graph reference / expression */
dm-procedure   ::= ... /* procedure name + arguments */
```

Semantically, the query steps determine a working table of bindings, while the data-modification primitives contribute a staged write-set (delta) relative to a chosen graph. The remainder of this chapter explains how the primitives in `dm-primitive` contribute

node/edge insertion, deletion, and property/label updates, and how the resulting delta is validated and applied.

The simple data-modification statements require optional feature GD01; label additions and removals require GD02. A delete item containing a procedure body requires GD03; one simply containing a value expression other than a binding-variable reference requires GD04. Insertion of an undirected edge requires GH02.

14.1 INSERT and CREATE

An INSERT statement inserts new nodes and edges into the current working graph. Its body is an *insert graph pattern* composed of insert path patterns that (i) may declare element variables for newly created nodes/edges and (ii) may specify label and property sets for those elements. The statement has an empty declared type and returns an omitted result; it may, however, augment the working table with columns for declared element variables (see below).

Concrete syntax for INSERT. At the level of concrete syntax, INSERT uses a compact, path-shaped notation that mirrors the pattern language of MATCH. New nodes and edges are described by node and edge patterns, which may optionally introduce element variables, label lists, and property maps. The following simplified BNF sketch outlines the core structure of the INSERT pattern language.

```
insert-stmt    ::= "INSERT" insert-pattern
insert-pattern ::= insert-path { "," insert-path }
insert-path    ::= node-pattern { edge-pattern node-pattern }
node-pattern   ::= "(" [ var ] [ ( ":" | "IS" ) label-set ]
                                    [ property-map ] ")"
edge-pattern   ::= "-["  [ var ] [ ( ":" | "IS" ) label-set ]
                                    [ property-map ] "]->"
                 | "<-[" [ var ] [ ( ":" | "IS" ) label-set ]
                                    [ property-map ] "]-"
                 | "~["  [ var ] [ ( ":" | "IS" ) label-set ]
                                    [ property-map ] "]~"
label-set      ::= label { "&" label }
property-map   ::= "{" ... "}"  /* key: value pairs,
                                   as in expressions */
var            ::= identifier
label          ::= identifier
```

Intuitively, each insert-path contributes one or more insert node/edge patterns to the abstract insert graph pattern IGP: an isolated node-pattern describes a node insertion, while each adjacent edge-pattern node-pattern extends the path with an edge and node.

The semantic clauses below reinterpret these syntactic patterns as scheduled insertions of nodes and edges, equipped with label and property sets derived from the specified label lists and property maps. The undirected form requires optional feature `GH02`.

Typing and new columns. An insert element pattern is *bound* if the name of its declared element variable already occurs as a column of the incoming working table type or as a field of the incoming working record type. Its declared type is inherited from that column or field. Such a pattern reuses the corresponding incoming node reference and does not introduce a new column.

For every other insert element pattern in IGP that declares an element variable EV by name EVN, `INSERT` contributes a column named EVN whose type is an implementation-defined reference value type covering all nodes (resp. edges) that `INSERT` can bind to EV. If that type is *closed*, then each property key/value in the pattern is store-assigned as if to a transient site of the corresponding property type. Columns whose names are *system-generated* in IGP (introduced when an element variable name is omitted) are removed from the outgoing working table. The incoming record type is preserved; the outgoing table type equals the union of the incoming columns and the new EVN columns, minus the system-generated ones. The current working graph site must be defined.

Operational semantics. Let TABLE be the current working table, let G_{cur} be the graph referenced by the current working graph site, and let IDNSET be the set of system-generated variable names introduced by IGP. Execution of `INSERT` IGP transforms these objects according to

$$\langle G_{\mathrm{cur}}, \mathrm{TABLE} \rangle \xRightarrow{\texttt{INSERT IGP}} \langle G'_{\mathrm{cur}}, \mathrm{TABLE}', \mathsf{success}(\mathsf{omitted}) \rangle,$$

provided that no exception is raised. Here G'_{cur} is obtained from G_{cur} by performing all node and edge insertions scheduled by the procedure below, and TABLE' is obtained by extending each input record with reference values for the newly inserted graph elements declared by non-bound insert element patterns, and then removing the columns whose names are system-generated.

Initially, let NEW_TABLE be an empty binding table whose columns are the union of the columns of TABLE and the columns contributed by the element variables declared by non-bound insert element patterns in IGP. For each record $R \in \mathrm{TABLE}$, process IGP in a fresh child execution context amended with R. Construct a record CR containing the reference values of the graph elements created for R, by processing each insert path pattern immediately contained in IGP as follows:

1. *Inserting node pattern.* For every insert node pattern INP that is the defining pattern for an element variable EV, has the name EVN, and is not a bound insert element pattern:
   a) Determine the node label set NLNS, taking NLNS $= \emptyset$ if no label set is specified. Enforce the implementation-defined minimum and maximum node-label-set cardinalities. If |NLNS| is below the supported minimum, raise 22G0N (*number of node labels below supported minimum*); if it exceeds the supported maximum, raise 22G0P (*number of node labels exceeds supported maximum*).
   b) Compute the node property set NPS from the property key/value list, taking NPS $= \emptyset$ if no such list is specified. Each property value expression is evaluated in the child execution context and store-assigned to the corresponding new property. If the number of properties exceeds the implementation-defined maximum, raise 22G0S (*number of node properties exceeds supported maximum*).
   c) Schedule the insertion into G_{cur} of a new node NN with a new global object identifier, label set NLNS, and property set NPS. Let NR be the node reference value for NN, and include in CR the field named EVN with value NR. The same reference value is used by other insert element patterns that refer to this defining node pattern.
2. *Inserting edge pattern.* For every insert edge pattern IEP that declares an element variable EV with name EVN:
   a) Determine the edge label set ELNS, taking ELNS $= \emptyset$ if no label set is specified, and enforce the implementation-defined minimum and maximum edge-label-set cardinalities. If |ELNS| is below the supported minimum, raise 22G0Q (*number of edge labels below supported minimum*); if it exceeds the supported maximum, raise 22G0R (*number of edge labels exceeds supported maximum*).
   Compute the edge property set EPS analogously, evaluating each property value expression in the child execution context and store-assigning its result to the corresponding new property. If the number of properties exceeds the implementation-defined maximum, raise 22G0T (*number of edge properties exceeds supported maximum*).
   b) Determine the left and right endpoint nodes from the nearest surrounding insert node patterns. If an endpoint pattern is a bound insert element pattern, use the node denoted by its bound element variable; otherwise, use the node reference value computed for its defining insert node pattern. If either endpoint node is deleted, raise G1002 (*endpoint node is deleted*). If either endpoint node is not in the current working graph, raise G1003 (*endpoint node not in current working graph*).

Determine whether the new edge is directed or undirected from the syntactic form of IEP. For a directed edge, use the arrow orientation to determine which endpoint is the source and which is the destination.

c) Schedule the insertion into G_{cur} of a new edge NE with a new global object identifier, label set ELNS, property set EPS, and the determined endpoints, direction, and orientation. Let ER be the edge reference value for NE, and include in CR the field named EVN with value ER.

After all immediately contained insert path patterns have been processed for R, construct a record NR_R by amending R with CR, and append NR_R to NEW_TABLE.

After all records of TABLE have been processed, perform all scheduled node insertions, followed by all scheduled edge insertions. Let G'_{cur} be the resulting provisional graph. Dynamically validate every direct constraint $C \in \text{ran}(\mathcal{I}_G)$ on G'_{cur}; an enforcement failure raises 23000 (*integrity constraint violation*). If G_{cur} has a constraining graph type GT and G'_{cur} is not of GT, raise G2000 (*graph type violation*). Consequently, a successful `INSERT` has passed the required direct-constraint validation and graph-type conformance check, including the constraints of that graph type.

Finally, let TABLE′ be a copy of NEW_TABLE with all columns whose names belong to IDNSET removed, and set TABLE′ as the current working table. The execution outcome is successful with an omitted result; thus, `INSERT` modifies the graph and propagates the extended working table, but does not itself return a result value. See Section 4.3 for a complete insertion of library nodes, edges, labels, and properties.

Identifiers and allocation. Every created graph element (node or edge) receives a new *global object identifier*; copies likewise obtain fresh identifiers. Reference values reify globally resolved references to such objects. Element variable names that the syntax supplies automatically are system-generated regular identifiers and are *not* retained as output columns.

`CREATE GRAPH` (catalog). A `CREATE GRAPH` statement creates a new graph using an open `ANY` graph form or constraining it by an existing graph type reference, an inline graph type specification, or a graph type *LIKE* another graph. When a source graph is provided, all nodes and edges are *copied* into the new graph and each copy is assigned a new global object identifier. The created graph descriptor records the constraining graph type (if any). A violation of a constraint defined directly on the graph raises 23000; a graph-type violation raises G2000.

14.2 DELETE

Concrete syntax for `DELETE`. `DELETE` statements operate on a list of expressions whose declared types are graph-element reference value types. At runtime, valid node and edge

references are collected for deletion, while the null value and invalidated references are ignored without accessing their referents. An optional prefix keyword controls whether incident edges are automatically removed or must already be absent. The following simplified BNF captures the core concrete form:

```
delete-stmt ::= [ "DETACH" | "NODETACH" ] "DELETE"
                delete-arg { "," delete-arg }
delete-arg  ::= expr  /* must evaluate to a node or
                                 edge reference */
expr        ::= ...   /* value expression */
```

Intuitively, a `delete-stmt` with `NODETACH` (or without any prefix, for which GQL makes `NODETACH` implicit) is interpreted using the *plain delete* semantics below, which require that every incident edge of a deleted node is also designated for deletion. A statement prefixed by `DETACH` is interpreted using the *detach delete* semantics, which first removes incident edges and then removes the node itself. In both cases, the deletion of edges referenced directly by `delete-arg` follows the edge-deletion rules outlined next.

Incident edges. For $x \in V$, let $\mathrm{inc}(x) = \{e \in E \mid \mathrm{src}(e) = x \ \vee \ \mathrm{dst}(e) = x\}$.

Plain `DELETE`. A plain `DELETE`, for which `NODETACH` is implicit, first collects the sets $V_{\mathrm{del}} \subseteq V$ and $E_{\mathrm{del}} \subseteq E$ of nodes and edges designated for deletion. For every $v \in V_{\mathrm{del}}$, deletion is permitted only if

$$\mathrm{inc}(v) \subseteq E_{\mathrm{del}};$$

otherwise, the statement signals a dependent-object error (G1001, edges still exist). Every $e \in E_{\mathrm{del}}$ is deleted unconditionally, without affecting its endpoint nodes. If the condition holds for all $v \in V_{\mathrm{del}}$, all designated nodes and edges are deleted atomically, together with their labels and properties.

Example (plain deletion of an edge).

```
USE libraryGraph
MATCH (l:Loan)-[i:INCLUDES]->(b:Book)
WHERE b.isbn = '978-83-987...'
DELETE i
```

The statement deletes only the selected `INCLUDES` edge; its endpoint nodes `l` and `b` remain in the graph.

`DETACH DELETE`. For a node $v \in V$, `DETACH DELETE` v adds every edge $e \in \mathrm{inc}(v)$ to the set of elements to be deleted and then deletes v. It does *not* recursively delete nodes adjacent to v. Additional nodes must be designated explicitly by including them as

separate items in the comma-separated delete-item list, as in `DETACH DELETE` v_1`,` v_2. See Section 4.3 for a concrete use of `DETACH DELETE` in the library graph.

Constraint validation. The sets of nodes and edges to be deleted, including the incident edges added by `DETACH`, are collected before any deletion is performed. The resulting copy G' is then checked as one atomic result. Its direct constraints are dynamically validated, and, if the graph has a constraining graph type, G' is checked for conformance to that type. Deleting elements cannot by itself create a violation of the key or unique constraints defined by GQL. A graph-type violation raises G2000; this check is retained for implementation-defined extensions and future graph-type constraints even though the constraints currently defined by GQL do not make it reachable by deletion alone.

14.3 Read-then-Update Composition

A data-modifying statement commonly combines a read phase with an update phase. The read query Q is evaluated first against the current graph state G, under the authorization context of principal u, producing a binding table

$$B_Q^u \;:=\; [\![Q]\!]_{G,u}.$$

Each record in B_Q^u identifies graph elements or supplies values that may be used by the subsequent update U. Thus, the result of Q does not itself modify the graph; rather, it determines the bindings with respect to which U is evaluated.

For an update part U, graph G, binding table B, and authorization identifier u, write $\Delta_U(G, B, u)$ for the coalesced update delta produced by evaluating U for all records of B, as defined concretely in Section 14.4. The partial expression $G \oplus \Delta$ denotes application of a delta Δ to G. Let $\mathsf{GE} = \langle G, \mathsf{GT}^?, \mathcal{I}_G \rangle$ be the active graph entry. The full validation predicate is

$$\begin{aligned} \mathsf{ok}_{\mathsf{GE}}(G, \Delta) \quad \Longleftrightarrow \quad & G \oplus \Delta \text{ is defined,} \\ & \text{satisfies every direct constraint in } \operatorname{ran}(\mathcal{I}_G), \\ & \text{and is of its constraining graph type (if any).} \end{aligned}$$

The update phase computes a graph delta

$$\Delta_{U,Q}^u \;:=\; \Delta_U\big(G, B_Q^u, u\big),$$

which collects the insertions, deletions, label changes, and property changes requested by U for the bindings produced by Q. Conceptually, U is evaluated for the records of B_Q^u, and the individual changes are combined into a single delta. If B_Q^u is empty, an update

whose effects depend entirely on those bindings normally produces the empty delta and leaves the graph unchanged.

The denotation of the composed read-then-update operation is

$$[\![U \circ Q]\!]_{G,u} = \begin{cases} G \oplus \Delta^{u}_{U,Q}, & \text{if principal } u \text{ is authorized to execute both } Q \text{ and } U, \\ & \quad \text{and } \mathsf{ok}_{\mathsf{GE}}\left(G, \Delta^{u}_{U,Q}\right), \\ \mathsf{error}, & \text{otherwise.} \end{cases}$$

Here, $G \oplus \Delta^{u}_{U,Q}$ denotes the graph obtained by applying the complete delta to G. The authorization condition covers both access to the data inspected by Q and permission to perform the modifications requested by U. The predicate $\mathsf{ok}_{\mathsf{GE}}\left(G, \Delta^{u}_{U,Q}\right)$ states that the delta is applicable to G, that the resulting graph satisfies every direct constraint, and that it conforms to its constraining graph type, including the constraints of that graph type.

Accordingly, the composition is atomic at the level of this denotation: either the complete authorized and valid delta is applied, producing the new graph state, or the operation yields an error and no resulting graph state is defined by the statement.

14.4 Write-Set Coalescing and Conflicts

Concrete syntax for `SET`. A `SET` statement updates properties and labels of elements named by variables that are bound by preceding query steps (for example, by `MATCH`). Each item in the `SET` list either assigns a single property, replaces the entire property map, or adds a label to the element's label set. The following simplified BNF describes the core concrete form:

```
set-stmt ::= "SET" set-item { "," set-item }

set-item ::= var "." prop "=" expr
    | var "=" "{" ... "}" /* replace all properties of var */
    | var ( ":" | "IS" ) label /* add label to var */

var   ::= identifier /* binding variable from MATCH, etc. */
prop  ::= identifier /* property name */
label ::= identifier
expr  ::= ...     /* value expression */
```

Informally, a `set-item` of the first form assigns a value to a single property of the element denoted by `var`, the second form replaces all existing properties of that element by the given map, and the third form adds a label to the element's label set (using either the shorthand `var:Label` or the more verbose `var IS Label`). In the semantic model below, these surface forms are reinterpreted as combinations of primitive property-setting,

property-clearing, and label-addition atoms that constitute an update template. See Section 4.3 for a concrete read-then-`SET` operation on a library edge property.

Update context. For a graph pattern P, predicate ϕ, and concrete write part U, an *update pipeline* is represented at the meta-level as

$$\texttt{MATCH}\ P\ \texttt{WHERE}\ \phi\ \triangleright U$$

where $\triangleright$ is meta-notation, not a GQL keyword; examples of U include `SET`, `REMOVE`, and `DELETE`. The read part yields a bag $\mathcal{M} = [\![P\ \texttt{WHERE}\ \phi]\!]_G$ of mappings over variables (node-, edge-, and path-variables). Updates are then staged per $\mu \in \mathcal{M}$ and coalesced before application to the transaction-visible working state.

Targets. Evaluation of a target binding-variable reference under μ yields either a graph-element reference or the null value. A null target contributes no modification. A *target reference* that is used for an update is therefore either a node-id $v \in V$ or an edge-id $e \in E$; accessing the referent of an invalidated reference raises 22G11. We write $\mathrm{Tgt}(\mu) \subseteq V \cup E$ for the valid targets referenced by U under μ.

Update atoms. For $v \in V$, $e \in E$, $k \in \mathbb{K}$, $\ell \in \mathbb{L}$, and a value expression *expr*, the primitive update atoms are listed below. Here $\rho_V^+, \rho_E^+, \lambda_V^+, \lambda_E^+$ denote the provisional post-update maps obtained from $\rho_V, \rho_E, \lambda_V, \lambda_E$ by the indicated assignment; every unmentioned map entry is unchanged.

$$\begin{aligned}
\texttt{SET_NODE_PROP}(v,k,\mathit{expr}) &: \rho_V^+(v,k) \leftarrow [\![\mathit{expr}]\!]_\mu \\
\texttt{SET_EDGE_PROP}(e,k,\mathit{expr}) &: \rho_E^+(e,k) \leftarrow [\![\mathit{expr}]\!]_\mu \\
\texttt{CLEAR_NODE_PROPS}(v) &: \rho_V^+(v,\cdot) \leftarrow \emptyset \\
\texttt{CLEAR_EDGE_PROPS}(e) &: \rho_E^+(e,\cdot) \leftarrow \emptyset \\
\texttt{REMOVE_NODE_PROP}(v,k) &: \rho_V^+(v,k) \leftarrow \mathsf{absent} \\
\texttt{REMOVE_EDGE_PROP}(e,k) &: \rho_E^+(e,k) \leftarrow \mathsf{absent} \\
\texttt{ADD_NODE_LABEL}(v,\ell) &: \lambda_V^+(v) \leftarrow \lambda_V(v) \cup \{\ell\} \\
\texttt{ADD_EDGE_LABEL}(e,\ell) &: \lambda_E^+(e) \leftarrow \lambda_E(e) \cup \{\ell\} \\
\texttt{REMOVE_NODE_LABEL}(v,\ell) &: \lambda_V^+(v) \leftarrow \lambda_V(v) \setminus \{\ell\} \\
\texttt{REMOVE_EDGE_LABEL}(e,\ell) &: \lambda_E^+(e) \leftarrow \lambda_E(e) \setminus \{\ell\}.
\end{aligned}$$

The two property-setting rows abbreviate the standard store-assignment rules, including their nullability and conversion checks. A surface item `var = {...}` contributes the appropriate `CLEAR` atom followed by one property-setting atom for each property pair in textual order. Endpoint reattachment is *not* an in-place update; changing endpoints is modeled as deletion followed by recreation.

Notational shorthands (ground update operations). The primitive atoms above mix two concerns: (i) the *target* (a concrete node/edge id) and (ii) an r-value *expression* that must be evaluated under a mapping μ. For examples and for write-set reasoning, it is convenient to use *ground* operations where the r-value has already been evaluated to a concrete value.

Let $v \in V$ be a node identifier, $a \in E$ an edge identifier, $k \in \mathbb{K}$ a property key, $\ell \in \mathbb{L}$ a label, q a value expression, and $w \in \mathbb{V}$ a concrete value. Let $\mathsf{ground}_\mu(\cdot)$ be the function that maps update atoms to ground operations by evaluating r-values under μ. We use the following shorthands:

$\mathsf{ground}_\mu(\texttt{SET_NODE_PROP}(v,k,q))$	$\triangleq$	$\mathsf{setPropV}(v,k,[\![q]\!]_\mu)$	(set vertex property)
$\mathsf{ground}_\mu(\texttt{SET_EDGE_PROP}(a,k,q))$	$\triangleq$	$\mathsf{setPropE}(a,k,[\![q]\!]_\mu)$	(set edge property)
$\mathsf{ground}_\mu(\texttt{CLEAR_NODE_PROPS}(v))$	$\triangleq$	$\mathsf{clearPropsV}(v)$	(clear all vertex properties)
$\mathsf{ground}_\mu(\texttt{CLEAR_EDGE_PROPS}(a))$	$\triangleq$	$\mathsf{clearPropsE}(a)$	(clear all edge properties)
$\mathsf{ground}_\mu(\texttt{REMOVE_NODE_PROP}(v,k))$	$\triangleq$	$\mathsf{delPropV}(v,k)$	(delete vertex property)
$\mathsf{ground}_\mu(\texttt{REMOVE_EDGE_PROP}(a,k))$	$\triangleq$	$\mathsf{delPropE}(a,k)$	(delete edge property)
$\mathsf{ground}_\mu(\texttt{ADD_NODE_LABEL}(v,\ell))$	$\triangleq$	$\mathsf{addLabelV}(v,\ell)$	(add vertex label)
$\mathsf{ground}_\mu(\texttt{ADD_EDGE_LABEL}(a,\ell))$	$\triangleq$	$\mathsf{addLabelE}(a,\ell)$	(add edge label)
$\mathsf{ground}_\mu(\texttt{REMOVE_NODE_LABEL}(v,\ell))$	$\triangleq$	$\mathsf{delLabelV}(v,\ell)$	(remove vertex label)
$\mathsf{ground}_\mu(\texttt{REMOVE_EDGE_LABEL}(a,\ell))$	$\triangleq$	$\mathsf{delLabelE}(a,\ell)$	(remove edge label).

Thus, for example, $\mathsf{setPropV}(v,k,w)$ abbreviates the *ground effect* "set the vertex-property (v,k) to the value w".

Templates and determinism. An *update template* U is a finite, textually ordered list of atoms whose l-values are expressions over μ. For a fixed μ, evaluation produces a *per-binding write-set* $\mathsf{WS}_\mu(U)$, represented as a list of *ground* operations on concrete element identifiers. Concretely, $\mathsf{WS}_\mu(U)$ is obtained by applying ground_μ to the atoms of U, evaluating all r-values under μ, while preserving their textual order. A set item whose target is null contributes the empty list. A set-all- properties item contributes a clear operation followed by its evaluated property assignments; an attempted access through an invalidated target raises 22G11 rather than contributing a write.

Standard options. For a single `SET` statement that would make multiple assignments to the same (id, k), even if the assigned values are equal, the GQL standard specifies an implementation-defined choice between two behaviors:

(a) signal data exception 22G0M, or
(b) make an implementation-dependent choice of one assignment.

In this book we adopt a *canonical partial coalescer* $\boxplus$ that realizes option (a) while preserving the order of records and set items required by the standard. Property-assignment operations are pairwise compatible only when they target distinct (id, k) pairs. Label additions commute, and repeated additions of the same label to the same element are idempotent and may coalesce to one effective addition. Clear-all-properties operations are retained at their ordered positions. Whenever two property assignments target the same (id, k) in one statement, $\boxplus$ is undefined and the statement signals 22G0M. Different μ-instances are folded in incoming-table order, subject to these rules.

Staged delta and application. Identifying the binding table B with its bag $\mathcal{M}$ of mappings, the *staged delta* introduced in Section 14.4 is

$$\Delta_U(G, B, u) \;:=\; \boxplus_{\mu \in \mathcal{M}} \mathsf{WS}_\mu(U),$$

where the indexed $\boxplus$ folds the partial coalescer and is defined only when the combined write-set is pairwise-compatible as above. Within this section, Δ_U abbreviates $\Delta_U(G, B, u)$. Applying the ordered Δ_U to G yields G' by clearing property sets for set-all-properties items, performing their property assignments in textual order, overwriting individual properties, adding labels, and leaving all other components unchanged. After each non-null set item, exceeding the supported maximum for a node label set, node property set, edge label set, or edge property set raises 22G0P, 22G0S, 22G0R, or 22G0T, respectively. The direct constraints on G are dynamically validated against G'. If G has a constraining graph type GT and G' is not of GT, the statement raises G2000; this conformance check includes the constraints of GT.

Read stability. All l-values and r-values in U are evaluated over the *pre-update working state* that produced $\mathcal{M}$. Thus, intra-statement reads do not observe effects of the same statement's writes; only the final Δ_U is materialized atomically. This statement-level evaluation rule does not by itself select the transaction isolation extension discussed in Chapter 19.

Examples of coalescing and conflict detection. The following examples illustrate how the canonical coalescer handles repeated updates generated by bag-valued pattern matching. The first example considers multiple identical and idempotent label additions, which can

be safely reduced to a single effective update. The second considers incompatible property writes to the same target and property key, which cannot be coalesced and therefore result in an intra-statement conflict.

(A) Idempotent label addition under bag matches. Suppose the read pattern P matches the same vertex i with multiplicity 5. Formally, the match result is a bag $\mathcal{M}$ that contains five mappings $\mu_1, \ldots, \mu_5$ such that the l-value of the label update refers to i under each mapping (e.g., $[\![\mathtt{v}]\!]_{\mu_j} = i$ for $j = 1, \ldots, 5$). Let the write part be a single label-add atom $\texttt{ADD_NODE_LABEL}(i, \ell)$.

Then, for each $j = 1, \ldots, 5$, evaluation of the template yields the same per-binding write-set:

$$\mathsf{WS}_{\mu_j}(U) = \big[\texttt{ADD_NODE_LABEL}(i, \ell)\big].$$

Hence, the staged delta is obtained by folding these five identical contributions:

$$\Delta_U \;=\; \biguplus_{\mu \in \mathcal{M}} \mathsf{WS}_\mu(U) \;=\; \mathsf{WS}_{\mu_1}(U) \;\uplus\; \cdots \;\uplus\; \mathsf{WS}_{\mu_5}(U).$$

Because label addition is idempotent and duplicates are pairwise compatible under the canonical coalescer, the five identical atoms coalesce into a single effective update:

$$\Delta_U \;=\; \big\{\texttt{ADD_NODE_LABEL}(i, \ell)\big\}.$$

Therefore, although P matches i five times, the statement is equivalent, with respect to the final store state, to a single addition of ℓ to $\lambda_V(i)$.

(B) Conflicting property writes. Consider a mapping $\mu \in \mathcal{M}$ and two syntactically distinct l-values, for example `x.k` and `y.k`, that satisfy the static non-duplication rule but whose variable references alias at runtime: $\mu(\mathtt{x}) = \mu(\mathtt{y}) = i$. Assume that their expressions produce two *different ground* writes to the same pair (i, k):

$$\mathsf{setPropV}(i, k, 1) \in \mathsf{WS}_\mu(U) \qquad \text{and} \qquad \mathsf{setPropV}(i, k, 2) \in \mathsf{WS}_\mu(U), \quad \text{with } 1 \neq 2.$$

Equivalently, this arises from the atoms

$$\texttt{SET_NODE_PROP}(i, k, e_1) \quad \text{and} \quad \texttt{SET_NODE_PROP}(i, k, e_2),$$

such that $[\![e_1]\!]_\mu = 1$ and $[\![e_2]\!]_\mu = 2$.

Under the standard-permitted option (a) adopted by the canonical model here, the coalescer $\boxplus$ is defined only for pairwise-compatible operations. Since the two ground writes target the same pair (i, k) but prescribe different values, their union is not compatible.

Consequently, $\boxplus$ is undefined, the indexed $\boxplus$ rejects the combined write-set, and the statement signals 22G0M, irrespective of the textual ordering of the two writes.

Concrete syntax for `REMOVE`. A `REMOVE` statement removes properties and labels from elements named by variables that are bound by preceding query steps (typically by `MATCH`). The following simplified BNF mirrors the style of `SET` and isolates the core concrete form of the statement:

```
remove-stmt ::= "REMOVE" remove-item { "," remove-item }

remove-item ::= var "." prop
              | var ( ":" | "IS" ) label

var   ::= identifier  /* binding variable from MATCH, etc. */
prop  ::= identifier  /* property name */
label ::= identifier  /* label name */
```

Each `remove-item` uses a binding variable whose declared type is a graph-element reference (node or edge). The first form `var.prop` schedules the removal of a single property (if present) from the element denoted by `var`. The second form `var:Label` or `var IS Label` schedules removal of a label (if present) from the element's label set. The statement has empty declared type and does not change the working table: incoming and outgoing working record/table types coincide, and the result is omitted.

Remove templates and atoms. We reuse the update context and the notion of an update target introduced for `SET`. Consider an update pipeline

$$\texttt{MATCH}\ P\ \texttt{WHERE}\ \phi\ \texttt{REMOVE}\ I_1, \ldots, I_m,$$

whose read part yields the bag

$$\mathcal{M} = [\![P\ \texttt{WHERE}\ \phi]\!]_G$$

of mappings over node, edge, and path variables. Each occurrence of a mapping in $\mathcal{M}$ is processed separately; hence duplicate mappings may generate duplicate removal atoms.

The construction of the update proceeds in three stages. First, for every mapping occurrence $\mu \in \mathcal{M}$ and every remove item I_j, the binding-variable reference contained in I_j is evaluated under μ:

$$t_{\mu,j} := [\![\texttt{var}_j]\!]_{\mu,G}.$$

This *target-resolution stage* determines the node or edge to which the item refers but does not yet modify the graph. In particular, all targets are resolved with respect to the bindings produced by the read part before the corresponding removal effects are applied.

Second, the resolved items are inspected in textual order. Each item contributes either one concrete removal atom or no atom. Writing ϵ for the empty list, let

$$A_G(\mu, I_j) \in \left\{ \begin{array}{l} \epsilon, \\ [\texttt{REMOVE_NODE_PROP}(v, k)],\ [\texttt{REMOVE_EDGE_PROP}(e, k)], \\ [\texttt{REMOVE_NODE_LABEL}(v, \ell)],\ [\texttt{REMOVE_EDGE_LABEL}(e, \ell)] \end{array} \right\}$$

be the contribution of I_j. The per-binding write-set is the textual-order concatenation

$$\mathsf{WS}_\mu(U_{\text{rem}}) = A_G(\mu, I_1) +\!\!+ \cdots +\!\!+ A_G(\mu, I_m),$$

where $+\!\!+$ denotes list concatenation.

Finally, the per-binding write-sets may be folded over all occurrences to give an extensional summary of the removals completed by a successful statement:

$$\Delta_{\text{rem}} = \biguplus_{\mu \in \mathcal{M}} \mathsf{WS}_\mu(U_{\text{rem}}).$$

This delta notation is not a replacement for the operational order required by `REMOVE`. Records are processed in current-working-table order and, for each record, remove items are performed in textual order. The applicable label-cardinality check is made immediately after each label item. After every remove item, the graph's direct constraints are dynamically validated and its conformance to the constraining graph type is checked. An exception stops that ordered execution. Repeated removals remain harmless once the property or label is absent.

The contribution $A_G(\mu, I_j)$ is determined as follows.

- *Property removal* `var.prop`. Let
$$t = t_{\mu,j}$$
be the previously resolved target and let k be the property name.
If t is the null value, then the item has no effect:
$$A_G(\mu, I_j) = \epsilon.$$
If t is an invalidated reference, inspecting its property set raises 22G11. Otherwise:
  - if $t \in V$ and k occurs in the node-property set of t, then
$$A_G(\mu, I_j) = [\texttt{REMOVE_NODE_PROP}(t, k)];$$

– if $t \in E$ and k occurs in the edge-property set of t, then

$$A_G(\mu, I_j) = [\texttt{REMOVE_EDGE_PROP}(t, k)];$$

– if k is not present in the property set of t, then

$$A_G(\mu, I_j) = \epsilon.$$

- *Label removal* `var:Label` or `var IS Label`. Let

$$t = t_{\mu,j}$$

be the previously resolved target and let ℓ be the label name.
If t is the null value, then the item has no effect:

$$A_G(\mu, I_j) = \epsilon.$$

If t is an invalidated reference, inspecting its label set raises 22G11. Otherwise:

– if $t \in V$ and $\ell \in \lambda_V(t)$, then

$$A_G(\mu, I_j) = [\texttt{REMOVE_NODE_LABEL}(t, \ell)];$$

– if $t \in E$ and $\ell \in \lambda_E(t)$, then

$$A_G(\mu, I_j) = [\texttt{REMOVE_EDGE_LABEL}(t, \ell)];$$

– if ℓ is not contained in the label set of t, then

$$A_G(\mu, I_j) = \epsilon.$$

Thus, $\mathsf{WS}_\mu(U_{\mathrm{rem}})$ records the removal effects generated by one occurrence of a match, whereas Δ_{rem} collects the effects generated by the complete match bag. Applying Δ_{rem} removes the indicated properties and labels without deleting the affected nodes or edges themselves.

Extensional coalescing for `REMOVE`. For describing the final graph of a successful execution, write-sets generated for different binding occurrences can be summarized using the canonical coalescer $\boxplus$:

$$\Delta_{\mathrm{rem}} := \boxplus_{\mu \in \mathcal{M}} \mathsf{WS}_\mu(U_{\mathrm{rem}}).$$

Because U_{rem} contains only removal atoms, all contributions are pairwise compatible:

- Removing the same property or label from the same element multiple times is idempotent; duplicate atoms may therefore be coalesced into a single removal.
- Removing different properties or labels, whether from the same or different graph elements, commutes; their order does not affect the final graph state.

Consequently, $\boxplus$ is defined when combining raw write-sets that contain only removal atoms, and there is no analog of the *multiple assignments to the same property* conflict (status 22G0M) that may arise for `SET`. This extensional commutativity does not license reordering the operational steps: intermediate label-cardinality and graph-type checks remain observable. A successful `REMOVE` statement has a uniquely determined summary Δ_{rem}. This delta may be empty when every remove item resolves to null, or when none of the indicated properties or labels is present; an invalidated reference instead raises 22G11 when its referent is accessed.

Application and label-set cardinalities. Applying Δ_{rem} to G produces a graph G' in which:

- Every `REMOVE_NODE_PROP` and `REMOVE_EDGE_PROP` atom deletes the indicated property from the respective property set, if present.
- Every `REMOVE_NODE_LABEL` and `REMOVE_EDGE_LABEL` atom removes the designated label from the respective label set, if present.

Immediately after each label-removal item, the node-label set $\lambda_V(v)$ (resp. edge-label set $\lambda_E(e)$) must still satisfy the implementation-defined minimum label-set cardinality. If the resulting cardinality for a node falls below the supported minimum, the statement signals data exception 22G0N; if it falls below the supported minimum for an edge, it signals 22G0Q. These label-set cardinality checks are data-exception rules of the operation. After each remove item, the graph's direct constraints are dynamically validated. If the containing graph has a constraining graph type but the graph is not of that type, the operation raises G2000; this conformance check includes the constraints of that graph type.

Read stability. A `REMOVE` statement inherits the same statement-level read stability as `SET`. All variable references used as l-values in `remove-item`s are evaluated over the incoming working-table records. The removals are then performed in the record and item order described above; Δ_{rem} is only an extensional summary of a successful final effect. Transaction rollback rules govern the effects of a failed procedure.

14.5 Update Model for Property Graphs

The mutable graph state is the complete property-graph tuple introduced in Chapter 6:

$$G = \langle V, E, \lambda_V, \rho_V, \lambda_E, \rho_E, \mathrm{src}, \mathrm{dst}, \mathrm{dir} \rangle.$$

Primitive mutations (expressible using GQL data-modification statements) are modeled as deltas over all of these components:

$$\begin{gathered}
\mathsf{insNode}(v, \Lambda, P), \\
\mathsf{delNode}(v), \\
\mathsf{insEdge}(e, \Lambda, P, u, v, d), \\
\mathsf{delEdge}(e), \\
\mathsf{setPropV}(v, k, w), \quad \mathsf{setPropE}(e, k, w), \\
\mathsf{clearPropsV}(v), \quad \mathsf{clearPropsE}(e), \\
\mathsf{delPropV}(v, k), \quad \mathsf{delPropE}(e, k), \\
\mathsf{addLabelV}(v, \ell), \quad \mathsf{addLabelE}(e, \ell), \\
\mathsf{delLabelV}(v, \ell), \quad \mathsf{delLabelE}(e, \ell),
\end{gathered}$$

where Λ and P are the new element's label and property sets, $u, v \in V$ are endpoints, and d records whether an edge is directed and, when directed, its orientation. Application of a well-formed delta updates the corresponding components together; deleting a node is defined only when every incident edge is also deleted. A statement U therefore denotes a partial transformation $G \mapsto G \oplus \Delta_U$. Failure of dynamic validation for a direct constraint in $\mathrm{ran}(\mathcal{I}_G)$ raises 23000. Independently, if the result of a primitive data-modifying statement is not of its constraining graph type, its mandatory conformance check raises G2000.

14.6 Dynamic Constraint Validation

Some DML forms are *set-based*: effects are computed from the whole input multiset to avoid statement-order anomalies (e.g., key and unique-constraint satisfaction), then applied atomically. After a primitive data-modifying operation, every constraint defined directly on the modified graph is dynamically validated. The graph is then checked for conformance to its constraining graph type, when present; that check includes the constraints of the graph type. Constraints registered for unrelated catalog subjects are not consulted.

Chapter 14 — Takeaways

- **Purpose:** Specified how data-modification statements transformed the graph store via structured, compositional update semantics.
- **Introduced:** Update actions as deltas derived from bindings; statement-level conflict detection and write-set coalescing; implementation configuration affecting update behavior.
- **Key distinctions:** Computing updates from the full input multiset versus stepwise application; direct-constraint validation versus graph-type conformance.
- **Consequences:** Made updates analyzable and predictable by describing effects as explicit deltas; ensured well-defined behavior under multiple matches and duplicates; clarified when statements failed due to intra-statement conflicts or constraint interactions.
- **Bridge:** Motivated a static layer (types and schemas) that constrained what graphs and updates were admissible.

Part VII

Types, Schemas, and Constraints

15 Types and Schemas

This chapter introduces the static typing layer of GQL, which constrains the shape of property graphs beyond their raw data model. We describe how property types, node and edge types, and graph types are specified, and how they interact with keys, label tests, and integrity constraints to define admissible graphs.

15.1 Catalog and Name Universes

Universes of names. Let Catalog be the abstract carrier of well-formed hierarchical GQL-catalogs, with $\mathcal{C}, \mathcal{C}' \in \mathsf{Catalog}$. Its internal directory/schema structure is defined in Chapter 17. Let $\mathbb{K}$ be the (implementation-defined) universe of property keys (identifiers admissible as property names). Let $\mathbb{N}_N$, $\mathbb{N}_E$, $\mathbb{N}_G$ be name universes for node types, edge types, and graph types, respectively, and let $\mathbb{N}_{\mathsf{graph}}$ be the universe of graph entry names. These are schema-local name sets used inside the single hierarchical catalog defined in Chapter 17; they are not additional top-level catalogs. Node and edge types are secondary objects contained in graph types.

15.2 Property Types

Type universe. We reuse the value and type universes $\mathbb{V}$ and $\mathbb{T}$ fixed in Section 6.1. Let $\mathbb{T}_{\text{prop}} \subseteq \mathbb{T}$ be the implementation's supported property value types. We distinguish the selected scalar and collection fragments $\mathbb{T}_{\text{scal}}$ and $\mathbb{T}_{\text{coll}}$ of this set, with

$$\mathbb{T}_{\text{scal}} \cup \mathbb{T}_{\text{coll}} \subseteq \mathbb{T}_{\text{prop}}.$$

For $\tau \in \mathbb{T}_{\text{prop}}$, we write $[\![\tau]\!] \subseteq \mathbb{V}$ for its values. Null belongs to this set exactly when τ is nullable.

Scalar types. The selected scalar fragment used below contains (feature-set dependent precisions omitted for brevity)

$$\begin{aligned}
\mathbb{T}_{\text{num}} &= \{\mathsf{INTEGER}, \mathsf{BIGINT}, \mathsf{DECIMAL}(p, s), \mathsf{FLOAT}, \mathsf{DOUBLE}\},\\
\mathbb{T}_{\text{bool}} &= \{\mathsf{BOOLEAN}\},\\
\mathbb{T}_{\text{text}} &= \{\mathsf{STRING}\},\\
\mathbb{T}_{\text{bin}} &= \{\mathsf{BYTES}\},
\end{aligned}$$

$$\mathbb{T}_{\text{temporal}} = \{\mathsf{DATE}, \mathsf{ZONED_TIME}, \mathsf{LOCAL_TIME},\\ \mathsf{ZONED_DATETIME}, \mathsf{LOCAL_DATETIME},\\ \mathsf{DURATION(YEAR\ TO\ MONTH)}, \mathsf{DURATION(DAY\ TO\ SECOND)}\},$$

$\mathbb{T}_{\text{scal}}$ is the union of these sets. (Additional scalar value types from the GQL standard can be added without changing the subsequent development.)

Collection types. One collection fragment used below consists of homogeneous list types:

$$\mathsf{LIST}\langle\tau\rangle \in \mathbb{T}_{\text{coll}} \qquad \text{(finite sequences over } [\![\tau]\!]\text{)}.$$

Other supported property value types, including applicable vector, record, and dynamic union types, remain in $\mathbb{T}_{\text{prop}}$ as governed by the implementation's conformance claim.

The denotation function $[\![\cdot]\!] : \mathbb{T} \to \mathcal{P}(\mathbb{V})$ assigns to each type the set of admissible values.

Nullability and absence. A property type is a pair consisting of a property name and one supported property value type. Consequently, a declaration $k : \tau$ admits exactly values in $[\![\tau]\!]$: it admits a list only when τ is a list type, and it admits null only when τ is nullable. An absent property is not a value and is represented only by the totalized lookup $\widehat{\rho}$ from Section 6.2.

Widening preorder and casts. Define the *widening* preorder $\sqsupseteq$ on value types by

$$\tau_1 \sqsupseteq \tau_0 \quad \Longleftrightarrow \quad [\![\tau_0]\!] \subseteq [\![\tau_1]\!].$$

A *cast* from τ_0 to τ_1 is in general partial, $c : [\![\tau_0]\!] \rightharpoonup [\![\tau_1]\!]$; failure is reported according to the cast and store-assignment rules. An implicit assignment conversion is admissible only where the GQL assignment rules establish it; widening alone does not invent a conversion. For every successful application,

$$v \in \mathrm{dom}(c) \;\Rightarrow\; c(v) \in [\![\tau_1]\!].$$

Static property signatures. For a type descriptor T (node or edge), its *property signature* is a finite map

$$\mathsf{ptype}_T : \; \mathbb{K} \rightharpoonup \mathbb{T}_{\text{prop}}$$

assigning a supported property value type to each declared property key; the run-time admissible values are given directly by the denotation of that type.

Key label sets (KLS). A key label set is the possibly empty label set, subject to supported cardinality limits, associated with a keyed node or edge type in a graph type. The graph type's node or edge key-label-set dictionary uses it to identify the corresponding keyed element type. A key label set is not by itself an integrity constraint.

15.3 Node and Edge Types

Label and property sets. Each GQL node or edge type has a fixed label set and a fixed property-type set. We represent the latter as the finite map ptype_T above. Thus type membership uses equality of label sets and property-name sets, not an open label test or a property-closedness switch.

Reduced node-type descriptor representation. In the reduced book-level representation used here, a node type descriptor is represented by the tuple

$$\mathbf{NDesc} = \langle N,\ L_N,\ \mathsf{ptype}_N \rangle,$$

where N is the local type name, $L_N \subseteq \mathbb{L}$ is its node type label set, and ptype_N is its property signature. Key label sets and named constraints belong to the containing graph type descriptor rather than to this node type descriptor. The fixed primary-base-type name and preferred-name metadata of the full ISO descriptor are elided from this representation.

A node $v \in V$ *conforms* to N (notation: $v :: N$) iff

$$\begin{aligned} \text{(labels)} \quad & \lambda_V(v) = L_N; \\ \text{(property names)} \quad & \mathrm{dom}(\rho_V(v)) = \mathrm{dom}(\mathsf{ptype}_N); \\ \text{(typing)} \quad & \forall k \in \mathrm{dom}(\mathsf{ptype}_N).\ \rho_V(v)(k) \in [\![\mathsf{ptype}_N(k)]\!]. \end{aligned}$$

For example, the `Book` node type declared in Section 4.2 is represented by

$$\begin{aligned} \mathsf{ptype}_{\texttt{Book}} = \{&\texttt{isbn} \mapsto \mathsf{STRING\ NOT\ NULL},\ \texttt{title} \mapsto \mathsf{STRING}, \\ &\texttt{author} \mapsto \mathsf{STRING}\}, \end{aligned}$$

$$\mathbf{NDesc}_{\texttt{Book}} = \Big\langle \texttt{Book},\ \{\texttt{Book}\},\ \mathsf{ptype}_{\texttt{Book}} \Big\rangle.$$

The node b_1 representing `Graph DB 101` in Section 4.1 conforms to `Book`: its label set is $\{\texttt{Book}\}$, its property names are exactly $\{\texttt{isbn}, \texttt{title}, \texttt{author}\}$, and each corresponding value belongs to the denotation of its declared type.

Reduced edge-type descriptor representation. In the reduced book-level representation used here, an edge type descriptor is represented by the tuple

$$\mathbf{ETypeDesc} = \langle \mathsf{ET},\ L_E,\ \mathsf{kind}_E,\ \mathsf{endSpec}_E,\ \mathsf{ptype}_E \rangle,$$

where ET is the local type name; L_E is its edge type label set; $\mathsf{kind}_E \in \{\mathsf{directed}, \mathsf{undirected}\}$ is its directionality; $\mathsf{endSpec}_E$ is an ordered source–destination pair of node types in the directed case and a set of one or two endpoint node types in the undirected case; and ptype_E is its property signature. The fixed primary-base-type name and preferred-name metadata of the full ISO descriptor are elided from this representation.

An edge $e \in E$ with endpoints $(\mathrm{src}(e), \mathrm{dst}(e))$ *conforms* to ET (notation: $e :: \mathsf{ET}$) iff

$$\begin{aligned}
&\text{(labels)} && \lambda_E(e) = L_E;\\
&\text{(property names)} && \mathrm{dom}(\rho_E(e)) = \mathrm{dom}(\mathsf{ptype}_E);\\
&\text{(typing)} && \forall k \in \mathrm{dom}(\mathsf{ptype}_E).\ \rho_E(e)(k) \in [\![\mathsf{ptype}_E(k)]\!];\\
&\text{(kind and endpoints)} && \mathrm{dir}(e) \text{ agrees with } \mathsf{kind}_E \text{ and the endpoints}\\
& && \text{have the node types in } \mathsf{endSpec}_E,
\end{aligned}$$

For a directed type the endpoint check preserves source and destination. For an undirected type it requires the relation between the two endpoints and the one or two endpoint node types to be both left-total and right-total.

For example, let $\mathsf{ET}_{\mathsf{inc}}$ denote the local type name, in the descriptor model used here, of the directed edge type specification carrying the label `INCLUDES` in Section 4.2. It is represented by

$$\mathsf{ptype}_{\mathsf{inc}} = \{\texttt{due_date} \mapsto \mathsf{DATE},\ \texttt{late_fee} \mapsto \mathsf{FLOAT}\},$$

$$\mathbf{ETypeDesc}_{\mathsf{inc}} = \Big\langle \mathsf{ET}_{\mathsf{inc}}, \{\texttt{INCLUDES}\}, \mathsf{directed}, \langle \texttt{Loan}, \texttt{Book} \rangle, \mathsf{ptype}_{\mathsf{inc}} \Big\rangle.$$

The edge $e_{\mathsf{inc},1}$ in the library instance conforms to this type: its label set is $\{\texttt{INCLUDES}\}$, it is directed from the `Loan` node l to the `Book` node b_1, and its two property values have the declared DATE and FLOAT types.

Element-type implication and structural consistency. GQL does not obtain element-type membership from an open-property inheritance rule. Within one graph type, an element type T_1 *implies* a keyed element type T_0 when both have the same element kind, the effective key label set of T_0 is a subset of the label set of T_1, and the key-label-set conditions of the graph type are met. Structural consistency then requires the label set and property-name set of T_0 to be subsets of those of T_1, and each pair of same-named

property value types to combine to the value type declared by T_0 under graph-type-specific combination. Implied edge types must additionally be structurally endpoint-consistent and have the same directionality. This is distinct from the ordinary value-type subtyping relation $\sqsubseteq$ of Section 6.1.

15.4 Graph Types

Reduced structural graph-type descriptor component. In the reduced book-level representation used here, write

$$\Sigma = \langle \mathsf{NDecls},\ \mathsf{EDecls},\ \mathsf{NodeKLS},\ \mathsf{EdgeKLS} \rangle.$$

Here NDecls is a finite map from node type names to **NDesc**, and EDecls is a finite map from edge type names to **ETypeDesc**. The node key-label-set dictionary $\mathsf{NodeKLS}$ maps every node type key label set to its keyed node type; $\mathsf{EdgeKLS}$ maps every edge type key label set to the corresponding set of keyed edge types. The declarations must satisfy GQL's key-label-set implication consistency, property-value-type consistency, and endpoint consistency conditions.

Constraint dictionary (operational part). Let **CDesc** be the abstract carrier of key and unique constraint descriptors; its components and satisfaction semantics are defined in Chapter 16. A graph type comprises a node type set, an edge type set, and a constraint set. Its reduced descriptor representation pairs Σ with the named constraint dictionary $\mathcal{I}_{\mathsf{GT}} : \mathbb{A} \rightharpoonup \mathbf{CDesc}$. See Section 15.5 below for this pair and its use in catalog entries.

Example: `libraryGraphType`. Let $\mathsf{NDecls}_{\mathsf{lib}}$ and $\mathsf{EDecls}_{\mathsf{lib}}$ be the descriptor maps induced by the node- and edge-type declarations in Section 4.2. In particular,

$$\mathsf{NDecls}_{\mathsf{lib}}(\texttt{Book}) = \mathbf{NDesc}_{\texttt{Book}}, \qquad \mathsf{EDecls}_{\mathsf{lib}}(\mathsf{ET}_{\mathsf{inc}}) = \mathbf{ETypeDesc}_{\mathsf{inc}}.$$

The key-label-set dictionaries are

$$\begin{aligned} \mathsf{NodeKLS}_{\mathsf{lib}} = \{&\{\texttt{Loan}\} \mapsto \texttt{Loan}, \\ &\{\texttt{LibraryAccount}\} \mapsto \texttt{LibraryAccount}, \\ &\{\texttt{Reader}\} \mapsto \texttt{Reader}, \\ &\{\texttt{LibraryCard}\} \mapsto \texttt{LibraryCard}, \\ &\{\texttt{Book}\} \mapsto \texttt{Book}\}, \\ \mathsf{EdgeKLS}_{\mathsf{lib}} = \varnothing. \end{aligned}$$

Hence the static schema is

$$\Sigma_{\mathsf{lib}} = \langle \mathsf{NDecls}_{\mathsf{lib}}, \mathsf{EDecls}_{\mathsf{lib}}, \mathsf{NodeKLS}_{\mathsf{lib}}, \mathsf{EdgeKLS}_{\mathsf{lib}} \rangle.$$

The associated constraint dictionary $\mathcal{I}_{\mathsf{lib}}$ satisfies

$$\mathrm{dom}(\mathcal{I}_{\mathsf{lib}}) = \{\texttt{LoanHasKey}, \texttt{LibraryAccountHasKey}, \texttt{ReaderHasKey}, \texttt{LibraryCardHasKey}, \texttt{BookHasKey}\},$$

and each name maps to the key-constraint descriptor induced by the corresponding declaration in Section 4.2.

15.5 Graph Entries and Constraining Types

Constraining graph type descriptor. In the reduced book-level representation used here, a **GTypeDesc** is the pair $\langle \Sigma,\ \mathcal{I}_{\mathsf{GT}} \rangle$, where Σ is the structural descriptor component above and $\mathcal{I}_{\mathsf{GT}}$ maps each constraint name to its key or unique constraint descriptor. The fixed primary-base-type name and preferred-name metadata of the full ISO descriptor are elided from this representation.

A graph G is of the graph type described by $\mathbf{GTypeDesc} = \langle \Sigma, \mathcal{I}_{\mathsf{GT}} \rangle$ exactly when every node of G is of a node type in NDecls, every edge is of an edge type in EDecls, and every constraint of the graph type is satisfied. There is no independent "open labels" or "open properties" policy inside a graph type; openness instead means that a graph has no constraining graph type.

Graph entries. A catalog graph entry abstracts the ISO graph descriptor as

$$\mathbf{GraphEntry} = \langle G,\ \mathsf{GT}^{?},\ \mathcal{I}_G \rangle,$$

where G is the instance graph, $\mathsf{GT}^{?}$ is either a globally resolved graph type reference or none (no constraining graph type), and $\mathcal{I}_G$ maps names to constraints defined directly on G. If $\mathsf{GT}^{?} = \mathsf{GT}$, constraint names are unique independently of their origin, so

$$\mathrm{dom}(\mathcal{I}_G) \cap \mathrm{dom}(\mathcal{I}_{\mathsf{GT}}) = \varnothing.$$

For the library graph created in Section 4.2 and populated in Section 4.3, let G_{lib} denote the resulting graph instance and let $\mathsf{GT}_{\mathsf{lib}}$ be the globally resolved graph type reference denoted by `libraryGraphType`. Since the example declares no constraints directly on the graph, its catalog entry is

$$\mathbf{GraphEntry}_{\mathsf{lib}} = \langle G_{\mathsf{lib}},\ \mathsf{GT}_{\mathsf{lib}},\ \varnothing \rangle\,.$$

The empty third component means only that the graph has no directly defined constraints. The key constraints contained in the resolved `libraryGraphType` still apply through the constraining graph type reference in the second component.

Catalog location. Cataloged graph types and cataloged graphs are primary objects held by a GQL-schema in the single catalog $\mathcal{C}$ of Chapter 17. Inline graph types and unnamed graphs can also be primary objects without being catalog objects. Node and edge type descriptors are secondary objects contained in a graph type; they are not parallel top-level catalog dictionaries.

Resolution. Name resolution is modeled by partial functions that consult $\mathcal{C}$. For graph types, write

$$\mathsf{resolve}_{\mathbf{GraphType}}(\mathcal{C}, \mathsf{GT}) = \langle \Sigma_{\mathsf{GT}}, \mathcal{I}_{\mathsf{GT}} \rangle$$

when the graph type reference identifies that descriptor in its parent GQL-schema.

Graph-entry conformance. Write $G \models_{\text{str}} \Sigma$ when every node and edge of G satisfies the applicable label, property, directionality, and endpoint requirements encoded by the node and edge type declarations in Σ. For a set $\mathcal{J} \subseteq \mathbf{CDesc}$, write $G \models_{\text{con}} \mathcal{J}$ when G satisfies every key or unique constraint descriptor in $\mathcal{J}$, with satisfaction as specified in Section 16.1. For $\mathsf{GE} = \langle G, \mathsf{GT}^?, \mathcal{I}_G \rangle \in \mathbf{GraphEntry}$, let

$$\mathcal{I}_{\text{direct}}(\mathsf{GE}) = \text{ran}(\mathcal{I}_G)$$

be the constraints defined directly on the graph. For bookkeeping, the set of all constraints registered for the graph is

$$\mathcal{I}_{\text{active}}(\mathsf{GE}) = \text{ran}(\mathcal{I}_G) \cup \begin{cases} \text{ran}(\mathcal{I}_{\mathsf{GT}}), & \mathsf{GT}^? = \mathsf{GT}, \\ \varnothing, & \mathsf{GT}^? = \mathsf{none}. \end{cases}$$

but dynamic validation retains the two origins. The constraints in $\mathcal{I}_{\text{direct}}(\mathsf{GE})$ are validated directly. If $\mathsf{GT}^? = \mathsf{GT}$, resolution must succeed and the separate test that G is of its constraining graph type requires

$$G \models_{\text{str}} \Sigma_{\mathsf{GT}} \ \wedge \ G \models_{\text{con}} \text{ran}(\mathcal{I}_{\mathsf{GT}}).$$

Every graph must additionally satisfy $G \models_{\text{con}} \mathcal{I}_{\text{direct}}(\mathsf{GE})$. A graph with a constraining graph type is *closed*; a graph with $\mathsf{GT}^? = \mathsf{none}$ is *open*. Constraints apply only to their registered subjects, so constraints collected for a different graph entry or graph type are never included in $\mathcal{I}_{\text{active}}(\mathsf{GE})$.

15.6 DDL Semantics

General form. A DDL statement D denotes a partial function

$$[\![D]\!] : \mathsf{Catalog} \rightharpoonup \mathsf{Catalog}$$

for the catalog-modifying statements modeled here. A successful result must be a well-formed hierarchical catalog. In particular, every surviving **GraphEntry** must still resolve its optional constraining graph type and satisfy its active graph-type and graph constraint dictionaries at the required validation point.

Node and edge type definitions. This edition of GQL has no standalone `CREATE NODE TYPE` or `CREATE EDGE TYPE` catalog-modifying statement. Node and edge type specifications are secondary objects created as body items of a graph type.

Creating graph types. The concrete syntax of `CREATE GRAPH TYPE` statements is governed by the core GQL grammar (where graph types are catalog objects whose bodies enumerate node types, edge types, and optional schema-level constraints). The following abbreviated BNF preserves the alternatives relevant here; the node- and edge-type nonterminals are the pattern/phrase forms of the core grammar:

```
create-graph-type-stmt ::=
  "CREATE" ( [ "PROPERTY" ] "GRAPH" "TYPE" [ "IF" "NOT" "EXISTS" ]
             | "OR" "REPLACE" [ "PROPERTY" ] "GRAPH" "TYPE" )
  catalog-graph-type-parent-and-name graph-type-source

graph-type-source ::= [ "AS" ] nested-graph-type-spec
                    | [ "AS" ] "COPY" "OF" graph-type-reference
                    | "LIKE" graph-expression

nested-graph-type-spec ::= "{" graph-type-body-item
                               { "," graph-type-body-item } "}"

graph-type-body-item ::= node-type-spec
                       | edge-type-spec
                       | "CONSTRAINT" constraint-spec

node-type-spec ::= node-type-pattern | node-type-phrase
edge-type-spec ::= edge-type-pattern | edge-type-phrase

property-types-spec ::= "{" [ property-type
                                  { "," property-type } ] "}"

property-type ::= property-name [ ( "::" | "TYPED" ) ] property-value-type
```

Semantically, a `create-graph-type-stmt` introduces a named graph type whose node- and edge-type items populate NDecls and EDecls of a tuple Σ as defined above, while each `property-type` contributes an entry to the corresponding property signature ptype_T. Each `CONSTRAINT` item populates $\mathcal{I}_{\mathsf{GT}}$ with a named key or unique constraint. The resulting graph type may be used as a constraining type in catalog graph entries as described in Section 15.5. A complete `CREATE GRAPH TYPE` statement using node types, edge types, property signatures, and key constraints appears in Section 4.2.

Alteration and conversion. The ISO standard defines creation, replacement, and dropping of graph types, but no `ALTER TYPE`, `ALTER PROPERTY`, `ADD KEY`, or DDL `USING` conversion clause. Such schema/data migration is therefore outside this core semantics. Whenever a value is assigned or explicitly cast by supported GQL syntax, the cast and store- assignment rules apply and may raise an exception; no type-set inclusion makes that conversion total by itself.

Dropping declarations. The core drop operation here is `DROP [PROPERTY] GRAPH TYPE [IF EXISTS] q`. It is permitted only when no existing graph in the catalog references that graph type. On success the descriptor is removed from its parent schema and destroyed. Consequently the operation neither silently changes the property type of existing data nor requires an invented property-level cast. `OR REPLACE` first performs this same drop and is subject to the same reference condition.

15.7 Schema Versioning

Scope. Let Sch be the current schema state extracted from the catalog and Sch' a schema state produced by a supported evolution step. ISO 39075 leaves it implementation-defined which catalog-modifying procedures are under transaction control and uses a feature to govern whether data modification and catalog modification may appear in the same transaction. Where an implementation supports transactional DDL and such mixing, we model schema evolution as producing a new schema state Sch' from Sch.

Schema states and versions. DDL commands produce a chain

$$\mathsf{Sch}^{(0)} \xrightarrow{\mathrm{DDL}_1} \mathsf{Sch}^{(1)} \xrightarrow{\mathrm{DDL}_2} \cdots \xrightarrow{\mathrm{DDL}_n} \mathsf{Sch}^{(n)}.$$

Whether these catalog changes participate in a transaction and any physical reorganization are implementation-defined. The standard's statement-specific validation and execution rules remain controlling; the formal development assumes a conforming catalog state after each successful committed step.

Chapter 15 — Takeaways

- **Purpose:** Introduced the static typing layer and schema objects that constrained property graphs beyond the raw data model.
- **Introduced:** Cataloged graph types containing node and edge types as secondary objects; property-type declarations; schema evolution behavior and how DDL affected the catalog state.
- **Key distinctions:** Graph conformance to a declared type versus mere existence of a named type in the catalog; schema evolution versus ordinary data updates.
- **Consequences:** Enabled static checking of queries and updates against declared types; assigned a precise semantic meaning to DDL operations; prepared the ground for integrity constraints defined over typed populations.
- **Bridge:** Set up integrity constraints as the next layer that enforced correctness beyond typing.

16 Constraints

This chapter formalizes integrity constraints for property graphs in GQL and explains how they interact with the underlying typing discipline. We begin with key and uniqueness constraints based on constraint graph patterns, and then describe their enforcement during updates and transactions. Finally, we connect these dynamic checks to a lightweight effect system, which approximates write footprints and write conflicts without claiming to determine evaluation-order or context dependence, or result determinism.

16.1 Keys and Uniqueness

Recall from Section 15.3 that a key label set identifies a keyed node or edge type in a graph type and is not itself an integrity constraint.

Remark (scope as a graph pattern). The scope of a constraint is a deterministic constraint graph pattern P introduced by `FOR`. Its main element variable identifies the constrained node or edge; an edge scope also declares its endpoint variables.

Key specifications. A key or unique specification is a tuple

$$\chi = \langle P, \bar{c} \rangle,$$

where $\bar{c} = \langle c_1, \ldots, c_n \rangle$ is a sequence of $n \geq 0$ key value components. A component is the main element variable, an endpoint variable of an edge scope, or a property reference rooted at one of those variables. For a binding b matched by P, let

$$\mathbf{t}_\chi(b) = \langle [\![c_1]\!]_b, \ldots, [\![c_n]\!]_b \rangle.$$

Property references use the expression semantics of GQL: an absent property yields null when the declared result type is nullable, and otherwise raises the prescribed data exception. Thus absent never becomes a key value component.

Non-distinctness (for `KEY` and `UNIQUE`). Write $a \approx b$ when a and b are not distinct under GQL's recursive value-comparison rules, and lift this relation pointwise to key tuples:

$$\langle a_i \rangle_i \approx \langle b_i \rangle_i \quad \text{iff} \quad \forall i.\ a_i \approx b_i.$$

In particular, two null values are non-distinct; the rule also applies recursively to composite supported property values.

Constraint-scope notation. For a deterministic constraint graph pattern P, let $\mathsf{match}_G(P)$ be the set of bindings in its match result over G, with multiplicities ignored. For $b \in \mathsf{match}_G(P)$, $\mathsf{main}(b)$ is the graph element bound to the main element variable of P. The predicate $\mathsf{nullContaining}(v)$ holds exactly when v is the null value or is a composite value having a recursively null-containing component.

Key constraint (`KEY`). A graph G satisfies KEY(χ) iff every key tuple is not null-containing and any two constrained elements with non-distinct key tuples are identical:

$$\forall b \in \mathsf{match}_G(P).\ \neg\mathsf{nullContaining}(\mathbf{t}_\chi(b)),$$
$$\forall b_1, b_2 \in \mathsf{match}_G(P).\ \mathbf{t}_\chi(b_1) \approx \mathbf{t}_\chi(b_2) \Rightarrow \mathsf{main}(b_1) = \mathsf{main}(b_2).$$

Null-containment is recursive, so a composite component containing the null value also violates the key constraint.

Uniqueness constraint (`UNIQUE`). A graph G satisfies UNIQUE(χ) iff any two constrained elements with non-distinct key tuples are identical:

$$\forall b_1, b_2 \in \mathsf{match}_G(P).\ \mathbf{t}_\chi(b_1) \approx \mathbf{t}_\chi(b_2) \Rightarrow \mathsf{main}(b_1) = \mathsf{main}(b_2).$$

No separate null-containment test is performed for UNIQUE. When $n = 0$, all empty key tuples compare alike, so either form permits at most one constrained element.

Example: `BookHasKey`. The `BookHasKey` declaration in Section 4.2 has the key specification

$$P_{\texttt{Book}} = \texttt{(b:Book)}, \qquad \overline{c}_{\texttt{Book}} = \langle \texttt{b.isbn} \rangle, \qquad \chi_{\texttt{Book}} = \langle P_{\texttt{Book}}, \overline{c}_{\texttt{Book}} \rangle.$$

For the bindings β_1 and β_2 produced by this constraint pattern for the two book nodes in the library example,

$$\mathbf{t}_{\chi_{\texttt{Book}}}(\beta_1) = \langle \texttt{'978-83-123...'} \rangle,$$
$$\mathbf{t}_{\chi_{\texttt{Book}}}(\beta_2) = \langle \texttt{'978-83-987...'} \rangle.$$

Both tuples are non-null-containing and they are unequal, so the two book nodes satisfy the `KEY` requirement. If two bindings for distinct book nodes instead produced the same non-null ISBN tuple, then $\mathsf{main}(\beta_1) \neq \mathsf{main}(\beta_2)$ would violate KEY($\chi_{\texttt{Book}}$). Under a hypothetical `UNIQUE` declaration with the same specification, equal ISBN tuples would be non-distinct and would likewise require the main elements to be identical; unlike `KEY`, `UNIQUE` performs no separate null-containment test.

General constraint specifications. Let **CDesc** be the disjoint union of the carriers of key-constraint and unique-constraint descriptors. Every constraint defined by this edition is of exactly one of those two kinds. Its descriptor contains its name, registered-subject set, deterministic scope, and key value component specification. Constraints in a graph type apply to all graphs registered as subjects of that graph type; constraints defined in a graph apply only to that graph. Concrete examples are shown in Section 4.2.

For each constraint body C, let $\mathsf{CBNF}(C)$ be the implementation-defined, deterministic and idempotent normal form representing its equivalence class. Two bodies are *body-equivalent* exactly when their normal forms are equal; equivalence accounts for both equivalent constraint scopes and equivalent key value component specifications. The analogous normal form CSNF for a constraint specification also retains its constraint name. These normal forms determine the scope and requirement information represented in constraint descriptors.

Syntactic form of constraint specifications. Constraint specifications occur as `CONSTRAINT` body items of a nested graph type specification and in standalone constraint-management statements:

```
graph-type-body-item ::= ... | "CONSTRAINT" constraint-spec
constraint-spec ::= constraint-name constraint-body
constraint-body ::= "FOR" constraint-graph-pattern
                    "REQUIRE" constraint-requirement
constraint-requirement ::= key-value-spec "IS" "KEY"
                         | key-value-spec "IS" "UNIQUE"
key-value-spec ::= key-value-component
                 | "(" [ key-value-component-list ] ")"
key-value-component-list ::= key-value-component
                             { "," key-value-component }
key-value-component ::= element-variable [ "." property-name ]

create-constraint-stmt ::=
  "CREATE" ( "CONSTRAINT" [ "IF" "NOT" "EXISTS" ]
           | "OR" "REPLACE" "CONSTRAINT" )
  [ constraint-parent-graph-spec ] constraint-spec

drop-constraint-stmt ::= "DROP" "CONSTRAINT" [ "IF" "EXISTS" ]
                         [ constraint-parent-graph-spec ] constraint-name

constraint-parent-graph-spec ::= graph-expression "."
```

If the parent graph specification is omitted, `CURRENT_GRAPH` is implicit. Let $\mathcal{I}_G$ be the selected graph's direct constraint dictionary and, when it has a constraining graph type,

let $\mathcal{I}_{\mathsf{GT}}$ be that type's constraint dictionary. Constraint names are unique independently of origin, so their domains are disjoint. A plain `CREATE CONSTRAINT` using a name already in $\mathcal{I}_G$ raises G1004; without `IF NOT EXISTS`, a name already in $\mathcal{I}_{\mathsf{GT}}$ is not permitted. If the new body is body-equivalent to a constraint in either dictionary and `IF NOT EXISTS` is absent, creation raises G1006. With `IF NOT EXISTS`, an existing name or a body-equivalent constraint in either dictionary makes the statement a no-op. `OR REPLACE` can replace only a same-named constraint in $\mathcal{I}_G$; it cannot replace an inherited constraint, and the body-equivalence rule still applies. Otherwise, creation validates the constraint against the selected graph before adding it to $\mathcal{I}_G$; dropping removes a direct constraint from that dictionary. An inherited constraint cannot be dropped in this way.

16.2 Constraint Enforcement and Effects

Graph-type conformance Σ_{typing}. For an optional constraining graph-type reference, define $\Sigma_{\text{typing}}(\mathsf{GT}^?)$ as follows. When $\mathsf{GT}^? = \mathsf{GT}$, it is the clause requiring every node and edge to be of a node or edge type specified by the resolved graph type, including exact label/property sets, property value types, directionality, and endpoint node types, and requiring every constraint of that graph type to be satisfied. Failure means that the graph is no longer of its constraining graph type and is reported as graph type violation G2000. For $\mathsf{GT}^? = \mathsf{none}$ the component is empty.

Direct graph constraints and graph-type conformance. Let $\mathsf{GE} = \langle G, \mathsf{GT}^?, \mathcal{I}_G \rangle$ be the active graph entry. Its direct constraint set is

$$\mathcal{I}_{\text{direct}}(\mathsf{GE}) = \text{ran}(\mathcal{I}_G).$$

The derived set $\mathcal{I}_{\text{active}}(\mathsf{GE})$ from Section 15.5 remains useful when referring collectively to every constraint registered for the graph, but validation retains the two origins: graph-type constraints are part of the graph-type-conformance test, while $\mathcal{I}_{\text{direct}}$ is validated directly. Define

$$[\![\mathsf{GE}]\!] = \Sigma_{\text{typing}}(\mathsf{GT}^?) \cup \mathcal{I}_{\text{direct}}(\mathsf{GE}).$$

The first component is empty for an open graph and otherwise states that the graph is of its constraining graph type, including satisfaction of that graph type's constraints. The second contains the key or unique constraints defined directly on this graph. No declaration from an unrelated graph entry or graph type is consulted.

Satisfaction of a constraint set. Each semantic clause $\sigma \in [\![\mathsf{GE}]\!]$ has a satisfaction relation $G \models \sigma$. For any set of clauses S, write

$$G \models S \quad \text{iff} \quad \forall \sigma \in S.\ G \models \sigma.$$

This is the meaning of the satisfaction test used by the enforcement below. The effect of these checks on insertion, update, and deletion is illustrated in Section 4.3.

Key and uniqueness constraints. Every descriptor in $\mathcal{I}_{\text{active}}(\mathsf{GE})$ contributes exactly the KEY(χ) or UNIQUE(χ) clause described in Section 16.1. This uses the descriptor's deterministic constraint graph pattern as scope and retains its complete key value component sequence; it is not reduced to a global label/property dictionary.

Validation predicate. For an active graph entry GE with pre-state graph G and a coalesced update delta Δ, define

$$\mathsf{ok}_{\mathsf{GE}}(G, \Delta) \quad \text{iff} \quad G \oplus \Delta \text{ is defined and } (G \oplus \Delta) \models [\![\mathsf{GE}]\!].$$

Constraint enforcement. After a primitive data-modifying statement has produced its provisional graph, every constraint in $\mathcal{I}_{\text{direct}}(\mathsf{GE})$ is dynamically validated by that statement's General Rules; failure raises *integrity constraint violation* (23000). The separate graph-type-conformance test also checks the structural requirements and every constraint of the resolved graph type; failure raises *graph type violation* G2000. The statement cannot complete successfully while either validation fails.

16.3 Effect System and Write-Purity

The following informative effect system is defined only for query statements and data-modifying statements. Catalog-modifying statements are outside its domain and retain their statement-specific rules.

Here Σ is the applicable static graph-type descriptor and $\mathcal{T}$ is a finite typing environment for binding variables. For $X \in \{\mathbf{V}, \mathbf{E}\}$, let $\mathcal{T}_X$ be the restriction of $\mathcal{T}$ to variables whose declared graph-element reference type has kind X; membership of a variable in $\mathcal{T}_X$ abbreviates membership in the domain of that restricted environment. The carrier of static effects is written $\mathfrak{E}$, so it is distinct from the typing environment. Effects form a commutative idempotent monoid $(\mathfrak{E}, \sqcup_{\text{eff}}, \varnothing)$, where $\sqcup_{\text{eff}}$ combines effects and $\varnothing$ denotes the absence of effects.

Let P range over read parts, S over query or data-modifying statements, a over primitive update-action templates, U over finite write parts composed of such templates, $\mathcal{T}'$ over output typing environments, and $\varepsilon_{\text{eff}} \in \mathfrak{E}$. The rules below use two auxiliary static judgments:

$$\Sigma; \mathcal{T} \vdash P \Rightarrow \mathcal{T}' \qquad \text{and} \qquad \Sigma; \mathcal{T} \vdash a \ \mathbf{ok}.$$

The first holds when the read part P is well typed under Σ and $\mathcal{T}$, producing the extended environment $\mathcal{T}'$. The second holds when every target and embedded expression of the

action template a is well typed in the given environment. For a write part U, $\mathrm{fv}(U)$ is the set of binding variables occurring free in its action templates.

We use the following effect-oriented judgments:

$$\Sigma; \mathcal{T} \vdash U : \varepsilon_{\text{eff}} \qquad \text{(write-part templates have static effect } \varepsilon_{\text{eff}})$$

$$\Sigma; \mathcal{T} \vdash S : \varepsilon_{\text{eff}} \qquad \text{(query or data-modifying statements have static effect } \varepsilon_{\text{eff}})$$

The *static effect* ε_{eff} is a conservative description, derived without executing the statement, of the kinds of state changes that the evaluated construct may perform. A statement is *write-pure* in this effect model iff its static effect is empty, that is, iff $\varepsilon_{\text{eff}} = \varnothing$. This says only that the statement has no modeled graph-data write. It does not imply that evaluated expressions are deterministic, exception-free, independent of evaluation order, or independent of observable context.

Primitive update actions (meta-language). The operational semantics of DML (cf. Chapter 14) produces a *coalesced update delta* Δ, understood as a finite sequence of *primitive actions* over the store. We write $a \in \Delta$ for such actions and use the following representative constructors: Here $i, u, v \in V$ are node identities, $j \in E$ is an edge identity, $k \in \mathbb{K}$ is a property key, $w \in \mathbb{V}$ is a value, $\ell \in \mathbb{L}$ is a label, d is directedness, W is an initializer map from property keys to concrete values, and Λ_V, Λ_E are the node- and edge-label sets assigned at creation time.

$$\begin{aligned} a \;::=\; & \mathsf{insNode}(i, \Lambda_V, W) \mid \mathsf{insEdge}(j, \Lambda_E, W, u, v, d) \\ & \mid \mathsf{setPropV}(i, k, w) \mid \mathsf{setPropE}(j, k, w) \\ & \mid \mathsf{clearPropsV}(i) \mid \mathsf{clearPropsE}(j) \mid \mathsf{delPropV}(i, k) \mid \mathsf{delPropE}(j, k) \\ & \mid \mathsf{addLabelV}(i, \ell) \mid \mathsf{delLabelV}(i, \ell) \mid \mathsf{addLabelE}(j, \ell) \mid \mathsf{delLabelE}(j, \ell) \\ & \mid \mathsf{delNode}(i) \mid \mathsf{delEdge}(j). \end{aligned}$$

These constructors are *not* user syntax; they are meta-level names for the store-level steps that realize the write part U. Where we use schematic typing-level actions such as $\mathbf{Create}\langle\cdot\rangle$, they are understood as abstracting the same runtime updates that are ultimately lowered to the corresponding insNode or insEdge actions.

Primitive effects and algebra. For static analysis, each primitive update action is abstracted by a *primitive effect* that records its write footprint while omitting runtime-specific information such as element identities, endpoint identities, and computed property values. We model these effects by a small set of constructors used in the meta-language.

Parameters range over vertex-label sets $\Lambda_V \subseteq \mathbb{L}$, edge-label sets $\Lambda_E \subseteq \mathbb{L}$, property keys $k \in \mathbb{K}$, element kinds $X \in \{\mathbf{V}, \mathbf{E}\}$, and labels $\ell \in \mathbb{L}$. The symbol $\top_\Lambda$ denotes an implementation-agnostic "unknown or unconstrained" label-set abstraction.

$$\begin{aligned} e_{\text{eff}} \;::=\; & \mathbf{CreateV}[\Lambda_V] \mid \mathbf{DeleteV}[\Lambda_V] \mid \mathbf{CreateE}[\Lambda_E] \mid \mathbf{DeleteE}[\Lambda_E] \\ & \mid \mathbf{Set}(X, k) \mid \mathbf{ClearProps}(X) \mid \mathbf{RemProp}(X, k) \\ & \mid \mathbf{AddLbl}(X, \ell) \mid \mathbf{RemLbl}(X, \ell) \quad (X \in \{\mathbf{V}, \mathbf{E}\}). \end{aligned}$$

The effect combination introduced above is commutative, so the order in which effects are combined is irrelevant, while idempotence means that recording the same effect more than once does not change the combined effect.

We write $e_{\text{eff},1} \leq_{\text{eff}} e_{\text{eff},2}$ when $e_{\text{eff},2}$ is a less specific abstraction of $e_{\text{eff},1}$; for example,

$$\mathbf{CreateV}[\Lambda_V] \leq_{\text{eff}} \mathbf{CreateV}[\top_\Lambda].$$

The join operator is monotone with respect to this order. In particular, for every finite family of effects, if $e_{\text{eff},i} \leq_{\text{eff}} e'_{\text{eff},i}$ for each $i \in \{1, \ldots, n\}$, then

$$\bigsqcup_{\substack{\text{eff} \\ i=1}}^{n} e_{\text{eff},i} \leq_{\text{eff}} \bigsqcup_{\substack{\text{eff} \\ i=1}}^{n} e'_{\text{eff},i}.$$

Thus, replacing any component effect by a less specific abstraction yields a correspondingly less specific abstraction of the combined effect.

Effect of actions (representative). The abstraction function peff maps primitive runtime actions to their corresponding primitive static effects:

$$\begin{aligned} \mathsf{peff}(\mathsf{insNode}(_, \Lambda_V, _)) &= \mathbf{CreateV}[\Lambda_V], \\ \mathsf{peff}(\mathsf{insEdge}(_, \Lambda_E, _, _, _, _)) &= \mathbf{CreateE}[\Lambda_E], \\ \mathsf{peff}(\mathsf{setPropV}(i, k, w)) &= \mathbf{Set}(\mathbf{V}, k), \\ \mathsf{peff}(\mathsf{setPropE}(j, k, w)) &= \mathbf{Set}(\mathbf{E}, k), \\ \mathsf{peff}(\mathsf{clearPropsV}(i)) &= \mathbf{ClearProps}(\mathbf{V}), \\ \mathsf{peff}(\mathsf{clearPropsE}(j)) &= \mathbf{ClearProps}(\mathbf{E}), \\ \mathsf{peff}(\mathsf{delPropV}(i, k)) &= \mathbf{RemProp}(\mathbf{V}, k), \\ \mathsf{peff}(\mathsf{delPropE}(j, k)) &= \mathbf{RemProp}(\mathbf{E}, k), \\ \mathsf{peff}(\mathsf{addLabelV}(i, \ell)) &= \mathbf{AddLbl}(\mathbf{V}, \ell), \\ \mathsf{peff}(\mathsf{delLabelV}(i, \ell)) &= \mathbf{RemLbl}(\mathbf{V}, \ell), \\ \mathsf{peff}(\mathsf{addLabelE}(j, \ell)) &= \mathbf{AddLbl}(\mathbf{E}, \ell), \end{aligned}$$

$$\begin{aligned}\mathsf{peff}(\mathsf{delLabelE}(j,\ell)) &= \mathbf{RemLbl}(\mathbf{E},\ell),\\ \mathsf{peff}(\mathsf{delEdge}(j)) &= \mathbf{DeleteE}[\Lambda_E] \quad (\Lambda_E \text{ from } j\text{'s binding}),\\ \mathsf{peff}(\mathsf{delNode}(i)) &= \mathbf{DeleteV}[\Lambda_V] \quad (\Lambda_V \text{ from } i\text{'s binding}).\end{aligned}$$

Template sets. Let U be a finite, textually ordered list of action templates.

$$\frac{\forall a \in U.\ \Sigma;\mathcal{T} \vdash a\ \mathbf{ok} \qquad \varepsilon_{\text{eff}} = \bigsqcup_{\text{eff}\ a\in U} \mathsf{peff}(a)}{\Sigma;\mathcal{T} \vdash U : \varepsilon_{\text{eff}}}\ \textsc{WritePart}$$

Here $\bigsqcup_{\text{eff}}$ is the finite join in the effect algebra defined above.

Statement typing with effects. Let a statement be $S \equiv P \triangleright U$, where P is the read part, such as a pattern and its filters, and U is the write part.

$$\frac{\Sigma;\mathcal{T} \vdash P \Rightarrow \mathcal{T}' \quad \Sigma;\mathcal{T}' \vdash U : \varepsilon_{\text{eff}}}{\Sigma;\mathcal{T} \vdash (P \triangleright U) : \varepsilon_{\text{eff}}}\ \textsc{Stmt}$$

Hence, query-only statements, which have no write part, satisfy $\varepsilon_{\text{eff}} = \varnothing$ and are write-pure in this limited sense.

No unbound writers. A write part may refer only to variables bound by its preceding read part:

$$\frac{\Sigma;\mathcal{T} \vdash P \Rightarrow \mathcal{T}' \quad \mathrm{fv}(U) \subseteq \mathrm{dom}(\mathcal{T}') \quad \Sigma;\mathcal{T}' \vdash U : \varepsilon_{\text{eff}}}{\Sigma;\mathcal{T} \vdash P \triangleright U : \varepsilon_{\text{eff}}}\ \textsc{NoUnbound}$$

Meta-level single-writer approximation (informative). For optimizer analysis only, call a statement *stat-det* with respect to writes if, for each entity variable x and property key k, at most one setProp$*(x,k,\cdot)$ template remains after static simplification of U. This is neither GQL syntax nor a conformance condition; the concrete data-modifying statements retain their own static and dynamic rules for repeated target properties.

Static conflict approximation and ordering constraints Let MayConflict be a symmetric predicate on pairs of effects. It holds for $\varepsilon_{\text{eff},1}$ and $\varepsilon_{\text{eff},2}$ if

$$\varepsilon_{\text{eff},1} \sqcup_{\text{eff}} \varepsilon_{\text{eff},2}$$

contains, up to $\leq_{\text{eff}}$, a potentially conflicting pair of the primitive effect constructors defined above, possibly on aliased targets. A conservative alias analysis may mark bindings that can denote the same runtime element. The predicate MayConflict is only a meta-level optimizer aid: it may prevent a reordering whose equivalence has not otherwise been

established, but it introduces no GQL isolation-barrier annotation and no additional rejection rule.

CREATE (schematic). In the schematic rules below, storeAssignableTo abbreviates the applicable GQL store- assignment check. Predicates $\mathsf{closed}(x)$ and $\mathsf{open}(x)$ classify the declared graph-element reference type of x. A closed type supplies property signature Ψ; an open type admits any supported property value type. For vertex creation with initializer map P:

$$\frac{\begin{array}{l}\Sigma;\mathcal{T} \vdash (v : \Lambda_V) \Rightarrow \mathcal{T}, v : (\Lambda_V, \Psi) \\ \forall k \in \mathrm{dom}(P).\ \Sigma;\mathcal{T} \vdash P(k) : \sigma_k \ \wedge \\ \quad \Big[\mathsf{closed}(v) \wedge k \in \mathrm{dom}(\Psi) \wedge \sigma_k\ \mathsf{storeAssignableTo}\ \Psi(k) \\ \qquad \vee\ \mathsf{open}(v) \wedge \sigma_k \in \mathbb{T}_{\mathrm{prop}}\Big]\end{array}}{\begin{array}{c}\Sigma;\mathcal{T} \vdash \mathbf{Create}\langle v : \Lambda_V, P\rangle : \\ \mathbf{CreateV}[\Lambda_V]\ \sqcup_{\mathrm{eff}} \bigsqcup_{\substack{\mathrm{eff} \\ k \in \mathrm{dom}(P)}} \mathbf{Set}(\mathbf{V}, k)\end{array}}\ \textsc{Create-V}$$

SET/DELETE.

$$\frac{\begin{array}{l}x : (\Lambda, \Psi) \in \mathcal{T}_X\ (X \in \{\mathbf{V}, \mathbf{E}\}) \\ \Sigma;\mathcal{T} \vdash e : \sigma \\ \Big[\mathsf{closed}(x) \wedge k \in \mathrm{dom}(\Psi) \\ \qquad \wedge\, \sigma\ \mathsf{storeAssignableTo}\ \Psi(k)\Big]\ \vee\ \Big[\mathsf{open}(x) \wedge \sigma \in \mathbb{T}_{\mathrm{prop}}\Big]\end{array}}{\begin{array}{c}\Sigma;\mathcal{T} \vdash \mathbf{SET}(x.k := e) : \\ \mathbf{Set}(X, k)\end{array}}\ \textsc{Set}$$

$$\frac{e : (\Lambda_E, _) \in \mathcal{T}_{\mathbf{E}}}{\Sigma;\mathcal{T} \vdash \mathbf{DELETE}(e) : \mathbf{DeleteE}[\Lambda_E]}\ \textsc{Del-Edge}$$

$$\frac{v : (\Lambda_V, _) \in \mathcal{T}_{\mathbf{V}}}{\Sigma;\mathcal{T} \vdash \mathbf{DELETE}(v) : \mathbf{DeleteV}[\Lambda_V]}\ \textsc{Del-Vertex}$$

Null assignment and property removal. A `SET` property item performs store assignment to the property's declared type. The null value is therefore stored only when that type is nullable; otherwise it is not assignable and the prescribed error is raised. Assigning null is never lowered to property removal. Property removal is a separate data-modifying operation expressed by `REMOVE`.

Effect soundness (informative) Let $\varepsilon_{\mathrm{eff},S}$ be the static effect of S and let Δ be the coalesced sequence of primitive actions produced at runtime. Every primitive action in Δ

refines some $e_{\text{eff}} \in \varepsilon_{\text{eff},S}$ under $\leq_{\text{eff}}$. This establishes soundness only for the write-footprint abstraction. A rewrite cannot be justified from $\varepsilon_{\text{eff},S}$ alone: it must separately preserve evaluation order, exceptions, result bags, context-dependent expression behavior, and all other observable behavior.

Micro-derivation: write-purity of query-only statements (informative) If S has no write part (i.e. $S \equiv P$ with result production only), then $\Sigma; \mathcal{T} \vdash S : \varnothing$. Therefore S performs no writes described by this effect algebra. Reordering or common-subexpression elimination additionally requires the relevant expressions to be independent of the changed evaluation order and context, to preserve exception behavior, and to preserve the result bag.

Constraint-Preserving Execution Recall the graph-entry validation set $[\![\mathsf{GE}]\!]$ and validation predicate $\mathsf{ok}_{\mathsf{GE}}(G, \Delta)$ from the previous section. An execution of a data-modifying statement within a transaction, with pre-state G and coalesced delta Δ, is *constraint-preserving* if $\mathsf{ok}_{\mathsf{GE}}(G, \Delta)$ holds, that is, if the post-state $G \oplus \Delta$ satisfies its constraining graph type and all key/unique constraints defined directly on that graph. The operational semantics of DML rejects a structural graph-type violation or a violation of any direct graph constraint. Thus a primitive data-modifying statement completes successfully only if its post-state is constraint-preserving. Catalog-modifying statements are not modeled as $G \oplus \Delta$; each uses the checks specified by its own Syntax and General Rules.

Chapter 16 — Takeaways

- **Purpose:** Formalized integrity constraints and their interaction with typing and updates.
- **Introduced:** Key and uniqueness constraints over graph elements; enforcement rules and violation outcomes; an effect-based approximation of statement footprints to reason about potential conflicts.
- **Key distinctions:** Key constraints (identification) versus uniqueness constraints (non-duplication without identification semantics); static approximations of effects versus data-dependent, dynamic effects.
- **Consequences:** Determined when and how violations were detected and what observable outcomes followed; enabled reasoning about write footprints without conflating them with evaluation-order or context dependence; connected statement completion to dynamic constraint validation.
- **Bridge:** Required a precise catalog and session model so that names, schemas, and constraints could be resolved consistently at runtime.

Part VIII

Catalogs, Sessions, and Procedures

17 Catalog, Names and Session State

This chapter axiomatizes the abstract catalog that underlies GQL programs, explaining how named objects, schemas, and graphs are organized and resolved during query and update execution. We explicitly define the normative core catalog objects, specify qualified names and the rules for resolving them relative to a working schema and working graph, and describe the notion of a session, including its execution sites and policy settings. These ingredients provide the ambient context in which the semantic rules of earlier and later chapters are interpreted.

17.1 Named Objects and Namespaces

Catalog. A *GQL-catalog* is a persistent, hierarchically organized collection whose root is either a GQL-directory or a GQL-schema. Directories contain directories and schemas; schemas contain the normative primary catalog objects: graphs, graph types, binding tables, and named procedures. Node and edge types are secondary objects contained in graph types.

Identifiers and qualification. Let $\mathbb{R}_{\mathsf{cat}}$ denote the syntactically well-formed catalog object references of the GQL grammar, and let $\mathbb{Q}$ denote their globally resolved references to catalog objects. A syntactic reference consists of an optional catalog object parent reference and a local object name. Schema paths use "/" and may be absolute or relative (including `CURRENT_SCHEMA`, `HOME_SCHEMA`, ".", and ".."); periods separate any nested secondary-object names. Thus $\mathbb{R}_{\mathsf{cat}}$ is not modeled as dot-concatenated identifiers detached from the catalog hierarchy.

Catalog descriptors and derived projections. Abstractly, a catalog instance is a rooted tree

$$\mathcal{C} = \langle \mathsf{Root} \rangle,$$

where every directory descriptor maps local identifiers to directory or schema descriptors and every schema descriptor maps local identifiers to primary catalog object descriptors. A schema descriptor also records its owner, which is an authorization identifier; a schema created by `CREATE SCHEMA` has the session authorization identifier as its owner. Let **PDesc** denote the domain of named procedure descriptors; its tuple structure is defined

in Chapter 18. For concise semantic rules, the following are derived projections after standard reference resolution, not independent top-level dictionaries:

$$\begin{aligned} \mathsf{Graphs} &: \mathbb{Q} \rightharpoonup \mathbf{GraphEntry}, \\ \mathsf{GraphTypes} &: \mathbb{Q} \rightharpoonup \mathbf{GTypeDesc}, \\ \mathsf{BindingTables} &: \mathbb{Q} \rightharpoonup \mathsf{BT}, \\ \mathsf{Procs} &: \mathbb{Q} \rightharpoonup \mathbf{PDesc}. \end{aligned}$$

Here a member of BT is the binding-table object, including its record collection and associated descriptor: binding-table type, ordered/unordered indication, and preferred column-name sequence. The schema's single named-subobject dictionary ensures that two primary objects with the same local name cannot silently occupy separate sort-specific namespaces.

Active context and working schema/graph. An *active context* is

$$\mathcal{X} = \langle \mathcal{C},\ \mathsf{WSR},\ \mathsf{WG} \rangle,$$

where WSR (working schema reference) and WG (working graph site) determine relative resolution and execution scope as in the GQL standard object-reference and `USE` rules.

Working graph and unnamed patterns. The working schema component WSR is a working schema reference. The graph reference value occupying the working graph site WG identifies the graph against which unqualified pattern clauses and data-modification operations are evaluated. A `USE GE` graph-selection step evaluates `GE` and establishes the current working graph. Subsequent pattern clauses and data-modification operations use that current working graph; they do not carry a separate graph expression of their own.

Example: library catalog projections. Let

$$\mathbf{GTypeDesc}_{\mathsf{lib}} = \langle \Sigma_{\mathsf{lib}}, \mathcal{I}_{\mathsf{lib}} \rangle$$

represent the abstract graph type descriptor of `libraryGraphType` defined in Sections 15.4 and 15.5. Let $q_G, q_{\mathsf{GT}} \in \mathbb{Q}$ be the globally resolved references denoted in the current working schema by `libraryGraph` and `libraryGraphType`. A catalog containing those objects satisfies

$$\begin{aligned} \mathcal{C}_{\mathsf{lib}} &= \langle \mathsf{Root}_{\mathsf{lib}} \rangle, \\ \mathsf{Graphs}(q_G) &= \mathbf{GraphEntry}_{\mathsf{lib}}, \\ \mathsf{GraphTypes}(q_{\mathsf{GT}}) &= \mathbf{GTypeDesc}_{\mathsf{lib}}. \end{aligned}$$

The two maps are derived projections of $\mathcal{C}_{\mathsf{lib}}$, not additional components of its one-component tuple.

Example: active library context. In the active-context abstraction used here, let q_S be the current working schema reference and let the graph reference value denoted by q_G occupy $\mathsf{WG}_{\mathsf{lib}}$. Then

$$\mathcal{X}_{\mathsf{lib}} = \langle \mathcal{C}_{\mathsf{lib}}, q_S, \mathsf{WG}_{\mathsf{lib}} \rangle,$$

and an unqualified `MATCH` is evaluated against the graph referenced by $\mathsf{WG}_{\mathsf{lib}}$.

Name resolution. Given a syntactic reference $r \in \mathbb{R}_{\mathsf{cat}}$ of sort

$$S \in \{\mathbf{Graph}, \mathbf{GraphType}, \mathbf{BindingTable}, \mathbf{Proc}\},$$

$\mathsf{resolve}_S(\mathcal{X}, r)$ first resolves its explicit schema/object parent. If the parent is omitted, the grammar supplies the implicit parent "`./`", i.e. the current working schema. The local name must identify an object of sort S in that parent; on success, resolution determines its globally resolved reference in $\mathbb{Q}$. Absolute, relative, home, and current-schema references follow their distinct grammar rules, so resolution is not an ordered search between two textual names.

17.2 Qualified References and `USE`

Syntactic form of the `USE` graph clause. The standard use-graph clause declares a working graph and its syntactically defined scope:

```
use-graph-clause ::= "USE" graph-expr

graph-expr ::= ...  /* graph reference or expression */
```

Here `graph-expr` denotes an expression whose value is a graph reference. The clause has neither an intervening `GRAPH` keyword nor an `AS` alias. Uses in modifying and read-only statements appear in Sections 4.3 and 4.4. This clause requires optional Feature GQ01, *USE graph clause*.

Using a graph for a statement. Let GE be the graph expression in `USE` GE. It is evaluated in a new child execution context with a material unit incoming working table. The working graph site identified by the clause is set to the resulting graph reference value. Its scope is the enclosing focused linear query or focused linear data-modifying statement part determined by the grammar; there is no concrete `USE` GE `IN` S form.

17.3 Session State and Sites

A session context is represented here by the compact tuple

$$\mathsf{SCX} = \langle a, \pi, z, S^?, G^?, \mathsf{Params}, \mathsf{ListBeh}, T^?, R^?, f \rangle.$$

Here a is the session authorization identifier, π is the principal it identifies, z is the time-zone displacement, $S^?$ and $G^?$ are the session schema and session graph (each possibly not set), Params is the session-parameter dictionary, ListBeh records offset/ordinal indexing, negative-index, and out-of-range behavior, $T^?$ and $R^?$ are the current transaction and request context (each possibly not set), and f is the termination flag. The list-behavior policies affect list element references and indexing, not list unwinding. Session `SET` and `RESET` commands modify or restore the corresponding characteristics, while session close sets the termination flag.

GQL also operates with execution contexts and sites; assignments to sites (read-/write locations) respect declared types, including dynamic unions. Nullability and inference of site nullability are part of the type and assignment semantics. In general, a store assignment is modeled as a partial function from memory state to memory state, equipped with dynamic type checking and the possibility of raising exceptions.

Chapter 17 — Takeaways

- **Purpose:** Axiomatized the catalog and session context in which all earlier semantics were interpreted.
- **Introduced:** Qualified naming and name-resolution rules; cataloged objects such as graphs, schemas, and procedures; session state, including working graph/schema selection and related scope rules.
- **Key distinctions:** Qualified versus unqualified references and their resolution scope; global catalog state versus local session context.
- **Consequences:** Eliminated ambiguity about what object a name denoted at any point; enabled multi-graph workflows with explicit working-graph selection; ensured that procedures and updates inherited the correct ambient context.
- **Bridge:** Enabled reusable abstractions by defining how named procedures were cataloged and resolved.

18 Procedures

This chapter treats procedures as callable abstractions and formalizes named procedure descriptors, inline GQL procedure bodies, resolution, and calls. Functions are value-expression operators covered by the expression semantics; they are not primary catalog objects in the core catalog modeled here.

18.1 Procedure Signatures and Body Typing

Named procedure descriptors. For the lookup-oriented model used in this chapter, we use the following compact, lookup-augmented representation of a named procedure catalog entry. It adds the catalog object name p to the standard descriptor components and retains the components needed below:

$$\mathbf{PDesc} = \langle p,\ \overline{x_j : \tau_j},\ \overline{x'_m : \tau'_m = d_m},\ \tau_{\text{res}},\ \mathsf{kind} \rangle,$$

where p is its catalog object name, $\overline{x_j : \tau_j}$ is the ordered list of mandatory parameters, $\overline{x'_m : \tau'_m = d_m}$ is the trailing list of optional parameters with default values, τ_{res} is the procedure result type, and $\mathsf{kind} \in \{\mathsf{catalog}, \mathsf{data}, \mathsf{query}\}$ records the permitted kind of side effects. Parameter names are unique. Optional parameters form only a suffix; supplying one requires supplying every optional parameter before it.

The *core signature* of the declaration is

$$\mathsf{sig}(\mathbf{PDesc}) \ = \ p(\overline{x_j : \tau_j}, \overline{x'_m : \tau'_m = d_m}) \longrightarrow \tau_{\text{res}}.$$

If τ_{res} is a binding-table type, its record type supplies the names and types available to a `YIELD` clause. Write $\mathbf{BT}\langle \overline{z_i : v_i} \rangle$ for the material binding-table type whose record has the named fields $\overline{z_i : v_i}$. In a `CALL`-statement, the declared type of the procedure call is required to be either a material binding-table type or the empty type. The effect abstraction used later is obtained from kind: a query procedure has no side effects, a data procedure may have data-modifying effects only, and a catalog procedure may have catalog-modifying effects, possibly together with data- modifying effects.

In the catalog, procedure names are unique and procedure overloading is not modeled: the `Procs` component maps each qualified procedure identifier to at most one declaration,

$$\mathsf{Procs} : \mathbb{Q} \rightharpoonup \mathbf{PDesc}.$$

Consequently, a qualified procedure name uniquely determines its core signature whenever it is present in the catalog.

Procedure bodies and their scope. In this edition, a named procedure is an external procedure; the standard does not specify a GQL statement that defines its implementation. Accordingly, **PDesc** contains no GQL body. An *inline* procedure is instead a nested procedure specification with a GQL procedure body. Its statements determine whether it is catalog-modifying, data-modifying, or a query procedure, and its declared result type is computed from that body. Body typing uses the incoming working record/table types and the statement-block rules; it is not an undeclared B_p attached to a named catalog descriptor.

18.2 Calls and Scoping

Static environment. Typing judgments use a resolution environment $\mathfrak{R} = \langle \mathcal{X}, \mathsf{Loc} \rangle$, where $\mathcal{X} = \langle \mathcal{C}, \mathsf{WSR}, \mathsf{WG} \rangle$ is the active context (catalog, working schema, working graph site) and Loc is a finite map of parameters and binding variables used when typing argument expressions. A catalog procedure reference is resolved by its own grammar and is not shadowed by a binding variable of the same spelling.

Resolution order. For a named procedure call, the procedure reference identifies its explicit catalog parent or uses the implicit parent `./` (the current working schema), then identifies a named procedure in that parent. Absolute and relative catalog paths follow Chapter 17. Alternatively, a procedure reference may be supplied by a substituted reference parameter; its substitution is resolved as the corresponding catalog procedure parent and name. This is reference resolution, not an ordered search through local variables and catalog names.

Syntactic form of procedure calls. At statement level, a procedure call may be optional and may invoke either an inline or a named procedure. The named form is:

```
call-proc-stmt ::= [ "OPTIONAL" ] "CALL" procedure-call
procedure-call ::= inline-procedure-call | named-procedure-call
named-procedure-call ::= procedure-reference "(" [ arg-list ] ")"
                         [ "YIELD" yield-list ]

arg-list   ::= expr { "," expr }

yield-list ::= yield-item { "," yield-item }

yield-item ::= field-name [ "AS" binding-variable ]
```

```
11
12 expr      ::= ...  /* value expression */
```

Here `procedure-reference` identifies a cataloged procedure, `arg-list` supplies actual arguments as value expressions in call-position order, and `yield-list` specifies which output columns of the procedure are made visible to the enclosing statement, with `AS` providing an explicit alias for the corresponding column name when required by the context. The named form requires optional Feature GP04, *Named procedure calls*. A graph argument additionally requires GP15, and a binding-table argument additionally requires GP14.

Example: a named library procedure. Let $p_{\mathsf{fees}} = \texttt{recomputeFees}$ be a procedure object name and let $q_{\mathsf{fees}} \in \mathbb{Q}$ be its globally resolved reference in the current working schema. In the descriptor abstraction used here, an illustrative external data procedure may be represented by

$$\mathbf{PDesc}_{\mathsf{fees}} = \langle p_{\mathsf{fees}}, \langle \texttt{asOf} : \mathsf{DATE} \rangle, \langle \rangle, \\ \mathbf{BT}\langle \texttt{updatedCount} : \mathsf{INTEGER} \rangle, \mathsf{data} \rangle .$$

If it is registered through an implementation-defined mechanism, the derived catalog projection satisfies

$$\mathsf{Procs}(q_{\mathsf{fees}}) = \mathbf{PDesc}_{\mathsf{fees}}.$$

The relative name in the following call resolves to q_{fees}:

```
1 CALL recomputeFees(DATE '2025-12-31')
2 YIELD updatedCount AS cnt
```

The procedure name and implementation are illustrative; the library DDL in Section 4.2 does not declare this external procedure.

Example: procedure-call resolution environment. In the static-resolution abstraction used here, one possible environment for a call typed inside the library context is

$$\mathsf{Loc}_{\mathsf{lib}} = \{\texttt{loanVar} \mapsto \mathsf{NodeRef}, \texttt{feeVar} \mapsto \mathsf{FLOAT}\}, \\ \mathfrak{R}_{\mathsf{lib}} = \langle \mathcal{X}_{\mathsf{lib}}, \mathsf{Loc}_{\mathsf{lib}} \rangle.$$

The local map records types used to type argument expressions; the reference `recomputeFees` is resolved through $\mathcal{X}_{\mathsf{lib}}$.

Procedure call in statements. A `CALL`-statement that invokes a named procedure and yields selected columns has the form $\textbf{CALL}\ p(\bar{e})\ \textbf{YIELD}\ \overline{z_k\ \textbf{AS}\ y_k}$. Let Ξ be the applicable static type context and let Θ be the incoming binding-name-to-type environment;

Θ' is its extension by yielded columns. For a descriptor with j mandatory and m optional parameters, define $n_{\min} = j$ and $n_{\max} = j + m$. For a binding-table result $\mathbf{BT}\langle\overline{z_i : v_i}\rangle$, let $v(z_i) = v_i$. Finally, $\varepsilon_{\mathsf{kind}}$ is empty for a query procedure and is the applicable abstract data- or catalog-modifying effect for the other procedure kinds. The yielding form below therefore assumes a material binding-table result. If the procedure result type is the empty type, `YIELD` is not permitted and the call statement uses the material unit record type for its output contribution. For a permitted number of actual arguments, write $\overline{e}$ $\mathsf{assignableTo}$ $(\overline{\tau}, \overline{\tau'})$ when the statically derived type of each supplied argument passes the GQL store-assignment check against the corresponding mandatory or optional parameter type; omitted optional arguments use their descriptor defaults. A schematic typing rule is:

$$\frac{\begin{array}{c}\mathsf{resolve}_{\mathbf{Proc}}(\mathcal{X}, p) = \langle p, \overline{x_j : \tau_j}, \overline{x'_m : \tau'_m = d_m}, \\ \mathbf{BT}\langle\overline{z_i : v_i}\rangle, \mathsf{kind}\rangle \\ n_{\min} \le |\overline{e}| \le n_{\max} \quad \overline{e}\ \mathsf{assignableTo}\ (\overline{\tau}, \overline{\tau'}) \\ \{z_k\}_k \subseteq \{z_i\}_i \quad \Theta' = \Theta \cup \{y_k : v(z_k)\}_k\end{array}}{\begin{array}{c}\Xi; \Theta; \mathfrak{R} \vdash \mathbf{CALL}\ p(\overline{e})\ \mathbf{YIELD}\ \overline{z_k\ \mathbf{AS}\ y_k} \\ \Rightarrow \Theta'; \varepsilon_{\mathsf{kind}}\end{array}}\ \textsc{CallProc}$$

Here $\mathsf{resolve}_{\mathbf{Proc}}$ is the procedure-specific instance of the catalog resolution function from Chapter 17. A `YIELD` clause is permitted only for a binding-table procedure result, selects existing result columns, and produces the aliases shown in Θ'. The incoming and yielded record types must be field-name-disjoint. The effect $\varepsilon_{\mathsf{kind}}$ is determined by the descriptor's procedure kind, and is empty exactly for a query procedure.

Operationally, the procedure is called once for each incoming working-table record in a child execution context containing its argument record. If its execution result is a binding table, each result record is combined with that incoming record. If its result is omitted (the empty-type case), the call statement uses a unit binding table containing one unit record, so the incoming record is preserved once. For `OPTIONAL CALL`, an empty binding-table result instead contributes one record whose yielded fields are null; a non-optional empty binding-table result contributes no record.

Chapter 18 — Takeaways

- **Purpose:** Defined procedures as cataloged abstractions with typed interfaces and well-scoped execution behavior.
- **Introduced:** Procedure signatures with typed parameters and typed output columns; call and yield mechanics; scoping rules that controlled variable visibility across calls.

- **Key distinctions:** Expression functions versus procedures; named external procedures versus inline GQL procedures; argument evaluation versus output shaping through explicit yield clauses.
- **Consequences:** Made modular query/update pipelines semantically well-defined; allowed static typing of calls and their outputs; clarified how working-graph context and local variables interacted with invoked code.
- **Bridge:** Required transactional semantics to define how effects composed and how failures interacted with atomicity.

Part IX

Transactions and Concurrency

19 Transaction Model

This chapter specifies how GQL groups procedure executions into transactions and how their combined effects are applied to the shared store state. We formalize commit and rollback behavior and the visibility of intermediate updates. The GQL core assumes serializable transactional behavior; less restrictive isolation levels and their snapshot rules are implementation-defined extensions. The treatment of errors and aborts is integrated with this model so that transactional guarantees are preserved during constraint enforcement and update execution.

19.1 Demarcation and Visibility

Transaction model. A transaction Tx is a tuple

$$\mathsf{Tx} = \langle u,\ \text{iso},\ \overline{P},\ \mathsf{SP} \rangle,$$

where u is the session authorization identifier under which the transaction is executed, iso is SERIALIZABLE in the core model or an implementation-defined isolation extension, $\overline{P} = \langle P_1, \ldots, P_m \rangle$ is a finite sequence of procedure executions performed in order, and SP is an optional meta-model component for savepoints. Savepoints are not part of the GQL core; an implementation extension that supplies them must also define their syntax and semantics. Let Ω, Ω' be abstract store states and κ an exception GQLSTATUS code as described in Chapter 21. Let the abstract transaction-outcome carrier contain the distinct constructors

$$\mathsf{Committed}(\Omega'), \qquad \mathsf{RolledBack}(\Omega), \qquad \mathsf{Aborted}(\kappa).$$

They denote, respectively, successful commitment, successful rollback, and a failed outcome accompanied by a rollback attempt. These constructors are distinct from the state-transition functions defined below.

Transactions and atomicity. A transaction is a delimited unit of work. A successful commit takes effect atomically with respect to recovery, while a successfully processed rollback has no lasting effect on GQL-data or the GQL-catalog. For a well-typed program B within a transaction, let $\Delta(B)$ denote the finite update sequence (delta) that its execution produces over Ω. More generally, for any procedure execution P within B, $\Delta(P)$

denotes the delta produced by that execution. For deltas, $\Delta_1 \diamond \Delta_2$ denotes their ordered composition, characterized whenever the applications are defined by

$$\Omega \oplus (\Delta_1 \diamond \Delta_2) = (\Omega \oplus \Delta_1) \oplus \Delta_2.$$

Thus $\diamond$ composes deltas, whereas $\oplus$ applies a delta to a store state.

$$\mathsf{commit} : (\Omega, \Delta(B)) \mapsto \Omega \oplus \Delta(B),$$
$$\mathsf{rollback} : (\Omega, \Delta(B)) \mapsto \Omega.$$

These equations describe successfully processed termination requests. At the committed-state boundary, all changes of a committing transaction take effect together and no changes of a successfully rolled-back transaction remain. If a transaction-termination request cannot be processed successfully, the state of the GQL-catalog and GQL-data becomes indeterminate; how the success or failure status is made available is implementation-defined. This recovery atomicity is distinct from isolation: an implementation-defined relaxation of serializable behavior may permit some provisional changes to be perceived by a concurrent transaction.

Statement sequencing. Within a transaction, statements execute in program order; effects of each completed statement are visible to subsequent statements of the same transaction prior to commit.

Procedure-kind and working-graph restrictions. Without optional Feature GP18, mixing a catalog-modifying procedure and a data-modifying procedure in one transaction raises 25G02. Without optional Feature GT03, accessing a graph different from one already accessed in the transaction raises 25G04.

Commands. A command is an operation not specified by a procedure written in GQL.

Syntactic form of transaction control commands. At the concrete-syntax level, GQL provides dedicated commands to start, commit, and roll back transactions, with a standardized access mode and optional implementation-defined transaction modes. In a simplified BNF these commands are written as:

```
start-transaction-cmd ::= "START" "TRANSACTION" [ tx-characteristics ]
tx-characteristics ::= tx-mode { "," tx-mode }
tx-mode ::= tx-access-mode | impl-defined-mode
tx-access-mode ::= "READ" "ONLY" | "READ" "WRITE"
commit-cmd ::= "COMMIT"
rollback-cmd ::= "ROLLBACK"
impl-defined-mode ::= ...
```

The syntax requires exactly one transaction access mode. Additional modes, including any means of selecting a less restrictive isolation level, are implementation-defined. Explicit transaction commands require optional Feature GT01, and explicit transaction characteristics require optional Feature GT02. If the characteristics are omitted, an implementation-defined (ID006) default characteristic is implicit and contains `READ WRITE`.

Begin/commit/rollback. A transaction may begin in either of two ways. It is initiated explicitly by `START TRANSACTION`, optionally with characteristics, or, if none is active, implicitly at the commencement of executing a procedure. It may contain a sequence of procedure executions and is terminated by an attempt to `COMMIT` or `ROLLBACK`. On a successful `COMMIT` the system applies $\Delta(B)$ atomically to Ω. If the transaction is part of an encompassing transaction controlled by another agent, 2D000 is raised. If any other error prevents commitment, all changes are canceled and 40000 is raised with an implementation-defined subclass. An explicit `ROLLBACK` likewise raises 2D000 rather than canceling changes when such an encompassing transaction is controlled by another agent; otherwise, all changes made by the active transaction are canceled.

Example: transaction tuple and `COMMIT` boundary. Let `alice` be the session authorization identifier and let P_{fees} be the procedure execution denoted by the call in Section 18.2. Then, in the abstract serializable transaction model used here, an explicitly demarcated transaction can be represented by

$$\mathsf{Tx}_{\mathsf{alice}} = \langle \mathtt{alice}, \mathsf{SERIALIZABLE}, \langle P_{\mathsf{fees}} \rangle, \mathsf{none} \rangle.$$

A corresponding GQL-program transaction activity is:

```
START TRANSACTION READ WRITE
CALL recomputeFees(DATE '2025-12-31')
YIELD updatedCount AS cnt
COMMIT
```

The `COMMIT` command requests termination of the transaction by commit; it is not an element of the procedure-execution sequence $\overline{P}$.

Visibility. For the serializable behavior assumed by the GQL core, concurrent activity is observed as a serial order. Under an implementation-defined relaxation, provisional changes may be visible, at some point or to some degree, to other GQL-agents executing concurrent transactions. After a successful `COMMIT`, the transaction's changes are eligible to be perceived by all subsequent transactions.

19.2 Serializability and Isolation

Serializability. The GQL core models only serializable transactional behavior. Thus, whenever transactions are described concurrently in this core model, there exists a total order $\prec$ of the committed transactions such that the observable outcome (final state and results of each transaction) is equivalent to executing those transactions one-by-one in order $\prec$. An implementation-defined relaxation need not satisfy this criterion unless it is specified as a serializable extension. For $\mathsf{Tx} = \langle u, \mathrm{iso}, \langle P_1, \ldots, P_m \rangle, \mathsf{SP} \rangle$, write $\Delta(\mathsf{Tx})$ for the cumulative delta obtained by composing $\Delta(P_1), \ldots, \Delta(P_m)$ in procedure-execution order:

$$\Delta(\mathsf{Tx}) = \Delta(P_1) \diamond \cdots \diamond \Delta(P_m).$$

Let Ω_{init} be the initial shared store state and Ω_{final} the resulting final shared store state. Formally, if the committed transactions are $\mathsf{Tx}_1, \ldots, \mathsf{Tx}_n$ and ρ is a permutation witnessing serializability, then

$$\Omega_{\text{final}} = \Big(\cdots ((\Omega_{\text{init}} \oplus \Delta(\mathsf{Tx}_{\rho(1)})) \oplus \Delta(\mathsf{Tx}_{\rho(2)})) \cdots \oplus \Delta(\mathsf{Tx}_{\rho(n)})) \Big),$$

and each $\mathsf{Tx}_{\rho(i)}$ observes effects of $\mathsf{Tx}_{\rho(j)}$ for all $j < i$ and none of $j > i$.

Snapshots. For an implementation-defined isolation extension, let Ω_i^{base} denote the base store state made visible to statement S_i by that extension. It may, for example, be selected at transaction start or at a statement boundary. The GQL core does not standardize such snapshot-selection rules. In the serializable core model, Ω_i^{base} is the committed state established by the transactions preceding this transaction in the witnessing serial order.

Isolation levels (axiomatic sketch). Isolation levels other than **SERIALIZABLE**, their interactions, their granularity, and the means of selecting them are implementation-defined. The following are examples of policies an extension might choose, not GQL-defined levels:

- **READ COMMITTED** (RC): each statement reads from the latest committed state at its start; writes are private until commit.
- **REPEATABLE READ** (RR): the base state is fixed for the transaction.
- **SNAPSHOT ISOLATION** (SI): commit succeeds iff no write-write conflict on overlapping entities/fields (first-committer-wins); otherwise a rollback in class `40xxx`.
- **SERIALIZABLE** (SER): there exists a serial order of concurrent transactions producing an equivalent outcome.

Statement execution under isolation. Let Δ_i be the delta produced by a successfully completed statement S_i, and let

$$\Delta_{<i} = \Delta_1 \diamond \cdots \diamond \Delta_{i-1}, \qquad \Omega_i^{\text{vis}} = \Omega_i^{\text{base}} \oplus \Delta_{<i}.$$

Both the read part and the computation of the write part of S_i use Ω_i^{vis}. If S_i completes successfully, its delta is added to the transaction-local cumulative delta, so the next statement observes it. This explicitly provides the standard's statement sequencing rule that the effects of a successfully completed statement are visible to a successor statement in the same transaction. If S_i signals an exception, evaluation of the enclosing request stops with that GQLSTATUS and Δ_i is not added.

19.3 Errors and Aborts

Abort outcomes. In the abstract model, execution may terminate with a successful RolledBack$(\cdot)$ or with Aborted(κ), in addition to Committed$(\cdot)$. The first is an explicitly requested successful rollback; the second records a failed outcome and rollback attempt. In either case, a successful rollback leaves no lasting transaction effect, while Aborted(κ) also raises exception condition κ to the agent.

Sources of termination without commit. A transaction can terminate without committing in any of the following situations: (i) an explicit `ROLLBACK` command, unless disallowed by an externally controlled encompassing transaction, successfully cancels its changes; (ii) the execution of a procedure has a failed outcome, which causes the GQL-implementation to attempt to roll back the current transaction; (iii) a `COMMIT` encounters an error other than invalid termination 2D000 that prevents commitment, in which case its changes are canceled and 40000 is raised with an implementation-defined subclass; or (iv) the transaction becomes blocked, cannot complete without semantic inconsistency, or loses resources required to continue, in which case it is implementation-dependent whether rollback is forced.

Rollback (abort) effect. If rollback of the currently active transaction succeeds, then all changes to GQL-data and/or the GQL-catalog performed by that transaction are canceled; the transaction is terminated and the current transaction is set to "not set". No provisional effect remains in the store. Under a less restrictive, implementation-defined isolation extension, this does not retroactively erase an observation that the extension may already have allowed another agent to make.

Error effect and diagnostics. If a statement signals an exception condition, the GQL-request is unsuccessful, the applicable General Rules stop, and the request has no effect

on GQL-data or the GQL-catalog. If the exception propagates to a failed procedure outcome, the implementation attempts to roll back the current transaction. This does not replace every original statement condition with class `40`: a transaction-rollback condition has precedence only when such a condition is actually raised (for example, 40000 when commitment is prevented). The presentation of diagnostics and the reporting of additional conditions are implementation-defined; logging and retry strategies are not specified.

Chapter 19 — Takeaways

- **Purpose:** Specified how procedure executions were grouped into transactions and how their effects applied atomically under concurrency.
- **Introduced:** Transactions as structured units of work; explicit start/commit/rollback behavior; serializable core behavior; implementation-defined isolation extensions.
- **Key distinctions:** Recovery atomicity versus isolation visibility; commit (durable apply) versus rollback (cancellation of provisional effects).
- **Consequences:** Determined which updates a statement could observe under each isolation choice; integrated constraint enforcement and abort behavior with transactional guarantees; fixed a correctness criterion for concurrent executions.
- **Bridge:** Made authorization meaningful by providing the execution envelope within which privilege checks and object availability are applied.

Part X

Security and Diagnostics

20 Authorization

This chapter gives an abstract, implementation-neutral description of authorization in GQL, focusing on how principals, roles, and privileges constrain access to catalog and data objects. In the broader security context, we use the term *security* primarily in the sense of *authorization* (access control): the rules that determine which principals may reference which GQL-objects, and which operations are permitted on them. Rather than fixing a specific access-control mechanism, we model privilege grants, obligation sets, and object availability in a way that can be instantiated by concrete implementations while preserving the previously defined pure semantics.

20.1 Roles and Privileges

Privileges. Let $\mathbb{PRIV}$ be the (implementation-defined) set of *privilege kinds/levels* (e.g., read, write, execute, administer, or finer-grained variants thereof). Privileges are modeled as a finite poset $\langle \mathbb{PRIV}, \leq \rangle$, where $\varpi \leq \varpi'$ is read as "ϖ' dominates (implies) ϖ", i.e., possessing ϖ' suffices wherever ϖ is required. We write $\bigsqcup P$ for the *least upper bound* (join) of a finite set $P \subseteq \mathbb{PRIV}$ (assumed to exist in this abstract model). The concrete elements of $\mathbb{PRIV}$ and their ordering are implementation-defined; here we reason over abstract elements $\varpi \in \mathbb{PRIV}$ without fixing names or generators.

Authorization identifiers, roles, and authorization targets. Let $(\mathbb{AID})$ be the set of *authorization identifiers*. Each authorization identifier $(a \in \mathbb{AID})$ identifies a principal together with a set of privileges under which GQL requests associated with (a) are executed. A single principal may be identified by more than one authorization identifier, allowing the same principal to operate under different privilege sets, for example under a restricted context for ordinary queries and a more privileged context for management operations.

Let $(\mathbb{ROLE})$ be an implementation-defined set of roles used to structure privilege assignments. A role represents a reusable grouping of related privileges and may be associated with one or more authorization identifiers. Typical role categories may provide read-only access, permission to modify graph data, permission to execute named procedures, permission to manage schema objects, or permission to administer privilege assignments. The available roles, their names, and the mechanism by which they are associated with authorization identifiers are implementation-defined.

Let $(\mathbb{OBJ})$ be the set of *authorization targets*, that is, entities to which privileges may apply. In the core model, authorization targets are complete GQL-objects, such as graphs, graph types, binding tables, and named procedures. Thus, a privilege may

permit a particular class of operations, such as reading, modifying, executing, defining, or administering, on an individual authorization target or on a class of such targets. The set of privileges identified by an authorization identifier is implementation-defined. An implementation may additionally define privileges on graph elements, labels, or properties, but that is an extension whose effect on query evaluation must also be defined by that implementation; the core semantics below does not silently turn a stored graph into a filtered subgraph.

Grants. An authorization state $\mathcal{Z}$ is a finite set of facts:

$$\begin{aligned} &\mathsf{hasRole}(u, r) \quad (u \in \mathbb{AID},\ r \in \mathbb{ROLE}), \\ &\mathsf{grant}(g, x, \varpi, o) \end{aligned}$$

where $g \in \mathbb{AID}$ is a grantor, $x \in \mathbb{AID} \cup \mathbb{ROLE}$, $\varpi \in \mathbb{PRIV}$, $o \in \mathbb{OBJ}$. (Grant-option policy and catalog specifics are implementation-defined.) Effective privilege for authorization identifier u on object o is

$$\mathsf{eff}(u, o) \;=\; \bigsqcup \{\, \varpi \mid \exists r.\ \mathsf{hasRole}(u, r) \wedge \mathsf{grant}(-, r, \varpi, o) \;\text{ or }\; \mathsf{grant}(-, u, \varpi, o) \,\},$$

taking joins in the poset $\langle \mathbb{PRIV}, \leq \rangle$.

Note

Formally, the GQL standard reserves the keywords `GRANT` and `REVOKE`, but at the time of writing, it does not assign them any normative statement syntax or execution semantics. In this chapter, we therefore model authorization abstractly, in terms of roles, grants, and effective privileges, without committing to a particular concrete language for privilege-management statements. Future versions of the standard are expected to refine this situation by introducing explicit syntax and semantics for such statements, at which point this abstract account can be instantiated accordingly.

20.2 Privilege Obligations and Enforcement

Obligation judgments. Let Θ be the binding-name-to-type environment used while typing a statement, and let $\mathfrak{S}$ be the applicable static schema/type context. We extend typing with a privilege obligation set:

$$\Theta; \mathfrak{S} \;\vdash\; S : \tau \;\triangleright\; \Pi_S,$$

where $\Pi_S \subseteq \mathbb{PRIV} \times \mathbb{OBJ}$ is the least set of privilege–object pairs required by the syntactic constituents of S. The set is computed compositionally: each primitive language construct contributes the obligations specified by its typing rule, while the obligations of a compound

construct are obtained by taking the union of the obligations computed for its immediate subconstructs. For example, sequential composition satisfies

$$\frac{\Theta; \mathfrak{S} \vdash S_1 : \tau_1 \triangleright \Pi_1 \quad \Theta; \mathfrak{S} \vdash S_2 : \tau_2 \triangleright \Pi_2}{\Theta; \mathfrak{S} \vdash S_1; S_2 : \tau_2 \triangleright \Pi_1 \cup \Pi_2}\ \textsc{Seq}.$$

Thus, no privilege obligation belongs to Π_S unless it is introduced by a constituent of S or by a rule for composing such constituents; repeated occurrences of the same obligation collapse because Π_S is a set.

The mapping from language constructs to required privilege kinds is implementation-defined; we therefore use abstract placeholders: let $\varpi_{\mathsf{match}}, \varpi_{\mathsf{insert}} \in \mathbb{PRIV}$ be representative required privilege kinds and let $o_\varphi, o_G \in \mathbb{OBJ}$ be the corresponding authorization targets.

$$\frac{\Theta; \mathfrak{S} \vdash \varphi \text{ well-typed on object } o_\varphi}{\Theta; \mathfrak{S} \vdash \mathsf{MATCH}\ \varphi : \text{empty type} \triangleright \{(\varpi_{\mathsf{match}}, o_\varphi)\}}\ \textsc{Match}$$

$$\frac{\Theta; \mathfrak{S} \vdash (v{:}L) \text{ well-typed in graph } o_G}{\Theta; \mathfrak{S} \vdash \mathsf{INSERT}\ (v{:}L) : \text{empty type} \triangleright \{(\varpi_{\mathsf{insert}}, o_G)\}}\ \textsc{Insert}.$$

Authorization check (dynamic). At execution under session authorization identifier u, the engine verifies that, for every required privilege–object pair $(\varpi, o) \in \Pi_S$, the effective privileges of u on o include or dominate the required privilege ϖ:

$$\forall (\varpi, o) \in \Pi_S.\ \ \mathsf{eff}(u, o) \geq \varpi.$$

Equivalently, authorization succeeds only if there is no $(\varpi, o) \in \Pi_S$ such that $\mathsf{eff}(u, o) \not\geq \varpi$. If this condition is not satisfied, the statement is rejected before its evaluation begins, and an appropriate authorization diagnostic is raised.

The objects in Π_S are determined from the syntactic form and catalog context so that authorization completes before the statement's General Rules are evaluated. Refining this required check later from objects actually accessed during execution is outside the core model.

Minimality/monotonicity. Π_S is minimal up to poset equivalence. We write $S \preccurlyeq S'$ to mean that S' is a *contextual extension* of S, i.e., there exists a (single-hole) statement context $C[\cdot]$ such that $S' = C[S]$ (so S occurs as a syntactic substatement of S'). Then obligations are monotone under such extensions: if $S \preccurlyeq S'$, then $\Pi_S \subseteq \Pi_{S'}$.

20.3 Authorized Object Access

Object availability. An execution context provides access only to the complete GQL-objects available to it. We represent this implementation-defined availability by

$$\mathsf{available} \subseteq \mathbb{AID} \times \mathbb{OBJ}.$$

Availability controls object reference and name resolution; permission to perform an operation is separately determined by the effective privileges of the session authorization identifier. In particular, availability of a graph does not denote availability of an induced subgraph.

Query semantics under authorization. Let $[\![S]\!]_{\mathcal{K},G}$ denote the pure semantics of a query statement S. Its authorization-identifier-indexed semantics is

$$[\![S]\!]^{u}_{\mathcal{K},G} = \begin{cases} [\![S]\!]_{\mathcal{K},G} & \text{if every referenced object is available to } u \\ & \quad \text{and } \forall(\varpi, o) \in \Pi_S.\ \mathsf{eff}(u, o) \geq \varpi, \\ \text{exception } 42000 & \text{otherwise.} \end{cases}$$

Thus an authorized query reads the ordinary, complete graph state. If the execution would perform an operation not permitted by the privileges identified by the session authorization identifier, the request instead raises *syntax error or access rule violation* (42000). Any finer-grained row-, element-, label-, or property-filtering semantics is an implementation-defined extension and is not inferred from this core model.

20.4 Write Checks

Write-related obligations. The obligation set Π_S for a statement S may contain privileges corresponding to read-only operations, write operations (data-modifying or catalog-modifying), or both. GQL characterizes privileges as permissions to cause certain kinds of operations to be performed and specifies that if the execution of a GQL-request would perform an operation not permitted by the privileges of the session authorization identifier, then the exception 42000 is raised. We model this by requiring that, for every operational step of S that attempts to modify a catalog or data object o, there is some $(\varpi, o) \in \Pi_S$ such that $\mathsf{eff}(u, o) \geq \varpi$; otherwise, execution raises 42000 before that modification takes effect.

Interaction with transaction access mode. GQL distinguishes the transaction access modes READ ONLY and READ WRITE, abbreviated as RO and RW, respectively, and mandates that:

- Executing a *linear catalog-modifying statement* when the current transaction access mode is RO raises *invalid transaction state—read-only GQL-transaction* (25G03).
- Executing a *linear data-modifying statement* when the current transaction access mode is RO likewise raises 25G03.
- Executing such statements when the current transaction access mode is RW is not rejected by the transaction access-mode rule (i.e., 25G03 is not raised on that basis); any rejection must arise from other rules, such as privilege checks.

Let

$$\mathsf{txMode} \in \{\mathsf{RO}, \mathsf{RW}\}$$

be the current transaction access mode. For a statement S whose syntactic statement kind is data-modifying or catalog-modifying, dynamic execution under session authorization identifier u proceeds as follows: first perform the privilege check

$$\forall(\varpi, o) \in \Pi_S.\ \mathsf{eff}(u, o) \geq \varpi,$$

raising 42000 on failure before the statement's General Rules are applied. If authorization succeeds, those General Rules then apply the transaction-mode check

$$\text{if } \mathsf{txMode} = \mathsf{RO} \text{ and } S \text{ may modify catalog or data, then raise 25G03.}$$

The modifying operations of S are permitted to proceed only if both checks succeed; the authorization step precedes the statement-level read-only check.

Atomicity and failure behavior. Transaction management in GQL requires that a transaction is atomic with respect to recovery: a request that raises 25G03 or 42000 has no effect on the GQL-catalog or on GQL-data. If the condition propagates to a failed outcome of the executing procedure, the implementation attempts to roll back the current transaction. Thus:

- No partial writes performed by S are observable in the committed state.
- No later component of the failed request is executed. If rollback succeeds, the transaction is terminated; the effect of a failed rollback attempt is governed by the transaction rules rather than by the authorization condition itself.

An authorization failure therefore supplies the standard access-rule condition; it is not itself reclassified as a transaction-rollback condition. If a transaction-rollback condition is also raised, the diagnostic precedence rules select that condition as primary.

Chapter 20 — Takeaways

- **Purpose:** Gave an abstract, implementation-neutral semantics of authorization for GQL programs.
- **Introduced:** Principals and roles; privileges and grants; object-availability and obligation concepts that control access; a read-only versus read-write transaction mode distinction.
- **Key distinctions:** Availability of a complete GQL-object versus permission to perform an operation on it; catalog-object privileges versus data-object privileges.
- **Consequences:** Preserved ordinary graph semantics for authorized queries; prevented unauthorized updates by making permission checks part of the semantics; left finer-grained filtering to explicitly defined implementation extensions.
- **Bridge:** Motivated diagnostics as the mechanism that made failures, warnings, and authorization denials observable and standardized.

21 Diagnostics

This chapter describes how the core semantics of GQL is augmented with diagnostics, in particular how successful completions, warnings, and failures are represented and propagated. We separate the pure result of statement evaluation from error outcomes, and we model the standardized space of GQLSTATUS codes together with their categories and precedence rules. This provides a bridge between the denotational semantics of earlier chapters and the observable behavior of a conforming implementation when exceptions or other conditions arise.

21.1 Errors and Exceptions

Let Err be the abstract carrier of exception conditions represented in the core semantics, and let $e_{\mathrm{err}} \in \mathsf{Err}$. Let $\mathsf{GQLStatus}$ be the opaque carrier of GQL-status objects, $\zeta \in \mathsf{GQLStatus}$ a primary status object, and $\overline{\zeta} \in \mathsf{GQLStatus}^*$ a finite list of additional status objects. Any execution may yield a failure outcome $\mathsf{Fail}(e_{\mathrm{err}}, \zeta, \overline{\zeta})$. Let the *failure-outcome carrier* be

$$\mathsf{Fail} \;=\; \{\, \mathsf{Fail}(e_{\mathrm{err}}, \zeta, \overline{\zeta}) \mid e_{\mathrm{err}} \in \mathsf{Err},\ \zeta \in \mathsf{GQLStatus},\ \overline{\zeta} \in \mathsf{GQLStatus}^* \,\}.$$

A failed outcome records an exception condition and has no result.

In accordance with the execution-outcome model, a regular result is either a value or a binding table. Write $\amalg$ for a disjoint sum. Let

$$\mathsf{RegularResult} \;=\; \mathbb{V} \;\amalg\; \mathsf{BT}, \qquad \mathsf{Result} \;=\; \mathsf{RegularResult} \;\amalg\; \{\, \mathsf{omitted} \,\}$$

be the carriers of regular results and all results, respectively. A successful outcome records exactly one regular or omitted result. The carrier of successful outcomes is

$$\mathsf{Succ} \;=\; \{\, \mathsf{Ok}(R, \zeta, \overline{\zeta}) \mid R \in \mathsf{Result} \;\wedge\; \zeta \in \mathsf{GQLStatus} \;\wedge\; \overline{\zeta} \in \mathsf{GQLStatus}^* \,\}.$$

Execution outcomes are then modeled by the disjoint sum

$$\mathsf{Res} \;=\; \mathsf{Succ} \;\amalg\; \mathsf{Fail}.$$

Compound statements are evaluated from left to right. If a component yields $\mathsf{Fail}(e_{\mathrm{err}}, \zeta, \overline{\zeta})$, no subsequent component is evaluated, and the same failure outcome is propagated to the enclosing compound statement.

21.2 Status Codes

Status domain. Let $\mathbb{CH}$ be the standard digits together with the simple Latin upper-case letters, the repertoire of characters that may occur in a GQLSTATUS code. Define the lexical domains

$$\mathbb{CLASS} = \mathbb{CH}^2, \qquad \mathbb{SUBCLASS} = \mathbb{CH}^3,$$

where, for $n \in \mathbb{N}$, $\mathbb{CH}^n$ denotes the set of strings of exactly n characters drawn from $\mathbb{CH}$. Let $\mathbb{STATUS} \subseteq \mathbb{CH}^5$ be the set of *status codes* (GQLSTATUS codes). Each $\kappa \in \mathbb{STATUS}$ identifies a single *condition*. Concretely, every $\kappa \in \mathbb{STATUS}$ is assumed to admit a unique factorization $\kappa = c_\kappa \cdot s_\kappa$ with $c_\kappa \in \mathbb{CLASS}$ and $s_\kappa \in \mathbb{SUBCLASS}$, where $\cdot$ denotes string concatenation. We then define

$$\begin{aligned} \mathrm{class}(\kappa) &:= c_\kappa \in \mathbb{CLASS}, \\ \mathrm{subclass}(\kappa) &:= s_\kappa \in \mathbb{SUBCLASS}. \end{aligned}$$

The exact lexical form and the assignment of class/subclass codes to conditions are defined by the standard and are not modeled here.

Categories. Each $\kappa \in \mathbb{STATUS}$ has a *category*

$$\mathrm{cat}(\kappa) \in \{\, \mathbf{S}, \mathbf{W}, \mathbf{N}, \mathbf{I}, \mathbf{X} \,\},$$

where:

- **S** (“successful completion”),
- **W** (“warning”),
- **N** (“no data”),
- **I** (“informational”),
- **X** (“exception”).

We write:

$$\begin{aligned} \mathbb{STATUS}_{\mathsf{compl}} &= \{\, \kappa \in \mathbb{STATUS} \mid \mathrm{cat}(\kappa) \in \{\mathbf{S}, \mathbf{W}, \mathbf{N}\} \,\}, \\ \mathbb{STATUS}_{\mathsf{exc}} &= \{\, \kappa \in \mathbb{STATUS} \mid \mathrm{cat}(\kappa) = \mathbf{X} \,\}, \\ \mathbb{STATUS}_{\mathsf{info}} &= \{\, \kappa \in \mathbb{STATUS} \mid \mathrm{cat}(\kappa) = \mathbf{I} \,\}. \end{aligned}$$

Completion, exception, and informational conditions. A *completion condition* is one whose code lies in $\mathbb{STATUS}_{\mathsf{compl}}$; these include in particular:

$$\kappa_{\mathsf{ok}} = \mathsf{00000}, \quad \kappa_{\mathsf{ok_omitted}} = \mathsf{00001}, \quad \kappa_{\mathsf{warning}} = \mathsf{01000}, \quad \kappa_{\mathsf{no_data}} = \mathsf{02000}.$$

Here κ_{ok}, and permitted implementation-defined subclasses of successful completion other than the omitted-result subclass, denote successful completion with a regular result;

$\kappa_{\mathsf{ok_omitted}}$ denotes successful completion with an omitted result, $\kappa_{\mathsf{warning}}$ denotes successful completion with a warning, and $\kappa_{\mathsf{no_data}}$ denotes successful completion with an empty result.

Except where otherwise specified, an *exception condition* is one whose code lies in $\mathbb{STATUS}_{\mathsf{exc}}$; such a condition causes the execution of a GQL-request to have no effect on GQL-data or the GQL-catalog beyond recording the diagnostics.

An *informational condition* is one whose code lies in $\mathbb{STATUS}_{\mathsf{info}}$; informational codes are used only in additional (nested) GQL-status objects and are not used as the primary status of an execution outcome. In particular, the standard informational condition has code `03000`.

Standard-defined vs. implementation-defined codes. The standard assigns GQLSTATUS codes to all conditions it defines and specifies that these assignments are stable. We write $\mathbb{STATUS}_{\mathsf{std}} \subseteq \mathbb{STATUS}$ for the set of codes assigned to conditions by this or another International Standard. Let $\mathbb{STATUS}_{\mathsf{impl}} \subseteq \mathbb{STATUS}$ be the separate set of codes used for implementation-defined conditions in the class/subclass ranges reserved for that purpose by the standard. It is not simply the complement of $\mathbb{STATUS}_{\mathsf{std}}$: an unassigned character string in a reserved standard-defined range does not thereby become an implementation-defined code. Implementation-defined completion conditions use implementation-defined subclasses of classes `00`, `01`, or `02`. An implementation-defined exception condition uses either a permitted implementation-defined subclass of a standard-defined exception class or an implementation-defined class (and subclass), as prescribed by the standard.

Status of execution outcomes. Every execution outcome (successful or failed) carries a *primary* GQL-status object whose GQLSTATUS field is $\kappa \in \mathbb{STATUS}$, and may carry a finite list of additional GQL-status objects for secondary conditions. The ISO standard requires:

- If an outcome is successful (i.e., yields a regular or omitted result), then its primary κ lies in $\mathbb{STATUS}_{\mathsf{compl}}$.
- If an outcome is failed (i.e., corresponds to some $\mathsf{Fail}(e_{\mathrm{err}}, \zeta, \overline{\zeta})$), then its primary κ lies in $\mathbb{STATUS}_{\mathsf{exc}}$.

The additional GQL-status objects are carried by the list $\overline{\zeta}$; their internal fields other than GQLSTATUS are left opaque in the core semantics.

When more than one condition could have been raised for a single GQL-request, the GQLSTATUS field of the primary object must respect the precedence rules of the standard:

- Exception conditions for transaction rollback (class code `40`) have precedence over all other exception conditions.

- Every exception condition has precedence over every completion condition.
- Among completion conditions, `02000` (no data) has precedence over `01000` (warning).
- `01000` (warning) has precedence over `00000` (successful completion) and its subclasses.

Our semantics is parametric in the implementation-dependent mechanism for choosing a primary exception condition among raised conditions of equal highest precedence.

Status codes for errors. Errors are reported via a *GQL-status object* whose GQLSTATUS field is a code $\kappa \in \mathbb{STATUS}_{\mathsf{exc}}$ and whose diagnostic record (if present) is a diagnostic record in the sense of the standard: it contains at least the standard fields describing the operation, its operation code, and the current schema, and may be extended with implementation-defined fields (whose names begin with an underscore). In this formalization, the diagnostic record is treated as an opaque value; we do not model its internal structure, and any additional information it carries is implementation-defined. We assume a total mapping

$$\mathsf{code} : \mathsf{Err} \longrightarrow \mathbb{STATUS}_{\mathsf{exc}}$$

that is consistent with the error classes introduced below. If an error denotes a condition defined by the standard, code returns that condition's stable, standard-assigned GQLSTATUS. Only an implementation-defined condition is mapped to a code from $\mathbb{STATUS}_{\mathsf{impl}}$, using the class and subclass ranges reserved by the standard.

21.3 Control Flow

Error classes. We partition the error domain as

$$\mathsf{Err} \;=\; \mathsf{static} \cup \mathsf{dynamic} \cup \mathsf{concurrency} \cup \mathsf{constraint} \cup \mathsf{resource}.$$

Static errors are detected before execution (e.g., unbound variables in the write-part shape, schema mismatch); dynamic errors occur during dynamic evaluation (e.g., numeric overflow or access through an invalidated graph-element reference); concurrency errors arise from isolation and conflict handling; constraint errors correspond to integrity or graph-type violations; resource errors include both standard-defined conditions and conditions introduced by an implementation. The mapping code respects this classification in the sense that each error class is associated with the corresponding family of GQLSTATUS exception classes prescribed by the standard (for example, data exceptions, transaction rollback, integrity constraint violation, graph type violation).

Primary-condition selection. Errors are reported using a *GQLSTATUS* code κ and a *GQL-status object* (with optional diagnostic record). When more than one condition could have been raised, the primary status is chosen in accordance with the precedence

rules summarized above. Within those constraints, if several raised exception conditions have the same highest precedence, the mechanism that selects the primary condition is implementation-dependent and is not modeled here. The set of conditions that is actually detected may itself vary where evaluation can short-circuit or where concurrency, resource availability, or external failures differ. Consequently, GQL does not guarantee that re-executing the same source text against an apparently identical application state yields the same primary status.

Chapter 21 — Takeaways

- **Purpose:** Integrated standardized diagnostics into the semantics so that failures and warnings became first-class outcomes.
- **Introduced:** A statement-result domain that distinguished normal results from failures; standardized status codes and their categories; precedence-based propagation and short-circuiting rules for compound statements.
- **Key distinctions:** Pure semantic result versus diagnostic outcome; failure short-circuiting versus primary-condition precedence.
- **Consequences:** Made implementation behavior observable while preserving implementation-dependent choices permitted by GQL; aligned abort behavior with standardized status precedence; clarified how exceptions interacted with query/update composition.
- **Bridge:** Opened the door to discussing what could evolve next and what remained research work.

Part XI

Future Directions

22 Future Enhancements

This chapter sketches plausible directions for the evolution of GQL beyond the current standardization snapshot, focusing on features that improve compositionality, interoperability, and practical deployability. We highlight enhancement themes that recur across vendor roadmaps and the research literature—graph-to-graph query composition, richer schema and constraint support, and extensions of the underlying data model. Finally, we outline how these directions connect to emerging needs around streaming graphs and AI-enabled workloads, while preserving the declarative and analyzable character of the language.

22.1 Compositional Enhancements

The ongoing development of GQL presents numerous opportunities to enhance its capabilities, particularly by introducing compositional features such as graph-returning subqueries and views defined using new operators for graph projection and combination (Deutsch et al. 2022; Francis et al. 2023a; Gheerbrant et al. 2025b; Kondylakis et al. 2025). These enhancements enable more complex, efficient, and secure data interactions. This section explores potential advancements in these areas and their implications for GQL.

GQL already supports nested query specifications, including correlated uses that can refer to bindings from an enclosing query. The principal compositional gap is therefore not ordinary table-producing subqueries but graph-returning subqueries whose results can serve as graph inputs to later query parts. Potential advancements include recursive graph-returning subqueries, which could support reusable fixpoint-style graph computations, and explicit subquery materialization for results that are reused within a larger query (Angles et al. 2017; Deutsch et al. 2022).

Views in GQL can provide a powerful mechanism for abstracting and reusing query logic, enhancing developer productivity and query performance. From a systems perspective, implementing property-graph views forces explicit choices between virtual evaluation and materialized storage, and it raises the need for view-aware query rewriting and indexing strategies. Recent work has developed and experimentally compared such implementation strategies across both native property-graph engines and relational backends, clarifying the performance trade-offs that arise when views become first-class and reusable program units (Han and Ives 2024; Han and Ives 2025). Moreover, when views are materialized (or partially materialized), incremental view maintenance becomes central; certified maintenance frameworks based on Regular Datalog demonstrate how to derive delta-style maintenance programs with formal soundness guarantees, which is directly relevant when view results must remain trustworthy under frequent graph updates

(Bonifati et al. 2018). Potential enhancements in this area include parameterized views, which enable views to accept input parameters, allowing for dynamic query construction and reusability across different contexts. Updatable views could permit modifications to the underlying graph data through views, simplifying data manipulation tasks. View composition would enable the creation of complex views by combining existing views, promoting modularity and code reuse (Angles et al. 2017; Deutsch et al. 2022).

Recent theoretical analyses of GQL and related graph querying facilities have also highlighted that current property graph query languages are, in practice, mostly graph-to-relation formalisms and therefore fall short of full compositionality at the graph level (Francis et al. 2023a; Bonifati 2025; Gheerbrant et al. 2025b). Future extensions of GQL could address this gap by providing a closed algebra of graph-to-graph operators whose results are again graphs and can be fed as inputs to subsequent queries (Angles et al. 2017; Gheerbrant et al. 2025a). Such an algebra would enable principled reasoning about query equivalence, rewriting, and optimization in a way that mirrors the role of relational algebra for SQL, while still respecting the semantics of path expressions, graph constraints, and schema conformance.

Recent work sharpens this limitation by showing that GQL and SQL/PGQ, despite being able to express reachability and all first-order queries, do not capture all queries in NLOGSPACE. The proposed language couples regular path queries with variables and data-value comparisons to #Datalog, a fully compositional graph-to-graph language that can construct nodes, edges, lists, and paths, and provides a concrete extension of both standards (Arenas et al. 2026).

A future graph-builder construct could offer graph-valued construction and transformation capabilities beyond the current standard. Such an extension would enable users to perform more sophisticated operations on graph data, facilitating tasks such as data integration, data warehousing, and graph-based machine learning applications. Potential features include conditional node and edge creation, which allows the creation of nodes and edges based on specific conditions, enabling dynamic graph construction. Batch operations could provide mechanisms for creating multiple nodes and edges in a single operation, improving efficiency for large-scale graph construction. Graph transformations would introduce functions for modifying existing graph structures, such as merging nodes, splitting edges, or applying graph algorithms (Angles et al. 2023; Kondylakis et al. 2025).

A concrete design point for such a facility uses GPC-based rules to construct output property graphs while accounting for property values rather than topology alone; for transformations using only simple and trail path restrictors, consistency checking is PSPACE-complete. A proof-of-concept implementation in openCypher shows that declarative transformation rules can be translated into executable scripts with practical advantages over ad hoc transformation scripts, suggesting that a GQL graph builder should

combine explicit constructors with run-time conflict detection and implementation-aware optimization (Bonifati et al. 2024).

22.2 Schemas in GQL

The ongoing development of GQL aims to bridge the gap between the current graph types and the complex schema requirements necessary for practical data graphs. This section outlines potential enhancements to the schema capabilities of GQL, drawing on the experiences and expertise of users, vendors, and researchers (Angles et al. 2021; Angles et al. 2023; Kondylakis et al. 2025).

The current GQL standard includes optional graph types that encompass sets of node types and edge types, together with key and unique constraints. However, the experience of enterprise graph deployments indicates that additional schema features are necessary to describe and prescribe the structure and values of practical data graphs, akin to the complexity found in SQL schema (Angles et al. 2023). Essential conceptual modeling and schema definition features include domains for user-defined refinements of data types (such as ranges and patterns), the ability to specify nested record types for graph element attribution, and the capacity to define subtyping relations for attribution types. Additionally, features like default and computed values, richer key modes, referential, cardinality, and participation constraints for entities and relationships, and support for N-ary relationships are desirable (Angles et al. 2021; Angles et al. 2023).

Community proposals for property graph schema systems already demonstrate that such rich schema facilities are both feasible and practically useful in real-world deployments (Angles et al. 2021; Angles et al. 2023; Manouvrier and Belhajjame 2024). They combine flexible typing with multi-inheritance, a spectrum of key modes for nodes, edges, and properties, and a unifying treatment of integrity constraints. Future revisions of GQL could explicitly align the schema language with these developments, for instance by standardizing key modes, reusable constraint templates, and mechanisms to relate graph types and schemas in a single DDL framework.

Schema evolution should also support controlled movement along the spectrum between descriptive and prescriptive schemas. One proposed framework models conformance by a homomorphism from an instance graph to a schema graph and models evolution by graph-rewriting operations propagated between the two levels, suggesting high-level GQL schema-manipulation operations for both data-driven schema adaptation and schema-driven data migration (Bonifati et al. 2019).

These prospective constraint templates can be grounded in the theory of graph dependencies, which provides reusable constraint schemata ranging from key- and equality-style constraints to generating dependencies that can prescribe the existence of additional graph structure when certain patterns are present (Fan and Lu 2019; Shimomura et al. 2020).

There are also graph-specific schema requirements that need to be addressed. These include the ability to specify sub-graphs as integral units for insertion or maintenance of data graphs, for access control purposes, or as the target of constraints. Another critical feature is schema inclusion or "schema unions", allowing the definition of nodes in a schema graph that can be the same or forcibly different from nodes of the same type in another schema graph. This capability facilitates the composition of schema graphs into larger, modular schemas and supports the creation of "simple sub-graph views" or "composite graphs" (Angles et al. 2023; Kondylakis et al. 2025).

The goal of extending the GQL schema language is to support a more restrictive set of constraints on the permissible values of GQL property graphs than currently allowed by graph types. The proposed schema definition language should be expressed as GQL DDL statements that either supplement or replicate the meaning of the existing DDL statements for graph-type definition and maintenance. This language will allow the creation, alteration, and deletion of graph schemas in a GQL catalog, including managing the relationship of an instance (data) graph to a graph schema throughout its lifecycle (Angles et al. 2021; Angles et al. 2023).

A comprehensive schema language for GQL should address the definition of schema conformance and the efficient maintenance of conformance during transactional updates. For instance, DML statements can explicitly reference schema sub-graphs to reduce the validation scope when executing mutations. The schema language should also support partial (extensible) schemas, allowing a "sliding scale of schema" that ranges from minimal constraints to schemas as or more restrictive than SQL schemas (Angles et al. 2023; Kondylakis et al. 2025). An extended GQL schema, possibly in conjunction with extensions to the data model (see below), has the opportunity to close the gap between graph schemas and attributed entity-relationship models, which permit N-ary relationships. This would more precisely align existing record-oriented conceptual data modeling techniques, such as Extended Entity Relationship Modeling or UML class diagrams, with graph schemas. These mechanisms could further interact with automated schema discovery and exploration tools for property graphs (Lbath et al. 2021; Bonifati et al. 2022).

22.3 Extending the Data Model and Integrating AI

The future of GQL holds great promise for advancements in data models and AI, enabling the development of sophisticated applications that leverage the full potential of graph databases. This section outlines potential directions for future versions of GQL, focusing on extending data models and integrating AI to enhance the language's capabilities and applicability (Angles et al. 2017; Kondylakis et al. 2025).

It is essential to explore extending the property graph data model in GQL to include features such as graph properties, stored paths, hyperedges, multiproperties, and support for first-class (multi)properties and property values (Kondylakis et al. 2025).

Graph properties enable annotating and organizing graphs in the catalog with additional rich metadata. Stored paths enable the storage and querying of frequently accessed routes within the graph, which is particularly useful in applications like social network analysis and recommendation systems. Hyperedges allow connections between more than two graph elements, providing a more expressive data model for representing complex relationships. Multiproperties enable multiple properties with the same name on a single node or edge, offering greater flexibility in data modeling. Support for first-class (multi)properties ensures that every multiproperty as well as every individual property value can be treated as a node, to allow their occurrence as the endpoints of edges in the graph topology and further (recursive) annotation with additional meta-properties. Finally, there are other relevant directions toward further development of the property graph data model landscape in extensions for temporal, geospatial, and probabilistic graph data, as well as the integration of property graph data management and stream processing (Kondylakis et al. 2025).

In particular, integrating streaming capabilities directly into GQL raises the prospect of a unified model for querying both static and continuously evolving property graphs. Future versions of the language could standardize windowing constructs, continuous query semantics, and incremental maintenance of graph views, so that streaming and at-rest graph data can be queried and optimized within a single conceptual framework rather than through ad hoc extensions (Rost et al. 2024; Kondylakis et al. 2025).

Standardizing concrete privilege-management and role-based access control (RBAC) statements in GQL can also enhance security and make user-permission management portable. RBAC allows for fine-grained control over access to graph data, ensuring that only authorized users can view or modify sensitive information. This is crucial for maintaining data integrity and confidentiality, especially in collaborative or multi-tenant environments.

Future versions of GQL should consider proposing statements for dynamically altering graph types, ensuring flexibility in adapting to evolving data requirements. This feature would enable users to modify the graph schema without recreating the entire graph, facilitating seamless schema evolution and minimizing disruptions to existing applications. Dynamic graph-type alteration can support a broader range of applications by allowing more adaptable and responsive data management practices (Angles et al. 2021; Angles et al. 2023).

Integrating vector search, knowledge graphs, and data science within the GQL framework can pave the way for building breakthrough Generative AI (GenAI) applications. These applications can deliver highly accurate responses, rich context, and deep explainability by leveraging the power of graph representations and AI models. Graphs can effectively represent hidden patterns and relationships within data, enabling GenAI

models to make sense of the world and understand a specific company's data and view of the world.

GQL already provides optional vector types and vector operations, including `VECTOR_DISTANCE` and `VECTOR_NORM`. The remaining opportunity is to integrate these values and operations with portable vector indexes, nearest-neighbor search, ranking, and query optimization. Such integration can improve the accuracy of search results and enable more effective information discovery, particularly in applications involving large and complex datasets, while complementing traditional graph querying with semantic relevance ranking.

A further opportunity lies in making AI techniques, such as graph embeddings and large language models, schema-aware. Current work on embeddings and generative models for graphs suggests that incorporating structural constraints and schema information remains challenging, yet essential for accuracy and interpretability (Hamilton et al. 2017). Tight integration of GQL with a rich schema language and constraint system could enable embeddings and GenAI components that respect keys, cardinalities, and type hierarchies by construction, and that can be explained back in terms of the underlying graph schema.

Incorporating data science techniques into GQL can enable advanced analytics and predictive modeling on graph data. By leveraging machine learning algorithms and statistical methods, GQL can uncover deep insights from the data, identify trends and patterns, and provide more informed decision-making. This integration can also enhance the capability of GenAI applications by combining structured graph data with powerful analytical tools (Hamilton et al. 2017).

The synergy of vector search, knowledge graphs, and data science can lead to the development of breakthrough GenAI applications that offer highly accurate responses, rich context, and deep explainability. These applications can leverage the interconnected nature of graph data to provide more meaningful and relevant insights, enhance user experiences, and drive innovation in various domains. For example, GenAI can utilize graph data in personalized recommendation systems to comprehend user preferences and relationships, resulting in more precise and tailored suggestions.

By incorporating these advancements, future versions of GQL can empower developers and data scientists to build more sophisticated, secure, and flexible applications that leverage the power of graph data models and AI. This will enhance the functionality and usability of GQL, enabling the development of next-generation intelligent systems that provide deeper insights and more effective solutions across various domains.

Chapter 22 — Takeaways

- **Purpose:** Sketched plausible evolution paths for GQL beyond the current snapshot, emphasizing compositionality and interoperability.
- **Introduced:** Candidate feature clusters such as stronger schema integration, data-model extensions, and AI-oriented capabilities framed as principled extensions.
- **Key distinctions:** Standardized language evolution versus implementation extensions; conservative, backward-compatible extensions versus features requiring new semantic commitments.
- **Consequences:** Identified which additions would require new value domains, typing rules, or catalog objects; highlighted compatibility constraints that shaped practical adoption; clarified how new features would interact with the existing semantic pipeline.
- **Bridge:** Motivated a broader research agenda by separating near-term extensions from foundational open problems.

23 Research Directions

This chapter outlines research directions that can shape the next generation of GQL systems and standards, with an emphasis on topics where formal foundations and engineering practice still diverge. We organize the discussion around core themes—expressiveness vs. complexity, query processing, updates and transactions, graph-to-graph compositionality, interoperability, and schema management—that repeatedly emerge in recent surveys and theoretical analyses of property-graph querying. Rather than prescribing a single roadmap, the chapter highlights open problems and promising alternatives that can be studied independently yet ultimately fit together into a coherent, implementable, and optimizable language design.

23.1 Expressiveness and Complexity

Understanding the trade-offs between the expressiveness and complexity of GQL queries is crucial (Angles et al. 2017; Gheerbrant et al. 2025b). Similar to the rich landscape of fragments and extensions in relational query languages, GQL requires a theoretical foundation to evaluate its expressiveness and computational complexity (Rogova 2024; Figueira et al. 2025). Researchers should focus on developing specific fragments of GQL, such as conjunctive regular path queries (CRPQs), and their extensions with counting, aggregation, and recursion (Barceló 2013; Gheerbrant et al. 2025a). They should also study the expressiveness of GQL's complex label expressions and path modes, including how they can be leveraged to solve real-world problems efficiently (Deutsch et al. 2022; Francis et al. 2023a). Analyzing the computational complexity of various GQL operations, particularly in pattern matching and aggregation, is also essential.

Recent theoretical work on GQL and related graph querying formalisms has begun to provide abstract models for these languages, clarifying their expressive power and limitations as primarily graph-to-relation query systems (Francis et al. 2023a; Gheerbrant et al. 2025b). A natural research direction is to refine such models so that they also capture genuinely graph-to-graph constructs, compositional query algebras, and the interaction between path modes, label expressions, and schema constraints (Angles et al. 2017; Kondylakis et al. 2025). This would enable a systematic comparison of GQL fragments with existing formalisms, such as CRPQs and their extensions, and support formal frameworks for query rewriting and optimization.

23.2 Query Processing and Optimization

Efficient query processing and optimization are central to the practical use of GQL. There is a need for advanced algorithms and data structures that can handle complex graph queries, both in native graph engines and in relational implementations of SQL/PGQ (Angles et al. 2017; Deutsch et al. 2022). Research in this area should focus on developing theories and algorithms for query equivalence and containment, which are essential for optimizing GQL queries (Gheerbrant et al. 2025b). Designing and implementing efficient algorithms for GQL query processing, particularly for native graph database systems and relational implementations, is crucial. Additionally, exploring innovative indexing and storage techniques that can enhance the performance of GQL queries, especially in large-scale graph datasets, is important (Kondylakis et al. 2025).

Schema-aware containment provides a concrete basis for semantic optimization, because inclusion and equivalence need only hold over graphs conforming to the schema. Recent results prove 2EXPTIME-completeness for unions of conjunctive regular path queries (UCRPQs) under $\mathcal{ALCI}$ or $\mathcal{ALCQ}$ schemas and for simple unions of conjunctive two-way regular path queries (UC2RPQs) under $\mathcal{ALCQ}$; they also give a 2EXPTIME upper bound for UC2RPQs under $\mathcal{ALCQI}$ schemas without participation constraints (Gutiérrez-Basulto et al. 2024).

Beyond traditional batch workloads, scalable processing of continuous and streaming graph queries remains an open challenge. Recent work on window-based continuous queries and stream processing for property graphs indicates that even relatively simple patterns can be difficult to evaluate efficiently under high update rates (Rost et al. 2024). Future research should therefore investigate cost models, indexing schemes, and execution architectures tailored to continuous GQL queries, including hybrid engines that jointly optimize over static and streaming graph data (Kondylakis et al. 2025).

23.3 Design Decisions and Alternatives

The current design of GQL includes specific decisions that might benefit from further exploration and validation. Academics should investigate alternative approaches and their implications, informed by emerging formal models of GQL and SQL/PGQ (Francis et al. 2023a; Gheerbrant et al. 2025b). This includes examining the implications of allowing concatenation of patterns containing different kinds of variables, which could enable more complex and flexible queries. Another area of interest is developing automaton models that incorporate group variables, enabling classical evaluation techniques and automata-theoretic constructions, such as product and determinization, in the spirit of existing work on graph pattern matching and regular path queries (Deutsch et al. 2022; Gheerbrant et al. 2025a). It is also important to conduct empirical studies to validate

current design decisions in GQL and propose alternatives that could enhance its usability and functionality.

23.4 Updates and Transaction Processing

The update and transaction processing capabilities of GQL are critical for maintaining data integrity and supporting dynamic graph modifications. Research should investigate the design of robust update languages that can efficiently handle complex graph updates, similar to SQL's transaction processing facilities, but adapted to property graph workloads. Exploring mechanisms for ensuring consistency and concurrency control in graph updates and preventing issues such as inconsistent changes and non-determinism is also necessary. Understanding how isolation levels, transactional semantics, and failure handling should be adapted or extended for graph workloads is another promising direction, complementing the current focus of the standards and surveys on query and schema aspects (Gheerbrant et al. 2025b; Kondylakis et al. 2025).

23.5 Graph-to-Graph Queries

One of the significant challenges in graph databases is designing a language that supports graph-to-graph queries, enabling composability and advanced query capabilities. Researchers should focus on designing languages that allow queries to produce graphs as output, supporting concepts like views and subqueries at the graph level rather than solely at the table level (Angles et al. 2017; Francis et al. 2023b; Bonifati 2025). Ensuring that the language supports the composability of queries, and allowing the output of one query to be used as the input for another, is also crucial. Investigating the algebraic properties of such graph-to-graph operators and their relationship to existing GQL constructs, path expressions, and prospective graph builders and graph views is an important aspect of this line of work (Deutsch et al. 2022; Gheerbrant et al. 2025b). Connections to work on graph-to-graph transformations between property graphs and RDF further motivate this direction (Bruyat et al. 2024).

23.6 Interoperability and Data Integration

Interoperability between property graphs and other data models, particularly RDF and relational data, is a central challenge for the broader adoption of graph technologies. One research direction is the design of meta-models that can faithfully capture both RDF and property graphs, enabling uniform reasoning and transformation across models (Angles et al. 2021; Angles et al. 2023). Complementary to this, semantics-preserving transformation frameworks are needed to translate RDF graphs into property graphs and back, while respecting schema information, constraints, and inference rules (Bruyat et al. 2024). Such frameworks should account for the different typing, labeling, and identity

mechanisms in the underlying models, and be closely aligned with the evolving property graph standards (Kondylakis et al. 2025).

Three direct mappings have been formalized: a simple mapping for schema-less RDF data, a schema-independent mapping to a generic property graph schema, and a schema-dependent mapping of complete RDF schema–instance pairs. The latter two preserve both semantics and information; under their formalization, this provides a basis for lossless RDF-to-property-graph translation and shows that the property-graph model subsumes the information capacity of RDF (Angles et al. 2020).

The S3PG approach extends this line by using SHACL and PG-Schema to coordinate schema and data transformations. It preserves RDF instance data and supported SHACL integrity constraints and is monotonic under source changes; it achieved 100% query-result accuracy on the evaluated workloads, while open problems include RDF-star, ontological definitions and inference, automated query translation, and optimization of generated property graphs (Rabbani et al. 2024).

A closely related topic is maintaining graph views across heterogeneous systems. As applications increasingly combine RDF knowledge graphs, property graphs, and relational data, there is a need for incremental, low-overhead mechanisms that maintain the consistency of derived or materialized graph views as the underlying sources evolve. Understanding how GQL graph-to-graph queries and transformation rules can serve as a unifying abstraction for such integration tasks, and how they can interoperate with schema languages such as PG-Schema and PG-Keys, is a promising research direction (Angles et al. 2021; Angles et al. 2023; Kondylakis et al. 2025).

23.7 Schema and Metadata Management

Effective schema and metadata management are essential for maintaining data integrity and facilitating advanced query optimization. Research in this area should include developing a more comprehensive schema language for GQL, with richer constraints and validations beyond the current key and unique constraints, as advocated by PG-Schema and PG-Keys (Angles et al. 2021; Angles et al. 2023). Investigating techniques for semantic query optimization based on schema information and enhancing the efficiency and accuracy of query processing are important (Gheerbrant et al. 2025b). Exploring methods for incremental validation of schemas and constraints and ensuring that changes to the graph data are validated efficiently is also essential.

Validation complexity is an important boundary for extensions to GQL's schema facilities. For PG-Schema graph types without integrity constraints, combined complexity is NP-complete and data complexity is in PTIME; combined complexity drops to PTIME when alternation between type combinations and unions is suitably restricted (Ciszewski et al. 2026).

A broader comparison of PG-Schema, ShEx, and SHACL shows that graph schemas should be studied not only for validation but also as inputs to optimization, data exchange, interoperability, and schema discovery. The comparison identifies central design questions relevant to GQL: whether constraints should be expressed in a controlled language or backed by an injected query language, and how expressive power should be divided between types and constraints (Hidders and Murlak 2026).

A complementary line of work is also important, focusing on the automatic discovery of schemas for property graphs. Existing approaches combine label-based and property-based heuristics, as well as probabilistic clustering, to infer hierarchical schemas that summarize large, heterogeneous graphs (Lbath et al. 2021; Bonifati et al. 2022). More recent work applies formal concept analysis to schema discovery and merging, yielding methods that can cope with multiple and evolving graphs while remaining interpretable for human designers. A key research problem is to design hybrid symbolic-statistical methods that remain robust under schema drift and graph evolution, and that can propose candidate GQL schemas or constraints that are both expressive and understandable to human designers.

Closely related is the largely unexplored topic of normalization for property graphs. While normalization theory is well-developed for relational schemas, only initial proposals exist for defining normal forms in the property graph setting. Future work should investigate how key constraints, cardinalities, and higher-arity relationships interact with redundancy, and how normalization procedures can be formulated to respect the graph structure and be expressed using GQL DDL and graph-to-graph transformations (Angles et al. 2021; Angles et al. 2023; Kondylakis et al. 2025).

Chapter 23 — Takeaways

- **Purpose:** Outlined research problems likely to shape the next generation of GQL systems and standards.
- **Introduced:** An agenda spanning expressiveness versus complexity, query processing and optimization, update and concurrency processing, interoperability, and schema/metadata management.
- **Key distinctions:** Expressiveness versus tractability; logical equivalence and rewriting versus physical optimization and execution strategy.
- **Consequences:** Identified where stronger formal foundations could enable better optimizers and more predictable behavior; highlighted design choices that remained underdetermined and required comparative evaluation; framed interoperability and metadata as central research drivers.
- **Bridge:** Closed the book by turning the completed formalization into a roadmap of concrete, evaluable open problems.

Bibliography

Abiteboul, Serge, Quass, Dallan, McHugh, Jason, Widom, Jennifer, and Wiener, Janet L. (1997). "The Lorel query language for semistructured data." In: *International Journal on Digital Libraries* 1.1, pp. 68–88. DOI: 10.1007/s007990050005.

Albahari, Joseph and Albahari, Ben (2008). *LINQ pocket reference: learn and implement LINQ for .NET applications*. O'Reilly Media.

Angles, Renzo (2012). "A comparison of current graph database models." In: *2012 IEEE 28th International Conference on Data Engineering Workshops*. IEEE, pp. 171–177.

Angles, Renzo, Arenas, Marcelo, Barceló, Pablo, Hogan, Aidan, Reutter, Juan L., and Vrgoč, Domagoj (2017). "Foundations of modern query languages for graph databases." In: *ACM Computing Surveys* 50.5, 68:1–68:40. DOI: 10.1145/3104031.

Angles, Renzo (2018). "The property graph database model." In: *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management*. Ed. by Dan Olteanu and Barbara Poblete. Vol. 2100. CEUR Workshop Proceedings. CEUR-WS.org, pp. 1–10. URL: https://ceur-ws.org/Vol-2100/paper26.pdf.

Angles, Renzo, Thakkar, Harsh, and Tomaszuk, Dominik (2020). "Mapping RDF databases to property graph databases." In: *IEEE Access* 8, pp. 86091–86110. DOI: 10.1109/ACCESS.2020.2993117.

Angles, Renzo, Bonifati, Angela, Dumbrava, Stefania, Fletcher, George, Hare, Keith W., Hidders, Jan, Lee, Victor E., Li, Bei, Libkin, Leonid, Martens, Wim, Murlak, Filip, Perryman, Josh, Savković, Ognjen, Schmidt, Michael, Sequeda, Juan, Staworko, Slawek, and Tomaszuk, Dominik (2021). "PG-Keys: keys for property graphs." In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD '21. Virtual Event, China: Association for Computing Machinery, pp. 2423–2436. DOI: 10.1145/3448016.3457561.

Angles, Renzo, Bonifati, Angela, Dumbrava, Stefania, Fletcher, George, Green, Alastair, Hidders, Jan, Li, Bei, Libkin, Leonid, Marsault, Victor, Martens, Wim, Murlak, Filip, Plantikow, Stefan, Savković, Ognjen, Schmidt, Michael, Sequeda, Juan, Staworko, Slawek, Tomaszuk, Dominik, Voigt, Hannes, Vrgoč, Domagoj, Wu, Mingxi, and Zivkovic, Dusan (June 2023). "PG-Schema: schemas for property graphs." In: *Proceedings of the ACM on Management of Data* 1.2, Article 198. DOI: 10.1145/3589778.

ArangoDB (2024). *AQL Documentation*. URL: https://docs.arangodb.com/stable/aql/.

Arenas, Marcelo, Libkin, Leonid, and Martens, Wim (2026). *A compositional language for property graphs*. arXiv: 2606.23399 [cs.DB].

Banker, Kyle, Garrett, Douglas, Bakkum, Peter, and Verch, Shaun (2016). *MongoDB in action: covers MongoDB version 3.0*. Simon and Schuster.

Barceló, Pablo (2013). "Querying graph databases." In: *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '13)*. New York, NY, USA: ACM, pp. 175–188. DOI: 10.1145/2463664.2465216.

Berners-Lee, Tim (Sept. 1998). *Semantic Web Road map*. URL: https://www.w3.org/DesignIssues/Semantic.html.

Bizer, Christian, Cyganiak, Richard, Gauss, Tobias, and Maresch, Oliver (2005). "The TriQL.P browser: filtering information using context-, content- and rating-based trust policies." In: *Proceedings of the Semantic Web and Policy Workshop at the 4th International Semantic Web Conference*. Vol. 7, pp. 12–20.

Blakeley, José A., Thompson, Craig W., and Alashqur, Abdallah M. (1991). "Strawman reference model for object query languages." In: *Computer Standards & Interfaces* 13.1–3, pp. 185–199. DOI: 10.1016/0920-5489(91)90026-V.

Bonifati, Angela, Dumbrava, Stefania, and Gallego Arias, Emilio Jesús (2018). "Certified graph view maintenance with regular Datalog." In: *Theory and Practice of Logic Programming* 18.3–4, pp. 372–389. DOI: 10.1017/S1471068418000224.

Bonifati, Angela, Furniss, Peter, Green, Alastair, Harmer, Russ, Oshurko, Eugenia, and Voigt, Hannes (2019). "Schema validation and evolution for graph databases." In: *Proceedings of the 38th International Conference on Conceptual Modeling (ER)*. LNCS. Springer, pp. 448–456. DOI: 10.1007/978-3-030-33223-5_37.

Bonifati, Angela, Dumbrava, Stefania-Gabriela, Martinez, Emile, Ghasemi, Fatemeh, Jaffré, Malo, Luton, Pacome, and Pickles, Thomas (2022). "DiscoPG: property graph schema discovery and exploration." In: *Proceedings of the VLDB Endowment* 15.12, pp. 3654–3657. DOI: 10.14778/3554821.3554867.

Bonifati, Angela, Murlak, Filip, and Ramusat, Yann (2024). "Transforming property graphs." In: *Proceedings of the VLDB Endowment* 17.11, pp. 2906–2918. DOI: 10.14778/3681954.3681972.

Bonifati, Angela (2025). "Versatile property graph transformations." In: *Proceedings of the VLDB Endowment* 18.12, pp. 5516–5526.

Botoeva, Elena, Calvanese, Diego, Cogrel, Benjamin, and Xiao, Guohui (2018). "Expressivity and complexity of MongoDB queries." In: *21st International Conference on Database Theory (ICDT 2018)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

Broekstra, Jeen and Kampman, Arjohn (2003). "SeRQL: a second generation RDF query language." In: *Proc. SWAD-Europe Workshop on Semantic Web Storage and Retrieval*. Citeseer, pp. 13–14.

Brown, Martin C. (2012). *Getting started with Couchbase server*. O'Reilly Media, Inc.

Bruyat, Julian, Champin, Pierre-Antoine, Médini, Lionel, and Laforest, Frédérique (2024). "PRSC: from PG to RDF and back, using schemas." In: *Semantic Web*. DOI: 10.3233/SW-243675.

Buneman, Peter, Fernandez, Mary, and Suciu, Dan (2000). "UnQL: a query language and algebra for semistructured data based on structural recursion." In: *The VLDB Journal* 9.1, pp. 76–110. DOI: 10.1007/s007780050084.

Carroll, Jeremy and Klyne, Graham (Feb. 2004). *Resource description framework (RDF): concepts and abstract syntax*. W3C Recommendation. W3C. URL: https://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

Cattell, R. G. G., Barry, Douglas K., Berler, Mark, Eastman, Jeff, Jordan, David, Russell, Craig, Schadow, Olaf, Stanienda, Torsten, and Velez, Fernando, eds. (2000). *The object data standard: ODMG 3.0*. San Francisco: Morgan Kaufmann.

Chen, Peter P. (1976). "The entity-relationship model – toward a unified view of data." In: *ACM Transactions on Database Systems* 1.1, pp. 9–36.

Ciszewski, Jacek, Kłos, Jakub, Jakubowski, Maxime, Tomaszuk, Dominik, and Murlak, Filip (2026). *Validation of graph databases against PG-Schema*. arXiv: 2606.06127 [cs.DB].

CODASYL. Systems Committee (1971). *Feature analysis of generalized data base management systems*. Association for Computing Machinery.

Consens, Mariano P. and Mendelzon, Alberto O. (1990). "GraphLog: a visual formalism for real life recursion." In: *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 404–416.

Cruz, Isabel F., Mendelzon, Alberto O., and Wood, Peter T. (1987). "A graphical query language supporting recursion." In: *ACM SIGMOD Record* 16.3, pp. 323–330.

Da Silva, Maxwell Dayvson and Tavares, Hugo Lopes (2015). *Redis essentials*. Packt Publishing Ltd.

Deutsch, Alin, Francis, Nadime, Green, Alastair, Hare, Keith, Li, Bei, Libkin, Leonid, Lindaaker, Tobias, Marsault, Victor, Martens, Wim, Michels, Jan, Murlak, Filip, Plantikow, Stefan, Selmer, Petra, Rest, Oskar van, Voigt, Hannes, Vrgoč, Domagoj, Wu, Mingxi, and Zemke, Fred (2022). "Graph pattern matching in GQL and SQL/PGQ." In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Philadelphia, PA, USA: Association for Computing Machinery, pp. 2246–2258. DOI: 10.1145/3514221.3526057.

Dyck, Michael, Spiegel, Josh, and Robie, Jonathan (Mar. 2017). *XQuery 3.1: an XML query language*. W3C Recommendation. W3C. URL: https://www.w3.org/TR/2017/REC-xquery-31-20170321/.

Elmasri, Ramez and Navathe, Shamkant B. (2015). *Fundamentals of database systems (7th edition)*. Pearson.

Fan, Wenfei and Lu, Ping (Feb. 2019). "Dependencies for graphs." In: *ACM Transactions on Database Systems* 44.2, Article 5. DOI: 10.1145/3287285.

Figueira, Diego, Lin, Anthony W., and Peterfreund, Liat (2025). *Complexity of evaluating GQL queries*. arXiv: 2407.06766 [cs.DB]. URL: https://arxiv.org/abs/2407.06766.

Fowler, Martin (2003). *UML distilled: a brief guide to the standard object modeling language*. 3rd ed. USA: Addison-Wesley Longman Publishing Co., Inc.

Francis, Nadime, Green, Alastair, Guagliardo, Paolo, Libkin, Leonid, Lindaaker, Tobias, Marsault, Victor, Plantikow, Stefan, Rydberg, Mats, Selmer, Petra, and Taylor, Andrés (2018). "Cypher: an evolving query language for property graphs." In: *Proceedings of the 2018 international conference on management of data*, pp. 1433–1445. DOI: 10.1145/3183713.3190657.

Francis, Nadime, Gheerbrant, Amélie, Guagliardo, Paolo, Libkin, Leonid, Marsault, Victor, Martens, Wim, Murlak, Filip, Peterfreund, Liat, Rogova, Alexandra, and Vrgoč, Domagoj (2023a). "A researcher's digest of GQL (invited talk)." In: *26th International Conference on Database Theory, ICDT 2023, March 28–31, 2023, Ioannina, Greece*. Ed. by Floris Geerts and Brecht Vandevoort. Vol. 255. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:22. DOI: 10.4230/LIPICS.ICDT.2023.1.

Francis, Nadime, Gheerbrant, Amélie, Guagliardo, Paolo, Libkin, Leonid, Marsault, Victor, Martens, Wim, Murlak, Filip, Peterfreund, Liat, Rogova, Alexandra, and Vrgoč, Domagoj (2023b). "GPC: a pattern calculus for property graphs." In: *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pp. 241–250. DOI: 10.1145/3584372.3588662.

Gheerbrant, Amélie, Libkin, Leonid, Peterfreund, Liat, and Rogova, Alexandra (2025a). "Database theory in action: Cypher, GQL, and regular path queries." In: *28th International Conference on Database Theory, ICDT 2025, March 25–28, 2025, Barcelona, Spain*. Ed. by Sudeepa Roy and Ahmet Kara. Vol. 328. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 36:1–36:5. DOI: 10.4230/LIPICS.ICDT.2025.36.

Gheerbrant, Amélie, Libkin, Leonid, Peterfreund, Liat, and Rogova, Alexandra (Feb. 2025b). "GQL and SQL/PGQ: theoretical models and expressive power." In: *Proceedings of the VLDB Endowment* 18.6, pp. 1798–1810. DOI: 10.14778/3725688.3725707.

Gorman, Michael M. (1978). *American National Standards Institute, X3H2 records (CBI 168)*. Minneapolis. URL: https://archives.lib.umn.edu/repositories/3/resources/283.

Gössner, Stefan, Normington, Glyn, and Bormann, Carsten (Feb. 2024). *JSONPath: Query Expressions for JSON*. Tech. rep. 9535. IETF. 62 pp. DOI: 10.17487/RFC9535. URL: https://www.rfc-editor.org/info/rfc9535.

Grumbach, Stéphane and Milo, Tova (1996). "Towards tractable algebras for bags." In: *Journal of Computer and System Sciences* 52.3, pp. 570–588. DOI: 10.1006/jcss.1996.0042.

Gutiérrez-Basulto, Víctor, Gutowski, Albert, Ibáñez-García, Yazmín A., and Murlak, Filip (2024). "Containment of graph queries modulo schema." In: *Proceedings of the ACM on Management of Data* 2.2, 77:1–77:26. DOI: 10.1145/3651140.

Güting, Ralf Hartmut (1994). "GraphDB: modeling and querying graphs in databases." In: *VLDB*. Vol. 94. Citeseer, pp. 12–15.

Gyssens, Marc, Paredaens, Jan, and Van Gucht, Dirk (1990). "A graph-oriented object model for database end-user interfaces." In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of data*, pp. 24–33. DOI: 10.1145/93597.93616.

Hamilton, William L., Ying, Rex, and Leskovec, Jure (2017). "Representation learning on graphs: methods and applications." In: *IEEE Data Engineering Bulletin* 40.3, pp. 52–74. URL: http://sites.computer.org/debull/A17sept/p52.pdf.

Han, Soonbo and Ives, Zachary G. (May 2024). "Implementation strategies for views over property graphs." In: *Proceedings of the ACM on Management of Data* 2.3, Article 146. DOI: 10.1145/3654949.

Han, Soonbo and Ives, Zachary G. (Apr. 2025). "Implementing views for property graphs." In: *ACM SIGMOD Record* 54.1, pp. 59–68. DOI: 10.1145/3733620.3733633.

Harris, Steven and Seaborne, Andy (Mar. 2013). *SPARQL 1.1 query language*. W3C Recommendation. W3C. URL: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

Hendler, James, Lassila, Ora, and Berners-Lee, Tim (2001). "The semantic web." In: *Scientific American* 284.5, pp. 34–43.

Hidders, Jan and Murlak, Filip (2026). "Schemas for graph data." In: *ACM SIGMOD Record* 55.2, pp. 7–22.

Hunger, M. (2012). *Good relationships: the Spring Data NEO4J guide book*. InfoQ enterprise software development series. C4Media.

International Organization for Standardization (June 2023). *ISO/IEC 9075-16:2023 Information technology – Database languages SQL – Part 16: Property Graph Queries (SQL/PGQ)*. Standard. Geneva, CH: International Organization for Standardization.

International Organization for Standardization (Apr. 2024). *ISO/IEC 39075:2024 Information technology – Database languages – GQL*. Standard. Geneva, CH: International Organization for Standardization.

ISO/IEC (2016). *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*. Standard ISO/IEC 9075-2:2016. SQL:2016. ISO/IEC.

ISO/IEC (2023a). *Information technology — Database languages — SQL*. Standard ISO/IEC 9075:2023. Parts 1–4, 9–11, 13–16 (SQL:2023 family of standards). ISO/IEC.

ISO/IEC (2023b). *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*. Standard ISO/IEC 9075-2:2023. ISO/IEC.

ISO/IEC (2024). *Information technology — Database languages — GQL*. Standard ISO/IEC 39075:2024. ISO/IEC.

Karvounarakis, Gregory, Alexaki, Sofia, Christophides, Vassilis, Plexousakis, Dimitris, and Scholl, Michel (2002). "RQL: a declarative query language for RDF." In: *Proceedings of the 11th International Conference on World Wide Web*, pp. 592–603.

Klein, Barbara, Long, Rick, Blackman, Kenneth Ray, Goff, Diane Lynne, Nathan, Stephen P., Lanyi, Moira McFadden, Wilson, Margaret M., Butterweck, John, and Sherrill, Sandra L. (Nov. 2011). *An introduction to IMS*. IBM Press.

Kondylakis, Haridimos, Dumbrava, Stefania, Lissandrini, Matteo, Yakovets, Nikolay, Bonifati, Angela, Efthymiou, Vasilis, Fletcher, George, Plexousakis, Dimitris, Tommasini, Riccardo, Troullinou, Georgia, and Ymeralli, Elisjana (Aug. 2025). "Property graph standards: state of the art & open challenges." In: *Proceedings of the VLDB Endowment* 18.12, pp. 5477–5481. DOI: 10.14778/3750601.3750698.

Kuper, Gabriel M. and Vardi, Moshe Y. (1984). "A new approach to database logic." In: *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 86–96.

Lbath, Hanâ, Bonifati, Angela, and Harmer, Russ (2021). "Schema inference for property graphs." In: *Proceedings of the 24th International Conference on Extending Database Technology (EDBT 2021)*, pp. 499–504.

Levene, Mark and Poulovassilis, Alexandra (1990). "The hypernode model and its associated query language." In: *Proceedings of the 5th Jerusalem Conference on Information Technology, 1990.'Next Decade in Information Technology'*. IEEE, pp. 520–530.

Libkin, Leonid and Wong, Limsoon (1997). "Query languages for bags and aggregate functions." In: *Journal of Computer and System Sciences* 55.2, pp. 241–272. DOI: 10.1006/jcss.1997.1523.

Lindaaker, Tobias (2018). *An overview of the recent history of graph query languages*. Tech. rep. openCypher.org.

Manouvrier, Maude and Belhajjame, Khalid (2024). "PG-FD: mapping functional dependencies to the future property graph schema standard." In: *European Conference on Advances in Databases and Information Systems*. Springer, pp. 45–59.

Miller, Libby, Seaborne, Andy, and Reggiori, Alberto (2002). "Three implementations of SquishQL, a simple RDF query language." In: *International Semantic Web Conference*. Springer, pp. 423–435.

Morra, Olof H. (2021). *A semantics of GQL: a new query language for property graphs formalized*. Internship final report. Eindhoven University of Technology. URL: https://github.com/OlofMorra/GQL-parser/blob/main/src/main/resources/report/A%20Semantics%20of%20GQL%3B%20a%20New%20Query%20Language%20forProperty%20Graphs%20Formalized.pdf. Implementation repository: https://github.com/OlofMorra/GQL-parser.

North, Ken (Sept. 1968). *Database systems: the first generation*. CODASYL Systems Committee.

Ong, Kian Win, Papakonstantinou, Yannis, and Vernoux, Romain (2014). "The SQL++ query language: configurable, unifying and semi-structured." In: *arXiv preprint arXiv:1405.3631*.

Paredaens, Jan, Peelman, Peter, and Tanca, Letizia (1995). "G-Log: a graph-based query language." In: *IEEE Transactions on Knowledge and Data Engineering* 7.3, pp. 436–453.

Passant, Alexandre, Polleres, Axel, and Gearon, Paula (Mar. 2013). *SPARQL 1.1 update*. W3C Recommendation. W3C. URL: https://www.w3.org/TR/2013/REC-sparql11-update-20130321/.

Poulovassilis, Alexandra and Levene, Mark (1994). "A nested-graph model for the representation and manipulation of complex objects." In: *ACM Transactions on Information Systems (TOIS)* 12.1, pp. 35–68.

Rabbani, Kashif, Lissandrini, Matteo, Bonifati, Angela, and Hose, Katja (2024). "Transforming RDF graphs to property graphs using standardized schemas." In: *Proceedings of the ACM on Management of Data* 2.6, 242:1–242:25. DOI: 10.1145/3698817.

Robie, Jonathan, Dyck, Michael, and Spiegel, Josh (Mar. 2017). *XML path language (XPath) 3.1*. W3C Recommendation. W3C. URL: https://www.w3.org/TR/2017/REC-xpath-31-20170321/.

Rogova, Alexandra (2024). "Design principles of property graph languages: a theoretical and experimental study." PhD thesis. Université Paris Cité.

Rost, Christopher, Tommasini, Riccardo, Bonifati, Angela, Della Valle, Emanuele, Rahm, Erhard, Hare, Keith W., Plantikow, Stefan, Selmer, Petra, and Voigt, Hannes (2024). "Seraph: continuous queries on property graph streams." In: *Proceedings of the 27th International Conference on Extending Database Technology (EDBT 2024)*, pp. 234–247. DOI: 10.48786/EDBT.2024.21.

Sadalage, Pramod J. and Fowler, Martin (2012). *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Addison-Wesley Professional.

Saryerwinnie, James (2015). *JMESPath Specification JMESPath — jmespath.org*. URL: https://jmespath.org/specification.html.

Seaborne, Andy and Prud'hommeaux, Eric (Jan. 2008). *SPARQL query language for RDF*. W3C Recommendation. W3C. URL: https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

Sheng, Lei, Ozsoyoglu, Meral, and Ozsoyoglu, Gultekin (1999). "A graph query language and its query processing." In: *Proceedings 15th International Conference on Data Engineering (Cat. No. 99CB36337)*. IEEE, pp. 572–581.

Shimomura, Larissa C., Fletcher, George, and Yakovets, Nikolay (2020). "GGDs: graph generating dependencies." In: *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. CIKM '20. Virtual Event, Ireland: Association for Computing Machinery, pp. 2217–2220. DOI: 10.1145/3340531.3412149.

Sivasubramanian, Swaminathan (2012). “Amazon DynamoDB: a seamlessly scalable non-relational database service.” In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 729–730.

Sowa, John F. (1976). “Conceptual graphs for a data base interface.” In: *IBM Journal of Research and Development* 20.4, pp. 336–357.

Sowa, John F. (1984). *Conceptual structures: information processing in mind and machine*. Addison-Wesley Longman Publishing Co., Inc.

Thalheim, Bernhard (2018). “Extended Entity-Relationship Model.” In: *Encyclopedia of Database Systems, Second Edition*. Ed. by Ling Liu and M. Tamer Özsu. Springer. DOI: 10.1007/978-1-4614-8265-9_157.

Tolle, Karsten (2006). “Semantisches Web und Kontext: Speicherung von und Anfrage auf RDF-Daten unter Berücksichtigung des Kontextes.” PhD thesis. Universitätsbibliothek Johann Christian Senckenberg.

The Unicode Consortium (Sept. 2025). *Unicode technical standard #10: Unicode collation algorithm*. Ed. by Ken Whistler and Markus Scherer. Unicode Technical Standard UTS #10. Version 17.0.0 (Revision 53), dated 2025-09-03. The Unicode Consortium. URL: https://www.unicode.org/reports/tr10/tr10-53.html.

Acronyms

2EXPTIME	Double-Exponential Time
ACID	Atomicity, Consistency, Isolation, Durability
ACM	Association for Computing Machinery
AI	Artificial Intelligence
ANSI	American National Standards Institute
AQL	ArangoDB Query Language
ASCII	American Standard Code for Information Interchange
BNF	Backus–Naur Form
BT	Binding Table
CG	Conceptual Graph
CODASYL	Conference on Data System Languages
COBOL	Common Business-Oriented Language
CRPQ	Conjunctive Regular Path Query
DBTG	Data Base Task Group
DDL	Data Definition Language
DML	Data Manipulation Language
DQL	Data Query Language
EDBT	International Conference on Extending Database Technology
ETL	Extract, Transform, Load
GenAI	Generative AI
GPBT	Graph Pattern Binding Table
GPC	Graph Pattern Calculus
GPML	Graph Pattern Matching Language
GPV	Graph Pattern Variable
GQL	Graph Query Language
GQLSTATUS	Status-code field of a GQL-status object
IBM	International Business Machines
ICDT	International Conference on Database Theory
ID	Implementation-Defined (ISO/IEC 39075 requirement-code prefix)
IEC	International Electrotechnical Commission
IDS	Integrated Data Store

IGP	Insert Graph Pattern
IL	Implementation-Defined Limit (ISO/IEC 39075 requirement-code prefix)
IMS	Information Management System
INCITS	International Committee for Information Technology Standards
ISO	International Organization for Standardization
ISBN	International Standard Book Number
IT	Information Technology
JSON	JavaScript Object Notation
KLS	Key Label Set
LINQ	Language Integrated Query
N1QL	Non-First Normal Form Query Language
NDL	Network Data Language
NLOGSPACE	Nondeterministic Logarithmic Space
NoSQL	Not Only SQL
NP	Nondeterministic Polynomial Time
ODMG	Object Data Management Group
OEM	Object Exchange Model
OQL	Object Query Language
PEL	Path Element List
PG	Property Graph
PGQ	Property Graph Queries
PODS	ACM Symposium on Principles of Database Systems
PSPACE	Polynomial Space
PTIME	Polynomial Time
RBAC	Role-Based Access Control
RC	Read Committed
RDBMS	Relational Database Management System
RDF	Resource Description Framework
RDQL	RDF Data Query Language
RO	Read Only
RPE	Regular Path Expression
RPQ	Regular Path Query
RQL	RDF Query Language
RR	Repeatable Read
RW	Read Write
S3PG	Standardized SHACL Shapes-based Property Graph Transformation
SER	Serializable
SHACL	Shapes Constraint Language

ShEx	Shape Expressions
SI	Snapshot Isolation
SIGMOD	Special Interest Group on Management of Data
SPARQL	SPARQL Protocol and RDF Query Language
SPP	Selective Path Pattern
SPSP	Selective Path Search Prefix
SQL	Structured Query Language
TOIS	ACM Transactions on Information Systems
UC2RPQ	Union of Conjunctive Two-Way Regular Path Queries
UCRPQ	Union of Conjunctive Regular Path Queries
UCS	Universal Coded Character Set
UML	Unified Modeling Language
UTS	Unicode Technical Standard
VLDB	Very Large Data Bases
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Symbols and Notations

$\mathbb{A}$	Universe of identifiers / atomic names (catalog names, labels, property keys, column names).
$\mathbb{L} \subseteq \mathbb{A}$	Universe of label names.
$\mathbb{K} \subseteq \mathbb{A}$	Universe of property keys (property names).
$\mathbb{I}$	Universe of globally unique graph-element identifiers.
$\mathbb{T}$	Universe of types (static, dynamic, and base types).
$\mathbb{V}$	Universe of material values together with the null value.
null	Distinguished null value `NULL`.
$\mathbb{N}$	Natural numbers.
∞	Infinity (used for implementation-defined maxima and unbounded path-pattern or RPE repetition).
$\mathbb{Z}_{>0}$	Positive integers.
X^*	Set of finite sequences over X, including the empty sequence.
$\mathcal{P}(X)$	Powerset of X.
$\mathcal{M}(X)$	Bag/multiset of elements from X (multiplicity function over X).
$\mathcal{M}_{\text{fin}}(X)$	Finite bag/multiset over X.
$\rightharpoonup$	Partial function / partial map.
$\mapsto$	Mapping constructor (e.g., $k \mapsto w$).
$\triangleq$	Definitional equality.
$[\![\cdot]\!]$	Denotation / semantic evaluation function.
$\vdash$	Typing / judgment turnstile.
$\models$	Satisfaction relation.
$\sqsubseteq$	Subtype relation.
Ξ	Static type environment.
G	A (GQL) property graph.
$V \subseteq \mathbb{I}$	Finite set of node identifiers.
$E \subseteq \mathbb{I}$	Finite set of edge identifiers, disjoint from V (multigraph permitted).
$\lambda_V : V \to \mathcal{P}(\mathbb{L})$	Node label map.
$\lambda_E : E \to \mathcal{P}(\mathbb{L})$	Edge label map.

$\rho_V : V \to (\mathbb{K} \rightharpoonup \mathbb{V})$	Node property map.
$\rho_E : E \to (\mathbb{K} \rightharpoonup \mathbb{V})$	Edge property map.
$\mathrm{src}, \mathrm{dst} : E \to V$	Edge endpoint functions; for an undirected edge their order is representational only.
$\rightarrow,\ \leftrightarrow$	Directed / undirected edge markers.
$\mathrm{dir} : E \to \{\rightarrow, \leftrightarrow\}$	Edge directedness (directed/undirected).
$\mathbb{R}_{\mathsf{cat}}$	Syntactically well-formed catalog object references of the GQL grammar.
$\mathbb{Q}$	Globally resolved references to catalog objects.
$\mathsf{Catalog}$	Abstract carrier of well-formed hierarchical GQL-catalogs.
$\mathcal{C}$	Catalog (logical collection of named GQL objects).
$\mathcal{X}$	Active context (catalog + working schema + working graph site).
WSR	Working schema reference (default parent for relative catalog-object references).
WG	Working graph site, occupied by a graph reference value whose referent is the working graph.
$\mathfrak{R}$	Resolution environment (used by typing judgments).
Loc	Local map of parameters and binding variables used while typing expressions; catalog procedure references use catalog resolution.
$\mathsf{resolve}_S(\mathcal{X}, r)$	Resolution of syntactic catalog reference $r \in \mathbb{R}_{\mathsf{cat}}$ to a globally resolved reference of sort S.
BT	Domain of descriptor-bearing binding tables.
$\mathsf{BindingTables} : \mathbb{Q} \rightharpoonup \mathsf{BT}$	Derived catalog projection from resolved binding-table references to binding-table objects.
SCX	Session context containing the session authorization identifier, principal, time zone, session schema and graph, parameters, list behavior, transaction and request context, and termination flag.
$\mathsf{ListBeh}$	Session list-behavior policy for offset/ordinal indexing, negative indices, and out-of-range behavior; it does not govern list unwinding.
$T, \mathrm{BT} \in \mathsf{BT}$	Binding-table objects.
$\mathrm{BT}_{\mathrm{unit}}$	Unit binding table containing one empty record; distinct from an empty binding table.
τ_{rec}	Record type of a binding table.
$\mathrm{fields}(\tau_{\mathrm{rec}})$	Declared field-name set of record type τ_{rec}.
$\mathrm{Rec}_{\tau_{\mathrm{rec}}}$	Set of records inhabiting record type τ_{rec}.

$T = \langle \tau_{\text{rec}}, B, \mathit{ord}, \mathit{pcs} \rangle$	Binding table represented by its record type, finite record multiset, ordering component, and preferred column-name sequence.
$B \in \mathcal{M}_{\text{fin}}(\text{Rec}_{\tau_{\text{rec}}})$	Underlying finite multiset of record occurrences.
$\text{supp}(B)$	Support of bag B: the set of elements with positive multiplicity.
$\mathbf{1}[P]$	Indicator of proposition P: 1 if P holds and 0 otherwise.
ord	Either the unordered marker or an ordering of the record occurrences of B.
pcs	Preferred column-name sequence: not set or a permutation of the table's column names.
$\text{bag}(T) = \langle \tau_{\text{rec}}, B \rangle$	Order-insensitive bag projection; it is not the complete binding-table object.
$\text{cols}(T)$	Column set of binding table T.
C_{new}	Newly introduced columns (e.g., by a pattern).
$\mathsf{PPat}, \mathsf{GPat}, \mathsf{GPBT}$	Sets of well-formed path patterns, graph patterns, and graph pattern binding tables.
$\Gamma \in \mathsf{GPat}$	Graph pattern, including an optional `WHERE` clause but not a `YIELD` clause.
$\Delta_{\text{gpbt}} \in \mathsf{GPBT}$	Graph pattern binding table: a graph pattern with an explicit or implicit `YIELD` clause; the subscript distinguishes it from an update delta.
$\text{vars}(\Gamma), \text{vars}_{\text{exp}}(\Gamma)$	Declared and exposed graph-pattern variables, including non-value subpath markers.
$\text{vars}_{\text{bind}}(\Gamma), \text{vars}^{\text{bind}}_{\text{exp}}(\Gamma)$	All and exposed value-bearing node, edge, and path variables.
$\text{D}_{\text{expo}}, \text{D}_{\text{ref}}$	Exposure degree; and reference degree, which is undefined for non-referenceable subpath variables.
$\mathsf{bind}(h)$	Binding extracted from match h and restricted to variables exported by the pattern.
$\mathsf{GPBT}_{\Delta_{\text{gpbt}}}(T)$	Graph-pattern binding-table operator for graph pattern binding table Δ_{gpbt}.
$\mathsf{MatchStmt}$	Abstract domain of match statements.
$\mathsf{MatchOp}, \mathsf{MatchBlock}$	Abstract domains of match operators and match blocks.
$\mathcal{Q} \in \mathsf{MatchBlock}$	Match-block metavariable, distinguished from bags $\mathcal{B}$ used by the binding-table algebra.
$[\![\mathbf{Match}(\Delta_{\text{gpbt}})]\!]_{\mathsf{match}}$	Denotation of `MATCH` as a binding-table transformer.
$\sigma_{\varphi},\ \pi_X$	Selection and projection operators.
$\bowtie$	Bag natural join.
$\uplus$	Bag union, adding record multiplicities.

$\mu \sim \nu,\ \mu \sqcup \nu$	Compatibility and merge of record mappings.
$\mathcal{B}_1 \bowtie_\theta \mathcal{B}_2$	Guarded bag join: natural join followed by filtering with guard θ.
$\triangleleft_\theta$	Left outer join with guard θ.
$\mathsf{OptExpand}_{\mathcal{F}}(T)$	Per-record optional expansion of T by correlated binding-table transformer $\mathcal{F}$.
$\mathsf{Out}_{\mathcal{F}}(\mu)$	New-column output of correlated transformer $\mathcal{F}$ for one incoming record occurrence μ.
$\mathrm{unnest}_{n:L}(T)$	`FOR`/list-unnesting operator, binding n to each list element occurrence in field L.
$\mathrm{hdr}(\mathcal{B})$	Variables in the declared record type of bag-valued binding-table representation $\mathcal{B}$.
$\pi_{\mathrm{L}},\ \ominus,\ \mathsf{pad}_{\mathrm{R}}$	Left-variable projection, bag difference on left projections, and null-padding of right-only variables.
$\mathcal{E}$	Static environment for variables, parameters, functions, and relevant catalog objects.
$\mathsf{Dyn} = (G, \mu_o)$	Dynamic subquery environment: current graph and current outer-row mapping.
WALK	Path mode: walk (default).
TRAIL	Path mode: trail.
ACYCLIC	Path mode: acyclic path.
SIMPLE	Path mode: simple path.
$M, \mathsf{mode}, \mathsf{pm}$	Generic path-mode parameters.
$\mathsf{Path}(G)$	Set of finite graph paths in G, represented by alternating node/edge sequences.
PathVal	Carrier of material GQL path values, which encapsulate path-element reference lists.
$w \in \mathsf{Path}(G)$	Graph path or RPE witness.
$p \in \mathsf{PathVal} \cup \{\mathsf{null}\}$	Material or null path value.
$\mathsf{ok}_M(w)$	Predicate that graph path w satisfies path mode M.
$\mathrm{head}(w), \mathrm{tail}(w)$	First- and last-node accessors for a graph path or RPE witness w.
$\mathrm{len}(w)$	Edge-step length of a graph path or RPE witness w.
$\mathsf{PEL}(p), \mathsf{len}(p)$	Path-element list and edge length of path value p.
$\mathsf{nodes}(p), \mathsf{edges}(p)$	Node- and edge-list accessors for a path value.
$\mathsf{head}(p), \mathsf{tail}(p)$	First- and last-node accessors for a path value.
$p \circ q$	Concatenation of path values whose boundary node references have the same referent.

Σ_L	Finite alphabet of edge-label names used by the auxiliary RPE calculus.
r, R	Regular path expressions.
ε	Auxiliary zero-edge regular path expression.
$\delta,\ S_\delta$	Step predicate and graph-induced ternary step relation.
$\psi : \mathsf{Path}(G) \to \{\top, \bot\}$	Boolean path predicate used to guard an auxiliary regular path expression.
$[\![r]\!]_G^{\mathsf{mode}}$	Bag denotation of r over graph G under the selected path mode.
$\cup_{\mathsf{ms}},\ \cup_{\mathsf{set}}$	Multiset and set alternation of regular path expressions.
$\bowtie_{\bowtie}$	Bag join on a shared endpoint with path-witness concatenation.
$\mathsf{Der}(R, \mathsf{pm})$	Successful auxiliary derivation trees for R under pm.
$\chi_{\mathrm{sp}}(d),\ \equiv_{\mathrm{sp}}$	Subpath-marker trace of derivation d, and equality modulo the same endpoint/path/marker trace.
$\mathsf{Cand}(R, \mathsf{pm})$	Candidate set obtained by quotienting successful derivations while retaining multiset-alternation marker distinctions.
$\Pi_{u,v}(\mathsf{Cand}(R, \mathsf{pm}))$	Endpoint-partition subset of $\mathsf{Cand}(R, \mathsf{pm})$ for fixed endpoints $u, v \in V$.
s	Selective path-search selector; κ is reserved for GQLSTATUS codes.
$\mathsf{SPSP}_s(\cdot)$	Selective path-search prefix operator with selector s.
ALL PATHS	Non-selective path-search prefix: return all paths; it is not an SPSP.
ANY_N PATHS	SPSP form: return any N paths.
$\mathsf{SHORTEST}_N$ PATHS	SPSP form: return N shortest paths.
$\mathsf{SHORTEST}_N$ GROUPS	SPSP form: return N shortest groups (partition-wise).
Ω	Abstract store state used by expression and query evaluation.
η_g, η_r	Global and row environments.
$\rightsquigarrow \subseteq \mathbb{T} \times \mathbb{T}$	Partial implicit-cast relation.
$\langle \Omega, \eta_g, \eta_r \rangle \vdash E \Downarrow u$	Expression-evaluation judgment, producing a value or error.
Var	Universe of variables used by expression and graph-pattern semantics.
$\mu : \mathsf{Var} \rightharpoonup \mathbb{V}$	Partial variable binding.
$[\![\phi]\!]_{G,\mu}^{\mathsf{tv}}$	Three-valued denotation of predicate ϕ.
$\top$	True.
$\bot$	False.
U	Unknown (Kleene 3-valued logic).
$\mathbb{B}_3 = \{\top, \bot, \mathsf{U}\}$	Three-valued Boolean domain (true/false/unknown).

$\preceq,\ \prec$	Applicable weak order on comparable values and its strict part.
$\approx_{\mathrm{nd}}$	Recursive GQL not-distinct relation used by grouping and duplicate treatment.
$K = \langle k_1, \ldots, k_s \rangle$	Ordered sequence of grouping keys; $\langle\rangle$ denotes global aggregation.
$\kappa_K(r)$	Grouping-key tuple extracted from record r in the order fixed by K.
$(B_i)_{i \in I}$	Pairwise disjoint group subbags induced by not-distinct grouping-key tuples.
$B_i \sqsubseteq_{\mathsf{bag}} B$	Subbag relation: every record multiplicity in B_i is at most its multiplicity in B.
F	Aggregator mapping a finite input bag to one aggregate result.
$\gamma_{\vec{g};\vec{h}}(B)$	Combined grouping and grouped-projection operator.
$\mathrm{Enum}_{\mathrm{impl}}(B)$	Implementation-dependent enumeration of an unordered bag's record occurrences.
$\mathrm{ORDER\ BY}_{\vec{o}}(B)$	Ordered binding table obtained using sort-specification list $\vec{o}$.
$\mathrm{OFFSET}_o(S)$	Sequence slicing by offset o.
$\mathrm{LIMIT}_\ell(S)$	Sequence slicing by limit ℓ.
$\mathbb{T}_{\mathrm{prop}} \subseteq \mathbb{T}$	Property value types supported by the implementation's conformance claim.
$\mathbb{T}_{\mathrm{scal}}, \mathbb{T}_{\mathrm{coll}}$	Selected scalar and collection fragments of $\mathbb{T}_{\mathrm{prop}}$.
$[\![\tau]\!] \subseteq \mathbb{V}$	Denotation of supported property value type τ (admissible values).
$\mathsf{LIST}\langle\tau\rangle$	Homogeneous list type constructor.
$\mathsf{NodeRef},\ \mathsf{EdgeRef}$	Reference value types whose referents are nodes and edges, respectively.
$\mathsf{Opt}(\tau)$	Meta-level shorthand for a value type containing null; not a GQL type constructor.
$\sqsupseteq$	Widening preorder on declarations/types (based on denotations).
$c : [\![\tau_0]\!] \rightharpoonup [\![\tau_1]\!]$	Partial cast between value types.
$\mathbb{N}_N$	Universe of node-type names.
$\mathbb{N}_E$	Universe of edge-type names.
$\mathbb{N}_G$	Universe of graph-type names.
$\mathbb{N}_{\mathsf{graph}}$	Universe of graph-entry names.
NDesc	Reduced book-level node-type descriptor representation $\langle N, L_N, \mathsf{ptype}_N \rangle$ contained in a graph type.

ETypeDesc	Reduced book-level edge-type descriptor representation $\langle \mathsf{ET}, L_E, \mathsf{kind}_E, \mathsf{endSpec}_E, \mathsf{ptype}_E \rangle$ contained in a graph type.
Σ	Reduced book-level structural graph-type component $\langle \mathsf{NDecls}, \mathsf{EDecls}, \mathsf{NodeKLS}, \mathsf{EdgeKLS} \rangle$.
$\mathsf{NDecls}, \mathsf{EDecls}$	Finite node- and edge-type maps inside Σ, not top-level catalogs.
$\mathsf{NodeKLS}, \mathsf{EdgeKLS}$	Node and edge key-label-set dictionaries inside Σ.
GTypeDesc	Reduced book-level graph-type descriptor representation $\langle \Sigma, \mathcal{I}_{\mathsf{GT}} \rangle$.
GraphEntry	Book-level abstraction $\langle G, \mathsf{GT}^?, \mathcal{I}_G \rangle$ of a stored-graph descriptor: graph, optional constraining graph-type reference, and direct constraints.
$\mathsf{GE} = \langle G, \mathsf{GT}^?, \mathcal{I}_G \rangle$	Active graph entry used by conformance, constraint-validation, and data-modification rules.
none	Marker that a graph entry has no constraining graph type.
Graphs, GraphTypes	Derived catalog projections for graphs and graph types after standard reference resolution.
GT	Globally resolved graph-type reference.
$\mathsf{resolve}_{\mathbf{GraphType}}$	Applied to $(\mathcal{C}, \mathsf{GT})$, resolves a graph type to its static descriptor Σ_{GT} and constraint dictionary $\mathcal{I}_{\mathsf{GT}}$.
$\mathsf{ptype}_T : \mathbb{K} \rightharpoonup \mathbb{T}_{\mathrm{prop}}$	Static property signature of type descriptor T.
$v :: N,\ e :: \mathsf{ET}$	Conformance judgments: node v conforms to node type N, and edge e conforms to edge type ET.
$\mathcal{I}_{\mathsf{GT}}, \mathcal{I}_G$	Named constraint dictionaries owned by a graph type and directly by a graph, respectively.
$\mathcal{I}_{\mathrm{direct}}(\mathsf{GE})$	Range of the constraint dictionary defined directly on graph entry GE.
$\mathcal{I}_{\mathrm{active}}(\mathsf{GE})$	Direct constraints of graph entry GE together with constraints of its constraining graph type, when present.
Sch	Schema state extracted from the catalog for the schema-evolution model.
absent	Sentinel for the absence of a property; it is not a GQL value.
CDesc	Carrier of key- and unique-constraint descriptors.
$\chi = \langle P, \bar{c} \rangle$	Key specification: deterministic constraint graph pattern and key value component sequence.
$\mathbf{t}_\chi(b)$	Key tuple obtained from binding b according to key specification χ.
$\approx$	Recursive GQL not-distinct relation on values and, pointwise, key tuples.

$\mathsf{nullContaining}(v)$	Predicate that value v recursively contains the null value.
$\mathsf{CBNF}(C), \mathsf{CSNF}(S)$	Deterministic normal forms of constraint body C and complete constraint specification S.
σ storeAssignableTo τ	Applicable GQL store-assignment compatibility check from source type σ to target type τ.
$\Sigma_{\text{typing}}(\mathsf{GT}^{?})$	Graph-type-conformance clause for an optional constraining graph-type reference.
$[\![\mathsf{GE}]\!]$	Graph-type-conformance clause and direct key/unique constraints of graph entry GE.
$\mathcal{T}$	Finite typing environment for binding variables in the effect judgments.
$\mathfrak{E}$	Carrier of static effects.
$\varepsilon_{\mathsf{eff}}$	Static effect set; the empty effect denotes write-purity in this metamodel.
e_{eff}	Primitive effect element.
$\Sigma; \mathcal{T} \vdash S : \varepsilon_{\text{eff}}$	Effect typing judgment for a query or data-modifying statement.
ok	Meta-level well-formedness judgment for primitive actions.
$\mathsf{peff}(a)$	Primitive-effect extractor for an action a.
$\sqcup_{\text{eff}}$, $\bigsqcup_{\text{eff}}$	Binary and finite indexed joins in the effect algebra.
$e_1 \leq_{\text{eff}} e_2$	Effect ordering: e_2 is a less specific abstraction of e_1.
$\mathsf{MayConflict}(\varepsilon_{\mathsf{eff},1}, \varepsilon_{\mathsf{eff},2})$	Static conflict-approximation predicate between effects.
$\top_\Lambda$	Unknown or unconstrained label-set abstraction.
$\mathbf{CreateV}[\Lambda_V]$	Primitive effect: create a vertex (with label-set parameter Λ).
$\mathbf{DeleteV}[\Lambda_V]$	Primitive effect: delete a vertex (with label-set parameter Λ).
$\mathbf{CreateE}[\Lambda_E]$	Primitive effect: create an edge (with label-set parameter Λ_E).
$\mathbf{DeleteE}[\Lambda_E]$	Primitive effect: delete an edge (with label-set parameter Λ_E).
$\mathbf{V}, \mathbf{E}$	Sort tags: vertex/edge (used in effect constructors).
$\mathbf{Set}(X, k)$	Primitive effect: set property k on sort $X \in \{\mathbf{V}, \mathbf{E}\}$.
$\mathbf{ClearProps}(X)$	Primitive effect: clear all properties on sort $X \in \{\mathbf{V}, \mathbf{E}\}$.
$\mathbf{RemProp}(X, k)$	Primitive effect: remove property k on sort $X \in \{\mathbf{V}, \mathbf{E}\}$.
$\mathbf{AddLbl}(X, \ell)$	Primitive effect: add label ℓ on sort $X \in \{\mathbf{V}, \mathbf{E}\}$.
$\mathbf{RemLbl}(X, \ell)$	Primitive effect: remove label ℓ on sort $X \in \{\mathbf{V}, \mathbf{E}\}$.
$\mathcal{K}$	Implementation configuration record (implementation-defined choices).
G_{cur}	Current working graph in data-modification rules; distinct from the acronym CG (Conceptual Graph).
$\mathsf{RefTy}_V, \mathsf{RefTy}_E$	Reference value types for nodes/edges (implementation-defined).

$\mathsf{LabMin}_V, \mathsf{LabMax}_V$	Supported min/max node-label counts.
$\mathsf{LabMin}_E, \mathsf{LabMax}_E$	Supported min/max edge-label counts.
$\mathsf{PropMax}_V, \mathsf{PropMax}_E$	Supported maximum property counts for nodes/edges.
Δ	Coalesced update delta of compatible primitive mutations.
$\oplus$	Apply a delta to a graph/store state (state update operator).
$\mathsf{insNode}(v, \Lambda, P)$	Primitive mutation: insert node v with labels Λ and properties P.
$\mathsf{delNode}(v)$	Primitive mutation: delete node v.
$\mathsf{insEdge}(e, \Lambda, P, u, v, d)$	Primitive mutation: insert an edge with labels, properties, endpoints, and directedness d.
$\mathsf{delEdge}(e)$	Primitive mutation: delete edge e.
$\mathsf{setPropV}, \mathsf{setPropE}$	Grounded setters for vertex/edge properties (template grounding).
$\mathsf{clearPropsV}, \mathsf{clearPropsE}$	Grounded operations that clear all vertex/edge properties for a set-all-properties item.
$\mathsf{delPropV}, \mathsf{delPropE}$	Grounded deleters for vertex/edge properties (template grounding).
$\mathsf{addLabelV}, \mathsf{addLabelE}$	Grounded add-label operations (vertex/edge).
$\mathsf{delLabelV}, \mathsf{delLabelE}$	Grounded remove-label operations (vertex/edge).
$\mathsf{ground}_\mu(\cdot)$	Grounding of an update atom by evaluating expressions under binding μ.
$\mathsf{WS}_\mu(U)$	Per-binding write-set (list of grounded operations produced by template U under μ).
$\Delta_U := \boxplus_{\mu \in \mathcal{M}} \mathsf{WS}_\mu(U)$	Staged delta obtained by folding the write-set coalescer over bindings.
$\boxplus$	Deterministic partial write-set coalescer, distinct from bag union and diagnostic disjoint sum.
$\mathsf{ok}_{\mathsf{GE}}(G, \Delta)$	Predicate that the post-state satisfies the graph's direct constraints and conforms to its constraining graph type, when present.
PDesc	Compact lookup-augmented representation $\langle p, \overline{x_j : \tau_j}, \overline{x'_m : \tau'_m = d_m}, \tau_{\mathrm{res}}, \mathsf{kind} \rangle$ of a named-procedure catalog entry.
$\mathbf{BT}\langle \overline{z_i : v_i} \rangle$	Material procedure binding-table result type with named, typed result fields.
τ_{res}	Procedure result type; for a `CALL`-statement, a material type $\mathbf{BT}\langle \overline{z_i : v_i} \rangle$ or the empty type.

$\overline{e}$ assignableTo $(\overline{\tau}, \overline{\tau'})$	Componentwise assignment-compatibility check for actual procedure arguments and mandatory/optional parameter types.
$\mathsf{sig}(\mathbf{PDesc})$	Core procedure signature derived from a named-procedure descriptor.
$\mathsf{Procs} : \mathbb{Q} \rightharpoonup \mathbf{PDesc}$	Catalog projection mapping qualified procedure references to named-procedure descriptors.
$\mathsf{resolve}_{\mathbf{Proc}}(\mathcal{X}, p)$	Resolution of procedure reference p in active context $\mathcal{X}$.
$\mathsf{Tx} = \langle u, \mathrm{iso}, \overline{P}, \mathsf{SP} \rangle$	Transaction: session authorization identifier, isolation choice, sequence of procedure executions, and optional extension state SP for savepoints outside the GQL core.
$\Delta(B)$	Update sequence (delta) produced by executing program/block B.
$\diamond$	Ordered composition of deltas.
$\mathsf{Committed}(\Omega')$	Successful commit outcome with resulting store state Ω'.
$\mathsf{RolledBack}(\Omega)$	Successful rollback outcome leaving the original store state.
$\mathsf{Aborted}(\kappa)$	Failed transaction outcome carrying status code κ.
$\Omega_{\mathrm{init}}, \Omega_{\mathrm{final}}$	Initial/final store states in a serializability argument.
ρ	Permutation witnessing serializability of committed transactions.
Ω_i^{base}	Base store state visible to statement S_i; snapshot selection outside the serializable core is implementation-defined.
$\Delta_{<i} = \Delta_1 \diamond \cdots \diamond \Delta_{i-1}$	Transaction-local cumulative delta preceding statement S_i.
$\Omega_i^{\mathrm{vis}} = \Omega_i^{\mathrm{base}} \oplus \Delta_{<i}$	Store state read by statement S_i.
$\mathbb{AID}$	Universe of authorization identifiers.
$\mathbb{ROLE}$	Universe of roles.
$\mathbb{OBJ}$	Universe of authorization targets (complete catalog or data objects in the core model).
$\mathbb{PRIV}$	Universe of privileges; modeled as a finite poset.
$\langle \mathbb{PRIV}, \leq \rangle$	Privilege poset (dominance order).
$\bigsqcup P$	Join (least upper bound) of a finite privilege set $P \subseteq \mathbb{PRIV}$.
$\mathcal{Z}$	Authorization state (set of role/grant facts).
$\mathsf{hasRole}(u, r)$	Fact: authorization identifier $u \in \mathbb{AID}$ has role r.
$\mathsf{grant}(g, x, \varpi, o)$	Fact: grantor g grants privilege ϖ on object o to grantee x.
$\mathsf{eff}(u, o)$	Effective privilege of authorization identifier u on object o.
$\Theta; \mathfrak{S} \vdash S : \tau \triangleright \Pi_S$	Authorization typing judgment producing privilege obligations, with binding and static schema/type contexts.
$\Pi_S \subseteq \mathbb{PRIV} \times \mathbb{OBJ}$	Obligation set (required privileges on objects).

$(u, o) \in \mathsf{available}$	Complete GQL-object o is available to authorization identifier $u \in \mathbb{AID}$.
txMode	Current transaction access mode, RO or RW.
$\varpi_{\mathsf{match}},\ \varpi_{\mathsf{insert}}$	Representative privilege tokens used in obligation rules (match/insert).
Err	Carrier of abstract exception-condition tokens.
$e_{\mathsf{err}} \in \mathsf{Err}$	Exception-condition token used by the diagnostic metamodel.
$\mathsf{GQLStatus}$	Opaque carrier of GQL-status objects.
$\zeta,\ \overline{\zeta}$	Primary GQL-status object and finite list of additional GQL-status objects.
$\mathsf{Fail}(e_{\mathsf{err}}, \zeta, \overline{\zeta})$	Failed outcome: exception condition and status objects, with no result.
$\amalg$	Disjoint sum used in diagnostic result domains.
$\mathsf{RegularResult} = \mathbb{V} \amalg \mathsf{BT}$	Carrier of regular results: values or binding tables.
$\mathsf{Result} = \mathsf{RegularResult} \amalg \{\mathsf{omitted}\}$	Carrier of regular and omitted results.
$\mathsf{Succ} = \{\, \mathsf{Ok}(R, \zeta, \overline{\zeta}) \mid R \in \mathsf{Result} \,\}$	Carrier of successful outcomes with their result and status objects.
$\mathsf{Res} = \mathsf{Succ} \amalg \mathsf{Fail}$	Execution-outcome carrier combining successful and failed outcomes.
$\mathbb{CH}$	Alphabet of status-code characters.
$\mathbb{CLASS}$	Status "class" strings (length 2).
$\mathbb{SUBCLASS}$	Status "subclass" strings (length 3).
$\mathbb{STATUS} \subseteq \mathbb{CH}^5$	Status-code space (GQLSTATUS codes).
$\kappa \in \mathbb{STATUS}$	A status code identifying a condition.
$\kappa_{\mathsf{ok}},\ \kappa_{\mathsf{ok_omitted}}$	Status codes `00000` (successful completion with a regular result) and `00001` (successful completion with an omitted result).
$\kappa_{\mathsf{warning}},\ \kappa_{\mathsf{no_data}}$	Status codes `01000` (completion with a warning) and `02000` (completion with an empty result).
$\cdot$	String concatenation.
$\kappa = c_\kappa \cdot s_\kappa$	Unique factorization into class and subclass parts.
$\mathrm{class}(\kappa)$	Extract class part of status code κ.
$\mathrm{subclass}(\kappa)$	Extract subclass part of status code κ.
$\mathrm{cat}(\kappa)$	Category function for status codes.
$\mathbf{S}, \mathbf{W}, \mathbf{N}, \mathbf{I}, \mathbf{X}$	Status categories (successful, warning, no-data, informational, exception), as used by $\mathrm{cat}(\kappa)$.
$\mathbb{STATUS}_{\mathsf{compl}}$	Completion-condition code subset.
$\mathbb{STATUS}_{\mathsf{exc}}$	Exception-condition code subset.

$\mathbb{STATUS}_{\mathsf{info}}$	Informational-condition code subset.
$\mathbb{STATUS}_{\mathsf{std}}$	Standard-assigned status-code subset.
$\mathbb{STATUS}_{\mathsf{impl}}$	Implementation-defined status-code subset.
$\mathsf{code} : \mathsf{Err} \to \mathbb{STATUS}_{\mathsf{exc}}$	Total mapping from error tokens to exception-condition GQLSTATUS codes.
$\mathsf{NLOGSPACE}, \mathsf{NP}$	Nondeterministic logarithmic-space and nondeterministic polynomial-time classes.
$\mathsf{PTIME}, \mathsf{PSPACE}$	Polynomial-time and polynomial-space classes.
$\mathsf{2EXPTIME}$	Double-exponential-time complexity class.
$\mathcal{ALCI}$	Description logic $\mathcal{ALC}$ extended with inverse roles.
$\mathcal{ALCQ}$	Description logic $\mathcal{ALC}$ extended with qualified number restrictions.
$\mathcal{ALCQI}$	Description logic $\mathcal{ALC}$ extended with qualified number restrictions and inverse roles.
$\#\mathsf{Datalog}$	Fully compositional graph-to-graph language used in the cited extension proposal.

Index

A

Abort, 62, 199
Absent property, 44
Access control, 63, 202
Active constraint set, 59, 177
Active context, 39, 186
Acyclic path, 45
Add label, 57
Additional status code, 65, 210
Admissibility filter, 46
Aggregate function, 55, 131
Aggregator, 55, 131
Alternation, 47, 82
Anonymous path, 46
Anti-join, 52
Approximate LIMIT, 55, 140
Assignment, 38
Atomicity, 62, 195
Authorization, 63, 202
Authorization check, 63, 204
Authorization identifier, 63, 202
Authorization state, 63, 203
Authorization target, 63, 202
Authorization-identifier-indexed semantics, 64, 205
AVG, 132

B

Bag (multiset), 40
Bag cardinality, 40
Bag difference, 51
Bag intersection, 51
Bag semantics, 51
Bag union, 51
Base type, 39
Base universes, 70
Binding, 35
Binding table, 41, 86
Binding table descriptor, 41
Binding table type, 41
Binding variable, 41
Body-equivalence, 58
Boolean type, 40
Bound insert element pattern, 56
Boundary variable, 49
Bounded repetition, 48
BT, 68, 86
Byte string type, 40

C

CALL, 190
Candidate set, 46, 127
Cast, 44, 96
Catalog, 38, 149, 185
Catalog entry, 38, 185
Catalog object, 38
Catalog-modifying procedure, 60
Character string type, 40
Closed graph, 37
Closed record type, 40
Coalescing, 57, 152
Collation, 54, 140
COLLECT_LIST, 132
Collection type, 40, 165

Column, 41
Column name-equal, 51
Column-combinable, 51
Command, 35, 196
Commit, 61, 197
Common type, 51
Comparable values, 39
Comparison, 42
Compatible mappings, 35
Completion condition, 64, 209
Complexity, 221
Composite, 71
Composite query, 53, 108
Computed path, 45
Concatenation, 47, 82, 115
Concurrency error, 66, 211
Condition, 64, 209
Conditional exposure, 49
Conformance, 37
Connective, 42, 99
Constraining graph type, 37, 169
Constraint, 37, 176
Constraint body, 58
Constraint descriptor, 58
Constraint dictionary, 37
Constraint enforcement, 59, 177
Constraint error, 66, 211
Constraint graph pattern, 58
Constraint normal form, 58
Constraint scope, 58
Constraint specification, 58
Constraint-preserving execution, 60
Constructed value type, 41
Continuous query, 218, 222
Correlated optional expansion, 52
Correlated subquery, 53, 105
COUNT, 132
CREATE GRAPH, 149
CREATE GRAPH TYPE, 23, 171
CRPQ, 28

D

Data, 34
Data integration, 223
Data type, 34
Data type descriptor, 39
Data-modification primitive, 56
Data-modification statement, 55
Data-modifying procedure, 60
DDL, 22, 171
Declaration, 60
Declared type, 38
Deduplication, 54, 93
DELETE, 26, 149
Delete property, 57
Delta, 57, 155
Dependent-object error, 56
Descriptor, 36
Destination node, 36
DETACH DELETE, 26, 150
Diagnostic choice, 66, 211
Diagnostic record, 65
Diagnostics, 64
Different edges match mode, 49
Direct graph constraint set, 59, 177
Directed edge, 36
Directionality, 36
Distinct, 93
Distinct values, 40
DML, 24
Document databases, 18
DQL, 26
Duplicate control, 54
Dynamic constraint validation, 59, 161
Dynamic environment, 53
Dynamic error, 65, 211
Dynamic union type, 41

E

Edge (relationship), 36
Edge label, 37
Edge reference value, 41
Edge type, 43, 166
Edge type descriptor, 43, 167
Edge variable, 48
Effect, 59
Effect algebra, 59
Effect soundness, 60
Effect system, 59, 178
Effective column name sequence, 41
Effective privilege, 63, 203
Effectively bounded group, 50
Effectively unbounded group, 50
Element pattern, 47
Element type, 43
Element variable, 48
Empty binding table, 51
Endpoint, 36
Endpoint partition, 46, 127
Endpoint reattachment, 56
Endpoint specification, 43
Endpoint structural consistency, 43
Environment, 53
Epsilon path expression, 47
Error class, 65, 211
Error token, 42
Evaluation, 35
Evaluation environment, 53
Evaluation judgment, 42, 96
Evaluation order, 97, 141
Exact LIMIT, 55, 140
EXCEPT, 110
EXCEPT ALL, 110
Exception condition, 64, 209
Execution, 35
Execution context, 38
Execution outcome, 65, 208
Explicit guard, 51
Expression, 42
Expressiveness, 221
Extend, 52
External procedure, 60

F

Failed outcome, 65, 208
Fallback query conjunction, 54, 111
Field, 40
FILTER, 27
FINISH, 136
Finiteness, 48
First-committer-wins, 62
FOR, 104
Function, 42

G

GenAI, 218
Generative AI, 218
Global environment, 53
Global filtering, 50
Global group, 55, 131
Global object identifier, 35
Globally identifiable object, 36
Globally resolved reference, 36
GPBT, 87
GQL graph, 37, 68
GQL procedure, 60
GQL-directory, 38
GQL-object, 35
GQL-object type, 39
GQL-program, 34
GQL-request, 35
GQL-schema, 38
GQL-status object, 65
Grant, 63, 203
Grantee, 63, 203
Grantor, 63, 203
Graph, 36

Graph descriptor, 37
Graph element, 36
Graph entry, 38, 169
Graph expression, 56
Graph pattern, 48, 74
Graph pattern variable, 49, 76
Graph pattern YIELD clause, 49, 87
Graph property, 217
Graph reference value, 41
Graph schema, 44
Graph type, 44, 168
Graph type descriptor, 44, 168
Graph type violation, 37
Graph-entry conformance, 37, 170
Graph-pattern binding table, 50, 87
Graph-selection header, 56
Graph-to-graph query, 214, 223
Graph-type conformance, 59, 170, 177
Ground operation, 57
Grounding, 57
GROUP BY, 30, 133
Group list value type, 40
Group variable, 50
Grouping, 55, 131
Grouping function, 55, 131
Grouping key, 55
Grouping set, 55
Guarded expression, 48
Guarded join, 52

H

HAVING, 133
Head of a path, 45
Header, 51
Homomorphic matching, 50
Hyperedge, 217

I

Idempotent label addition, 57
Identifier, 35, 185
Identifier allocation, 56
Immaterial value type, 40
Implementation configuration, 39
Implementation-defined, 35
Implementation-defined code, 65
Implementation-dependent, 35
Implicit cast, 44
Incident edge, 56
Informational condition, 65, 209
Inheritance, 44
Injectivity guard, 51
Inline GQL procedure, 60
INSERT, 24, 146
Insert element pattern, 56
Insert graph pattern, 56, 146
Insert path pattern, 56
Integrity constraint, 58
Integrity constraint violation, 59
Interior variable, 49
Interoperability, 223
INTERSECT, 110
INTERSECT ALL, 110
Inverse step, 47
Isolation level, 62, 198
Isomorphic matching, 51

J

JMESPath, 17
Join, 52
JSON, 17, 18
JSONPath, 17

K

KEEP clause, 49
KEY, 174, 175
Key, 58
Key constraint, 58, 174
Key label set, 44, 166
Key specification, 58, 174
Key tuple, 58, 174

Key value component, 58, 174
Key value component specification, 58, 174
Key-value databases, 18
Keyed element type, 44
KLS, 166

L

Label, 37
Label atom, 47
Label set, 37
Label test, 43
Label vocabulary, 44
Left outer join, 52, 102
Left-associative composition, 54
Length group, 46, 128
LET, 27
LIMIT, 30, 93, 139
Linear data-modification statement, 56
Linear query statement, 53
LINQ, 17
List, 72
List unnesting, 52
List value, 40
List value type, 40
List-behavior policy, 39, 188
Local generation, 50
Loop, 36
Lore, 12
Lorel, 12

M

Malformed path, 45
Mapping, 35
MATCH, 25–27, 87
Match, 49
Match mode, 49, 74
Matching semantics, 49, 82
Material data type, 39
Material value, 39
MAX, 132
Merge, 35
Metadata management, 224
MIN, 132
Mixed graph, 37
Mode filter, 45
Multi-path binding, 49, 74
Multigraph, 37
Multiple edges, 36
Multiple property assignments, 57
Multiplicity, 40
Multiset alternation, 47

N

N1QL, 18
Name, 35
Name resolution, 38, 187
Name-compatible join, 52, 93
Named procedure descriptor, 61, 189
Namespace, 38
Natural join, 52, 93, 101
Natural joinability, 52
No unbound writers, 60
No-data condition, 64
Node (vertex), 36
Node reference value, 41
Node type, 43, 166
Node type descriptor, 43, 166
Node variable, 48
Node/Edge predicate, 47
Non-distinctness, 58
Non-local reference, 50
Not-exposed variable, 49
NULL, 97
Null value, 34
Null-containment, 58
Null-intolerant predicate, 42, 99
Null-tolerant predicate, 42, 99
Nullable value type, 40

Nulls policy, 54, 139
Numeric type, 40

O

Object, 35
Object availability, 64, 205
Object Exchange Model, 12
Object Query Language, 12
Obligation judgment, 63, 203
Obligation set, 63, 203
OFFSET, 139
Omitted result, 65
Open graph, 37
Operation, 35
Optimization, 222
Optimizer ordering restriction, 60
Optional match, 52, 102
OQL, 12
ORDER BY, 30, 93, 139
Ordered binding table, 41
Ordering, 54, 93, 101, 139
Ordering metadata, 54
OTHERWISE, 111

P

Padding, 52
Pagination, 55, 139
Parameter, 60
Partial function, 67
Path, 71, 73, 114
Path accessor, 45, 115
Path binding, 46, 74
Path element list, 45
Path expression, 47
Path in a graph, 45
Path length, 45
Path mode, 45, 78, 114
Path pattern, 46
Path prefix, 46
Path primary, 48, 81
Path restrictor, 46
Path value, 45
Path value construction, 45, 116
Path value type, 45
Path variable, 46
Path witness, 47, 123
Path-value realizability, 45
Path-variable degree, 50, 81
Pattern matching, 48
Peer ordering, 54, 140
Pipelining, 53
Plain delete, 56
Post-aggregation filtering, 55
Powerset, 67
Predicate, 42
Predicate atom, 47
Preferred column name sequence, 41
Primary base type, 39
Primary catalog object, 38
Primary object, 36
Primary status code, 65, 210
Primitive effect, 59
Primitive mutation, 57, 161
Primitive query statement, 53
Primitive result statement, 55, 136
Principal, 63
Privilege, 63, 202
Privilege kind, 63
Privilege level, 63
Privilege obligation, 63, 203
Privilege poset, 63
Procedure, 35, 189
Procedure body, 61
Procedure invocation, 61, 190
PRODUCT, 132
Projection, 52, 92, 101
Propagation rule, 50
Property, 37
Property graph, 37, 68

Property key, 37
Property map, 37
Property set, 37
Property signature, 43
Property type, 43, 164
Property value, 37
Provisional change, 62

Q

Qualified name, 35
Quantification, 82
Quantifier, 47
Query block, 53, 104
Query part, 54
Query procedure, 60
Query processing, 222
Questioned path-pattern primary, 50, 82

R

RBAC, 218
RDF, 15
Read stability, 56
Read-only transaction, 61
Read-then-update composition, 56
Read-write transaction, 61
Record, 40, 72
Record type, 40
Reduced match, 49, 77
Reduction, 50
Reference degree, 50, 80
Reference resolution, 61
Reference value, 41
Reference value type, 41
Registered subject, 58
Regular path expression, 47, 120
Regular result, 65
Relational subquery, 53, 106
Relationship type, 43
REMOVE, 157
Remove label, 57
Repeatable elements match mode, 49
Requirement, 58
Resource error, 66, 211
Restrictive path mode, 48
Result multiplicity, 51
Result sum type, 65, 208
RETURN, 27
Reversal of expressions, 48
Role, 63, 202
Role-based access control, 218
Rollback, 61, 197
Row environment, 53
Row-producing subquery, 53
RPE, 120
Runtime error, 42

S

SAME, 99
Satisfaction relation, 59, 177
Savepoint, 61
Scalar subquery, 53, 106
Scalar type, 40, 164
Scalars, 71
Schema, 216
Schema evolution, 44, 172
Schema state, 44, 172
Schema versioning, 44, 172
Scope predicate, 58
Search prefix, 46
Secondary object, 36
Security, 63
Selection, 52, 93, 101
Selective path pattern, 50
Selective path search prefix, 46, 78, 79, 128
Selective stage, 50
Semantic constraint clause, 59
Semi-join, 52
Serializability, 62, 198

Session context, 39, 188
Session options, 39
Session state, 39, 188
SET, 25, 152
Set alternation, 47
Set operator, 54, 108
Set property, 57
Set quantifier, 54
Set semantics, 51
Setification, 51
Shadowing, 53
Short-circuit failure propagation, 65, 208
Short-circuiting, 42
Shortest-path semantics, 127
Side effect, 35
Signature, 60, 189
Simple path, 45
Singleton variable, 50
Site, 38
SKIP, 93
Slicing, 55, 93, 140
Snapshot, 62, 198
Sort direction, 54, 139
Sort key, 54, 139
Sort specification, 54, 139
Sort specification list, 54, 139
Source node, 36
SPARQL, 16
SPSP, 79, 127
SQL, 12, 17
Standard-defined code, 65
START TRANSACTION, 197
Statement, 35
Statement effect typing, 60
Statement sequencing, 62, 196
Statement snapshot, 62
Static conflict approximation, 60
Static environment, 53
Static error, 65, 211
Static judgment, 42
Static property signature, 43
Status class, 64
Status code, 64, 209, 211
Status domain, 64, 209
Status precedence, 65, 211
Status subclass, 64
Step predicate, 47, 122
Step relation, 47, 122
Store assignment, 38
Store state, 38, 96
Stored path, 46, 217
Streaming graph query, 218, 222
Strict function, 42
Strict interior variable, 49
Structural consistency, 43
Subpath variable, 48
Subquery, 53, 104, 214
Subtype, 44
Subtyping, 44
Successful completion, 64
Successful outcome, 65, 208
SUM, 132
Supported property value type, 40
Syntactically resolved reference, 36
System-generated identifier, 56

T

Tail of a path, 45
Target reference, 56
Temporal type, 40
Temporary element variable, 49
Theoretical analyses, 215
Theta-join, 52
Three-valued logic, 42, 97
Totalization, 42
Trail, 45
Transaction, 61, 195
Transaction access mode, 61, 196

Transaction characteristic, 61
Transaction control command, 61, 196
Transaction model, 61, 195
Transaction processing, 223
Transaction rollback condition, 64
Transaction snapshot, 62
Truth value, 40, 97
Type, 34
Type constructors, 71
Typing, 42, 96
Typing contexts, 71

U

Unbound variable, 35, 98
Unbounded repetition, 48
Unconditional exposure, 49
Uncorrelated subquery, 53, 105
Undirected edge, 36
Union, 93, 109
UNION ALL, 109
UNIQUE, 174
Uniqueness constraint, 59, 174
Unit record type, 40
Unit table, 51
Unknown truth value, 42
Unordered binding table, 41
UnQL, 12
UNWIND, 94
Update atom, 56, 153
Update context, 56, 153
Update pipeline, 56, 151
Update template, 57, 153
USE, 187

V

Validation predicate, 59, 178
Value, 34
Value type, 39
Value-level path concatenation, 45, 116
Variable, 35
Variable exposure, 49, 80
Variable hygiene, 53
Vector search, 218
Vector type, 40
View, 214
Visibility, 62, 195

W

Walk, 45
Warning, 64
Weak order, 42
Well-formedness, 48
WHERE, 27
Whole-query denotation, 53, 106
Widening, 44
Working graph, 39
Working graph site, 39
Working record, 41
Working schema reference, 38
Working table, 41
Write check, 64, 205
Write conflict, 57, 152
Write set, 57, 152
Write-purity, 60, 178
Write-write conflict, 62

X

XML, 17
XPath, 16
XQuery, 16

Y

YIELD, 190

Autor:
Dominik Tomaszuk
ORCID: 0000-0003-1806-067X

Recenzenci:
prof. dr hab. Krzysztof Stencel (UW)
dr hab. Piotr Wiśniewski, prof. UMK

Opracowanie graficzne:
Dominik Tomaszuk

Redakcja i korekta:
Julia Gierczak

Skład i redakcja techniczna:
Dominik Tomaszuk

© Copyright by Uniwersytet w Białymstoku, Białystok 2026

ISBN 978-83-7431-884-6

DOI: 10.15290/GQL_Essentials_2026

Wydanie publikacji sfinansowano ze środków Wydziału Informatyki Uniwersytetu w Białymstoku

Wydawnictwo Uniwersytetu w Białymstoku

ul. Ciołkowskiego 1M, 15-245 Białystok

tel. 85 745 71 20, 85 745 70 59

wydawnictwo@uwb.edu.pl

https://wydawnictwo.uwb.edu.pl

