Semantic Data Modeling, Graph Query, and SQL, Together at Last?

Jeff Shute
Google, Inc.
jshute@google.com

Colin Zheng
Google, Inc.
colinz@google.com

Romit Kudtarkar
Google, Inc.
romitk@google.com

ABSTRACT

Semantic data models express high-level business concepts and metrics, capturing the business logic needed to query a database correctly. Most data modeling solutions are built as layers above SQL query engines, with bespoke query languages or APIs. The layered approach means that semantic models can't be used directly in SQL queries. This paper focuses on an open problem in this space – can we define semantic models in SQL, and make them naturally queryable in SQL?

In parallel, graph query is becoming increasingly popular, including in SQL. SQL/PGQ extends SQL with an embedded subset of the GQL graph query language, adding property graph views and making graph traversal queries easy.

We explore a surprising connection: semantic data models are graphs, and defining graphs is a data modeling problem. In both domains, users start by defining a graph model, and need query language support to easily traverse edges in the graph, which means doing joins in the underlying data.

We propose some useful SQL extensions that make it easier to use higher-level data model abstractions in queries. Users can define a "semantic data graph" view of their data, encapsulating the complex business logic required to query the underlying tables correctly. Then they can query that semantic graph model easily with SQL.

Our SQL extensions are useful independently, simplifying many queries – particularly, queries with joins. We make declared foreign key relationships usable for joins at query time – a feature that seems obvious but is notably missing in standard SQL.

In combination, these extensions provide a practical approach to extend SQL incrementally, bringing semantic modeling and graph query together with the relational model and SQL.

1 INTRODUCTION

Schemas in real-world databases are very complex, often with thousands of tables, and they require complex business logic and domain knowledge to query correctly.

Semantic modeling tools are a common solution. Users describe business logic in a data model and then query that model with some API, which gets translated into SQL queries underneath. This helps tame the complexity for reporting and business intelligence (BI) tools.

But there's something missing: Semantic models work well in tools that use them, but they don't help users who still write SQL queries. This work is about bringing semantic data modeling into SQL, so the data models used by front-end tools can also be used directly by SQL users.

1.1 SQL and semantic models at Google

SQL is ubiquitous at Google. Data is stored in many systems. Some look like traditional databases and some don't, but they're all queried with SQL, using F1[11] as a federated query engine.

Most of Google's large businesses (Search, Ads, etc) use semantic modeling systems to help manage their complex business logic. ([12] is one example.) All these users struggle with the incompleteness of modeling layers built above SQL. They all end up needing to replicate the same logic in SQL. They all want a solution where business logic can be written once and shared.

GoogleSQL[13] is the SQL dialect and implementation shared across most SQL systems at Google, including F1, BigQuery, Spanner and Procella, and the open-source release, ZetaSQL[5]. GoogleSQL is a shared, reusable component, enabling these systems to share the same SQL dialect.

This work extends GoogleSQL with new query features and data modeling capabilities, building on existing support for querying structured data. So far, these features are being used in F1 by multiple teams to replace legacy data modeling solutions.

Our new SQL features aren't specific to GoogleSQL and could be supported in any SQL dialect.

2 BACKGROUND

2.1 Semantic data modeling

Many problems stem from the fundamental disconnect between the way data is stored and the way business users need to query it.

Storage schemas are complex for various reasons. They're often designed around cost, performance, or other practical constraints, particularly those imposed by transactional write patterns. They're often normalized (storing each piece of data exactly once) to help ensure data integrity. Schemas designed for efficiency and data integrity aren't generally optimized for simple querying. And over time, schemas become more complex, accumulating workarounds for legacy artifacts as a business evolves.

Storage schemas usually don't capture higher-level concepts business users need to query. Key concepts like "net revenue" or "active customer" likely don't exist as single columns with intuitive names, and often require complex expressions combining data from multiple columns (and sometimes tables) to compute correctly.

A semantic data model gives users a higher-level schema that represents meaningful business objects and metrics, and makes them easy to query, hiding the physical mappings and business logic underneath.

Semantic data models also hide the complexity inherent with SQL joins. Since SQL joins return all pairs of rows that join, most join outputs include duplicated rows from at least one of their inputs. This duplication makes it difficult to compute aggregate metrics from multiple source tables in the same query without double-counting. Data modeling tools have to account for this, generating complex query logic to ensure metrics are aggregated correctly.

There are many industry solutions for semantic data modeling, and we have experience building several for users at Google. Generally, these are built as layers in front of a SQL query engine. Semantic models are defined with some custom configuration language, and queried with a custom API or query language. The models are generally not queryable with SQL, although some systems support embedding data model queries into SQL queries, typically with a table-valued function (TVF) doing an external query written in a different query language. For example, Snowflake's semantic views[3] are defined outside the SQL data model and then queried with an embedded mini-language inside a TVF-like operator.

Semantic data models are often queried from interactive user interfaces that provide visualizations like dashboards or pivot tables. Building a query in these UIs typically involves picking columns (from a list of meaningful attributes and metrics), and optionally adding filters or aggregations.

These UIs make queries easy to build, and the data model underneath guarantees queries are correct, by construction. Users can only create queries that "make sense". The model blocks queries that aren't correctly answerable. Query results can also be presented to clearly show what question was answered. For example, when showing metrics in graphs or pivot tables, each data item can be labeled with its canonical name from the data model.

Writing SQL queries against the raw data is much more difficult and doesn't provide the same guarantees, but there are still good reasons users need data to be queryable with SQL.

2.1.1 *Analysis cliff problem.*

Semantic data models are a great solution up to a point, but for unusual queries or more complex analysis, it's sometimes necessary to fall back to SQL.

Once users reach the limit of what can be expressed in an interactive BI tool, falling back to SQL can mean starting over, working directly with the raw data, without using any of the business logic encapsulated in the data model. We call this the **analysis cliff problem**.

Replicating the model's business logic in raw SQL can be difficult, and keeping logic consistent across multiple implementations in different languages is a maintenance burden.

Query UIs often support exporting the SQL query used for a visualization, which can be a useful starting point, but the generated queries are often too complex for humans to understand or modify incrementally, forcing the user to reimplement business logic from scratch against the raw tables.

2.1.2 *Data silo problem.*

Since data models are created and queried with bespoke languages, interoperability is lacking. The higher-level models work well in specific tools that support their query API, but any other tools still have to work with lower-level SQL. Transactional applications and data pipelines also typically read and write tables directly, and can't use the data model defined on top.

The same data is often consumed with many tools, including SQL, Python for data science, map-reduce-style bulk processing pipelines, and maybe natural language tools and AI agents. It's difficult for these tools to use shared business logic when the data model is only accessible using a custom API designed for interactive analysis.

2.1.3 *All-or-nothing problem.*

Defining a semantic data model takes significant effort, working with experts across a business to discover, analyze, and curate commonly used business concepts and then encode them into a data model.

When finished, users get a great experience using the data model and interactive query tools.

Before it's sufficiently complete, its incremental value is limited. If the data model can't answer most user questions, users don't find it valuable, and still need to write SQL queries.

Because data modeling tools are non-standard and don't interoperate, organizations need to commit to specific tools. That commitment, plus the cost of data modeling, means adopting semantic modeling tools typically requires an organization-wide decision. It's hard for individual users or subteams to use semantic models on their own, or get incremental value from them, without a broader effort to model all useful business metrics.

The high onboarding cost has a corresponding lock-in factor. Data modeling tools are expensive to adopt and also expensive to escape.

2.2 Natural language query

Converting natural language to SQL (NL2SQL) is very difficult. While language models can generate executable SQL, generating *correct* queries reliably against non-trivial schemas is unsolved.[10]

There are many challenges, starting with table and column names that provide inadequate descriptions of their content. Significant domain knowledge is necessary to understand the data and how to query it correctly. Important business metrics often show up only in query logic and aren't visible in the schema, so non-trivial analysis requires something providing this additional semantic context.

With natural language query, it's important there's a way to validate the answers. This is especially true in data analysis contexts, where the answer might be just a number, a graph, or a table. How can users know where the numbers came from, or that they're correct? Business users cannot be expected to validate the SQL. Even SQL experts struggle to understand generated queries, and validating their business logic requires specific domain expertise.

So "NL2SQL" may be focused on the wrong problem. When the goal is to answer natural language questions (NL-to-answers?), SQL is an intermediate stage and implementation detail, not the goal. In fact, standard SQL is a low-level language that's difficult to use correctly for question answering.

Semantic models are a better target for natural language queries. Analysis queries over semantic data models ensure queries are correct by construction, and can describe clearly what result they're showing (e.g., by showing graphs using canonical labels for metrics – see 2.1).

2.3 Graph query

Graph queries are often written in GQL (Graph Query Language) or SQL/PGQ (Property Graph Query). In SQL/PGQ, the `GRAPH_TABLE` table-valued-function supports an embedded subset of GQL syntax, inside SQL queries.[1] These graph queries always return a table, which can be further processed with standard SQL.

Property graphs are declared using `CREATE PROPERTY GRAPH`, which defines a graph view producing nodes and edges from SQL tables. Columns are exposed as properties on nodes and edges.

Graph queries start with the `MATCH` operator, which searches for paths matching a requested pattern (of nodes and edges), returning a table that has one row for each match. The GQL operators that follow `MATCH` are essentially SQL's relational operators with alternate syntax.

Here's a simple example. See [1] for more.

```
SELECT *
FROM GRAPH_TABLE(AccountGraph
  MATCH (a1:Account) -[outflow:Transfers]-> (:Account)
  RETURN a1.ID, SUM(outflow.amount) AS total_outflow
  GROUP BY a1.ID
)
```

2.4 Structured data

Real-world data includes objects with hierarchical structure. Traditional schema normalization requires splitting complex objects across many tables, and then using joins to reassemble data at query time. Schemas storing objects as structured data types need far fewer tables. They're often easier to use and more efficient, for both transactions and queries[11].

Many SQL systems support structured data types. GoogleSQL supports `ARRAY`s, `STRUCT`s for tuples, Protocol Buffers for strongly-typed objects, and JSON for dynamically-typed objects.

GoogleSQL query syntax includes:

- Reading fields of objects:
  `SELECT object.field1.field2`
- Iterating over array elements:
  `FROM UNNEST(object.field.array_field) [WITH OFFSET]`
- Implicitly unnesting paths without writing `UNNEST`:
  `FROM path` and `JOIN path` work without `UNNEST(path)`.
- Traversing nested arrays in one step with generalized paths:
  `FROM array1.array2.field` iterates over elements found by traversing the path depth-first.

`JOIN UNNEST` maps structured data cleanly into the relational model, producing result tables equivalent to those produced by joins between standard tables. Then the rest of the query proceeds as usual from that output table.

2.5 Joins in SQL

Writing joins is more difficult than it should be, given their centrality in SQL. Joins require a condition, typically written with an `ON` or `USING` clause on a `JOIN`, or in the `WHERE` clause.

[1]In the rest of this paper, when referencing GQL, we mean either GQL or SQL/PGQ. Both support identical syntax in our implementation.

In practice, there's usually only one meaningful way to join any pair of tables. Those useful join conditions are often declared in the schema already, as `FOREIGN KEY`s.

For some reason, SQL only uses foreign keys as transactional integrity constraints. SQL *doesn't make foreign keys usable in queries*. How much time have users spent writing out the same join conditions repeatedly? Or made mistakes in join conditions?

In software engineering, the **Don't Repeat Yourself (DRY) principle** says that recurring logic should be written once and then referenced, rather than copy/pasted. Repeating the same code many times is an anti-pattern. SQL users apply this in schema design, normalizing data so there's a single source-of-truth. But SQL doesn't provide a way to do the same for join logic.

The syntax for unnesting arrays, `JOIN UNNEST(array)`, doesn't need a join condition. The condition is implicit, between the parent row and the array value's elements. Wouldn't it be nice if we could use similar syntax for joining through foreign keys?

3 SOLUTION – SQL BUILDING BLOCKS

Our overall solution uses several new SQL features as building blocks. These features can be implemented separately and each provide usability benefits or useful query capabilities on their own.

Using these features together provides the big-picture solution, bringing semantic data models into SQL, making query patterns more uniform across domains, and improving overall SQL usability. Appendix A has more examples using these features.

3.1 Virtual columns

A **virtual column** is defined with an expression, which gets evaluated *at query time*, when the column is accessed.

Many systems support generated columns in `CREATE TABLE`. Typically, these compute an expression at insertion time and store the value[2].

While the specifications look similar, the semantics and usage are different. Virtual column expressions can include arbitrary logic referencing other columns from the same row, including using correlated subqueries to fetch data from other tables. Since they aren't stored, there's no cost to adding lots of them. Users can add virtual columns encapsulating business logic for any attributes or expressions commonly computed on a table. (For aggregate metrics, see measure columns in 3.4.)

As a simple example, on a table with a `BirthDate` column, a virtual column `Age` could be defined as `DATE_DIFF(CURRENT_DATE(), BirthDate, YEAR)`. Then users can write queries reading both stored and virtual columns:

```
SELECT Name, BirthDate, Age
FROM Students
```

The same outcomes can be achieved using views. A view is a named virtual table, where the entire table is defined with a query. Virtual columns add some flexibility:

[2]A few systems already support non-stored generated columns, but with some limitations. Postgres restricts them to single-row deterministic expressions, so they're limited to the same expressive power as stored columns[2]. SQL Server supports computed columns without that restriction[4].

- Virtual columns don't require separate tables. They can be added in the primary tables, and users can query them without rewriting queries to use views.
- Virtual columns can be added incrementally with ALTER statements. Views typically cannot be altered incrementally without setting an entirely new query.
- A virtual column expression can reference other columns, including other virtual columns (as long as there are no cycles).
- Virtual *fields* can also be added inside structured objects, with expressions computed using sibling field values.

In essence, virtual columns allow adding business logic to existing tables incrementally, while views require all-or-nothing definitions of new tables.

3.2 Join columns and ROW types

A **join column** is a virtual column representing a join, producing values representing rows from the joined table. We say these columns have a row type, which is written `ROW<TableName>`.

A join column for an N:1 lookup join produces a single row and acts like an object embedded in the row. It can be queried just like a column storing a denormalized copy of the target object. Reading N:1 join columns in expressions has LEFT JOIN semantics. If no joined row is found, fields under the join column are NULL.

A join column for a 1:N (or M:N) join acts like a column containing an unordered array of row objects. If no rows join, it acts like an empty array. Existing array scan syntax allows choosing INNER JOIN or LEFT JOIN as appropriate.

Join columns are read like objects (or arrays of objects) using dotted path notation. One path can conveniently traverse multiple join columns, without writing each join separately. Join conditions are implicit because they're bound into the join column.

This query joins five tables using standard syntax:[3]

```
SELECT n.n_name,
       SUM(li.l_extendedprice)
FROM Customer c
JOIN Orders o ON c_custkey = o_custkey
JOIN Lineitem li ON o_orderkey = l_orderkey
JOIN Nation n ON c_nationkey = n_nationkey
JOIN Region r ON n_regionkey = r_regionkey
WHERE r.r_name = "EUROPE"
GROUP BY 1
```

With join columns, the query can be written with just one JOIN, using path expressions to read columns from related tables:

```
SELECT c.Nation.n_name,          # Read through join column
       SUM(li.l_extendedprice)  # Aggregate joined elements
FROM Customer c
JOIN c.Orders.Lineitems li              # Multi-hop join
WHERE c.Nation.Region.r_name = "EUROPE"  # Multi-hop read
GROUP BY 1
```

3.2.1 *Inferring join columns.*

Many schemas already include FOREIGN KEY declarations. For example:

```
ALTER TABLE Orders
ADD FOREIGN KEY (o_custkey) REFERENCES Customer(c_custkey)
```

Join columns don't require declared or enforced foreign keys, but where they exist, they can be translated naturally into join columns. For the foreign key above, the default is:

- On `Orders`, an N:1 virtual join column `Customer`.
- On `Customer`, a 1:N virtual join column `Orders`, *joining in the reverse direction.*

By default, join columns are added on both tables, allowing *bidirectional* joins. The peer table name makes a good default column name, allowing table traversal just using those names. Making column names singular or plural makes join cardinality clear.

This behavior can be overridden: whether to add one or both, and what to call them. Alternate names are necessary when there are name collisions. For self-joins (in a table describing a tree), the join columns might be named `Parent` and `Children`.

3.2.2 *Defining join columns.*

Join columns can also be defined manually, with FOREIGN KEY-like syntax, or with an arbitrary subquery that fetches rows. Subqueries are powerful here. They could filter rows, join through multiple tables (e.g., M:N linking tables like `Partsupp` in appendix A.2), or use WITH RECURSIVE for tree traversal.

3.2.3 *Row types.*

A `ROW<T>` value references a row of table `T`, *including its virtual columns.* Virtual and join columns defined on `T` can be read transitively through the ROW.

ROWs are nonconcrete values that cannot be stored or returned in final query output, but they can be converted to concrete STRUCTs materializing selected physical and virtual columns.

The table a ROW points at may also have outgoing join columns, including joins back to the original source table. All these joins can be traversed like a recursive tree structure (with unbounded depth). Like virtual columns, joined rows are materialized lazily, only when they're referenced in the query, avoiding infinite tree expansion.

3.3 Horizontal aggregation

Standard aggregation combines values *across multiple rows* vertically. **Horizontal aggregation** combines values *across array elements within a single row.* (The term comes from graph query, for aggregation across the nodes or edges in a path.)

Standard SQL doesn't provide an easy way to aggregate array elements. Typically, joins are required, expanding rows, and then using GROUP BY to collapse back to a single row:

```
SELECT StudentID, AVG(g)
FROM Students s JOIN UNNEST(s.grades) g
GROUP BY StudentID
```

This can also be done in a scalar subquery:

```
SELECT StudentID, (SELECT AVG(g) FROM UNNEST(s.grades))
FROM Students s
```

Neither syntax is convenient, particularly when aggregating multiple arrays.

We introduce a new syntax to express horizontal aggregation directly, using UNNEST inside the aggregate function.

[3]Most example queries use the TPC-H schema. See Appendix A.2 and Figure 2.

```
SELECT StudentID, AVG(UNNEST(s.grades))
FROM Students
GROUP BY StudentID
```

This UNNEST also allows multi-hop paths traversing through multiple arrays, and to fields inside array objects. The same syntax works over all array-like objects, including join columns.

The query from 3.2 (which joins five tables) can be written using horizontal aggregation without writing any JOIN at all:

```
SELECT c.Nation.n_name,
       SUM(UNNEST(c.Orders.Lineitems.l_extendedprice))
FROM Customer c
WHERE c.Nation.Region.r_name = "EUROPE"
GROUP BY 1
```

3.3.1 More details. The short-form UNNEST syntax works well for single-argument aggregate functions. A more general syntax is needed for multi-argument functions or post-UNNEST expressions: `SELECT CORR(s.x, 2*s.y FROM UNNEST(path) AS s)`

By default, horizontal aggregation does aggregation over *both* rows and array elements. Doing both together is necessary for uncombinable aggregates like AVG or COUNT(DISTINCT).

Sometimes we want to just aggregate array elements, without aggregating across rows or making the query an aggregate query. We are still evaluating syntax options for how to request horizontal-only aggregation.

Could UNNEST be implicit, so we could just write SUM(<path>)? Maybe in some cases, but there are some ambiguities and backwards-compatibility issues. For example, COUNT(array_value) counts the number of non-null array values, while COUNT(UNNEST(array_value)) counts the number of non-null elements of those array values.

3.4 Measure columns

Measure columns are virtual columns encapsulating *aggregate expressions*. These columns have **MEASURE<T> types**, which cannot be returned in final query output because they don't have concrete values. They must be aggregated using the "default aggregate function" AGG, which has signature AGG(MEASURE<T>) -> T.

A measure definition is like a lambda function defined with an aggregation expression. When a measure value propagates through a query, its concrete evaluation is deferred until the measure value is aggregated using AGG. The aggregate expression is then computed over *distinct rows* from the source table that contributed measure values into each aggregation group.

3.4.1 Grain locking. Double-counting is a pernicious problem in aggregation after joins. Measure columns solve this using **grain locking**. For example, assume we've defined this measure column:

```
ALTER TABLE Orders
ADD COLUMN TotalRevenue AS MEASURE(SUM(o_totalprice));
```

This measure column is automatically grain-locked to the table it's defined on, Orders. It will always be aggregated as if it was computed on that table alone.

This query joins Orders and Lineitem, computing aggregates of columns from both tables.

```
SELECT o_orderdate,
       SUM(o_totalprice),    # Wrong: Double-counts Orders
       AGG(o.TotalRevenue),  # Correct aggregation
       SUM(l_extendedprice)
FROM Orders o JOIN Lineitem ON o_orderkey = l_orderkey
GROUP BY o_orderdate
```

The SUM(o_totalprice) aggregation is incorrect because the join duplicates Orders rows, making the SUM double-count.

The measure aggregation AGG(o.TotalRevenue) is grain-locked so it counteracts the row duplication and gives a correct result that avoids double-counting.

Semantically, aggregating a grain-locked measure (with AGG) includes each distinct row from its source table at most once, so the aggregation avoids double-counting, regardless of joins in the query.

3.4.2 Implementation details. Measure expressions can reference any columns from the table on which they're defined. Measure *values* can be implemented as STRUCT values binding in all columns referenced in the expression, plus the row's unique key. These STRUCT values propagate through the query as usual, including through grouping and into aggregation. To compute the aggregate value of a measure, the STRUCT values are first deduplicated using the bound keys, and then the aggregate expression can be computed using the other bound values.

[15] discusses measure types in much more detail, including some alternate implementation strategies.

4 SOLUTION – SEMANTIC DATA MODELING

4.1 Semantic data graphs

We'll now show how we can use the features described above as part of a more complete solution for semantic data modeling.

We build the semantic data model as a graph, which we call a **semantic data graph**. The graph includes:

- Nodes (tables) – semantic objects
- Edges (join columns) – relationships between objects
- Attributes (columns, possibly virtual) – properties of objects
- Measures (measure columns) – aggregate metrics

Attribute columns and N:1 join columns can be queried using the same syntax. Both return objects which can have their own nested fields. In a path like customer.Nation.n_name, Nation could be a stored column, a virtual column, or an N:1 join column.

Data warehouses often use star schemas with a single central fact table and many dimension tables. As a semantic data graph, the fact table would be the primary node. Most queries will start there, joining in some dimension tables for attributes.

In a more complex schema, there can be many interesting object types to query. A query can select one to start from (the first table in its FROM clause) and then traverse edges to other connected objects.

In the TPC-H schema for example, a query could start from one of the central object tables like Customer, Orders, or Part. It could also start from a dimension table like Nation for a query listing countries.

Since most join edges are bidirectional, a query could start from the table on either side of a join. Choosing the starting node has a subtle but important effect on query semantics. For example, a query starting from `Orders` and reading attributes of their `Customers` won't include any rows for customers with no orders. A query starting from `Customer` will include all customers, even if they have zero orders (assuming the query uses `LEFT JOIN`).

Optionally, users can "flatten" parts of the graph so most queries can be written as single-table queries with minimal path traversal. For example, from TPC-H's `Customer` table, the nation and region names are available as `Nation.n_name` and `Nation.Region.r_name`. A flattened `Customer` table could contain `r_name` and `n_name` directly as virtual columns. The joins would only be executed if the query uses those virtual columns.

This flexibility lets users optimize their semantic schema for easy querying. The same source tables can be mapped as a normalized view with many tables, a denormalized "one big table" view, or anything in between. Flattened views can even include aggregate metrics from joined tables since measure columns ensure correct aggregation, without double-counting.

A single-table flattened view can be queried with any tool that can generate SQL queries, making full use of the modeled business logic, without knowing about any new SQL features. Existing tools that work with structured data types can even use join columns, querying them as if they were arrays, using existing SQL syntax.

Using the graph model at query time is an incremental choice. If the original tables and columns are left visible, existing queries work as before, so the schema is backwards-compatible. Mixing semantic graph features (like virtual columns and edge traversals) with traditional joins and query logic also works. This allows a gradual transition to querying higher-level semantic models while still joining other raw data.

4.2 Building a semantic data graph

A semantic data graph can be defined incrementally, starting from existing tables or views (i.e., virtual tables).

Join columns (edges in the graph), measure columns, and other virtual columns can be defined in any order, building on each other. Their expressions can reference other virtual columns laterally, and can fetch columns from other tables by reading through join columns.

Semantic graph views can optionally prune what's exposed to users, hiding physical tables or columns in favor of the semantic definitions. This is useful when querying the physical schema is complex or error-prone, and using virtual columns encapsulating correct business-logic is preferred. Some users might be given access only to the semantic view, while others are allowed to query the underlying tables too.

4.2.1 Privacy applications. Filtered semantic views can be useful for privacy-safe queries. The semantic view can expose only the attributes and metrics that are safe to query without exposing personally identifiable information (PII). For example, user IDs might be hidden and unqueryable, while graph traversals can still use those IDs for joins.

Semantic views can also define measures that bind in privacy constraints. For example, measures might automatically apply K-thresholding (only showing data that occurs for at least K users) or differential privacy[19].

5 ANALYSIS

5.1 Logical and physical data independence

Logical and physical data independence are key concepts in data management.

Logical data independence means that queries are not affected by unrelated changes to the database schema, like adding columns.

Physical data independence means that queries are not affected by choices in physical storage, or physical execution plans. A database can store and use different storage formats and apply various indexing strategies. Then query optimizers rewrite execution plans, choosing physical query plans while preserving logical query semantics.

Semantic data modeling raises the abstraction level for databases, adding higher-level business concepts and relationships. For users querying business metrics, *how* to join the underlying SQL tables or compute metrics is not interesting. Users care *what* to read and *what* measures to compute, and should be able to express that declaratively. For example, asking "Get the Orders for this Customer" without specifying how.

Going further, semantic data models can decouple the queryable schema presented to users from the physical table structure. Consider some common table structures for analytic data:

- Normalized flat tables arranged in a star schema
- Normalized tables using structured data types (fewer tables)
- Denormalized tables with copies of prejoined objects

How much to use structured objects or denormalize data is a schema design choice. This choice can be made separately for physical storage (optimizing for efficiency and data integrity) and in a user-facing view optimized for queryability. Users might find denormalized schemas easier to query. Physical and logical schema design choices *should be independent*. With semantic data modeling, *they can be!*

Using join columns, syntax for querying through joins is *exactly the same* as syntax for querying nested objects (2.4). The same queries work against either schema. Likewise, virtual columns are queried exactly like regular columns. Choosing to store a value or compute it dynamically doesn't affect queries.

In effect, the semantic data graph is a new abstraction layer providing an object-based view over the entire database, including relationships between objects. That view could (optionally) be quite different from what's physically stored in the underlying tables.

5.2 Comparison to property graphs

Property graphs are defined over a database using `CREATE PROPERTY GRAPH` and then queried with GQL, optionally embedded in SQL/PGQ `GRAPH_TABLE` clauses. The property graph definition creates nodes and edges from existing tables, and adds attributes to both from existing columns.

There's significant overlap between property graph definitions and our semantic graph model definitions. Both define a graph view, exposing data objects as nodes and relationships as edges. Our design allows one graph schema definition to be shared, and then queried with either SQL or GQL.

In standard SQL/PGQ, property graphs must be defined explicitly, requiring significant setup work before graph queries are possible.

Inferring a default property graph from a semantic graph definition is possible, with nodes for every table and edges for known foreign keys.

Property graphs support properties on edges. They can be derived from M:N linking tables representing edges (like `Partsupp` in the TPC-H schema). When M:N linking tables occur in SQL schemas, they're more naturally mapped as intermediate nodes in the graph rather than as edges with properties. When a linking node doesn't have interesting properties itself, the semantic model can also include join columns that flatten the traversal, traversing between the source and destination tables in a single step.

Many property graph queries use `MATCH` with path patterns just using explicit edge traversals between specific node types. These are just joins. These queries use the graph mainly as a convenient way to do joins without writing join conditions. This is very similar to using join columns in our syntax.

Property graphs also make more general graph searches easy, with multi-step traversals through trees or graphs. The traversals are often run underneath using recursive self-joins, which can be expressed in SQL using `WITH RECURSIVE`.

As future work, it seems interesting to try bringing both styles of graph query closer together. We can share the same graph definition, defining virtual columns and measure columns usable in both SQL and GQL. Maybe we can add an operator like `MATCH` directly into SQL, making recursive edge traversal as easy as join column traversal.

A few disconnects remain to be solved between the standard property graph data model and the SQL data model. Property graphs are weakly typed compared to SQL tables. Property graphs generally assume that "all nodes", "all edges", and "all properties" are meaningful sets that can be queried. Those graph queries act like SQL queries doing `UNION ALL` over all available tables, with an assumption that everywhere the same column name occurs, it has the same type and meaning. This is problematic in non-trivial SQL-style schemas. Mapping between SQL and property graphs will work better with more strongly-typed property graphs, with node and edge labels providing a stronger "schema" of meaningful properties, with locally (not globally) unique names.

6 RELATED WORK

The *object-relational impedance mismatch problem* is long-standing: Standalone tables are a poor representation of real-world objects and the relationships between them. Many workarounds have been built, using abstractions like object-relational mapping (ORM) layers, trying to hide some of the complexity.

Two recent papers by Deshpande[7] and Dittrich[8] explore this topic, both describing gaps in the relational model and suggesting we replace it with a data model that better captures the graph-like relationships between objects.

Deshpande suggests elevating the core abstraction from the relational model to the entity-relationship (E/R) model, using a new system (*ErbiumDB*) designed with the E/R model at its core as its natively queryable abstraction. The key benefit is improved Logical Data Independence, decoupling user-visible schema from physical storage decisions.

Dittrich argues for a more radical change, discarding SQL and the relational and E/R models, and replacing them with a new Functional Data Model (FDM) that represents all data concepts as functions, from tuples up to full databases. The FDM is queried with a Functional Query Language (FQL), seamlessly embedded into application programming languages, without ORMs or a standalone textual query language. This new paradigm requires entirely new systems, contrasting with our focus on incremental, backwards-compatible SQL extensions.

Decades ago, work on object-relational databases (ORDs)[18] tried to solve similar problems. ORDs aren't popular now, but ideas like user-defined types were incorporated into mainstream databases, and other features appeared in the application layer with ORMs. ORDs supported object traversal using paths, similar to our join column syntax, but those features have mostly disappeared.

SQL:1999[9] also added object-relational features, including `REF` types which are similar to our `ROW` types in join columns, but these features are not widely supported or used today.

This paper considers related problems, and suggests an incremental solution, adding semantic modeling features to SQL, including modeling joins and making them usable in queries. This could be interpreted as an alternative practical realization of previous work on higher-level graph and object-relational abstractions.

Modern SQL dialects like GoogleSQL and SQL++[6] support structured data objects natively (see 2.4). Users today are comfortable working with objects in SQL. Graph modeling is a natural extension, and may be more accessible than past ORD approaches that haven't gained traction. Graph query is more widely used now, following the standardization of property graphs and GQL.

Oracle's *JSON Relational Duality Views*[14] are another notable industry solution to object-relational mismatch. These views dynamically map normalized relational tables to hierarchical JSON objects. Critically, duality views are *updatable*, transforming object updates back to transactional updates on the underlying relational tables. This greatly simplifies OLTP applications, removing the need for ORMs or other mapping layers. Our work focuses on semantic graph modeling more than object construction, and on analytical queries rather than transactions. Both solutions let users work with higher-level data models using objects that encapsulate the complexity of the underlying physical tables.

Previous work on pipe syntax[16] describes many historical pain points in SQL, and solves some of them. In particular, pipe syntax makes relational operators arbitrarily composable, reducing the need for subqueries. This work addresses another long-standing pain point in SQL syntax – the need to write every join manually.

Combining SQL and GQL, and doing semantic modeling inside SQL rather than in a layer above, continues the pattern described in [17]. Relational databases and SQL keep expanding to cover use cases that were initially done in other tools, providing a simpler and more unified solution for users.

7 FUTURE WORK

Teams at Google are starting to adopt features described in this paper, replacing semantic layers previously built *above SQL*. Early results are promising, but we will learn more as we get more experience with users. Our work so far has focused on query support, and used non-SQL configuration files for schema setup. We need a SQL DDL language to complete an end-to-end SQL solution. Extending `CREATE PROPERTY GRAPH` syntax seems promising and allows for a single graph definition queryable with both SQL and GQL.

Queries over semantic models prompt some interesting topics for query optimization. These queries must still perform at least as well as queries that could be generated against the underlying tables. Graph traversals can generate some unusual new plan shapes, including heavy use of correlated subqueries. Standard techniques like subquery decorrelation and common subexpression elimination are helpful. More study will uncover other opportunities for optimization. Some optimizations are actually easier with queries written against higher-level schema abstractions because the simpler query logic is easier to reliably pattern-match and then replace with precomputed data.

Driving materialization from the semantic data model is another promising area. The data model captures business logic for measures that will be frequently queried. Requesting materialization for selected measures should be easy, without repeating the business logic. Then queries should automatically start using precomputed data when possible. Automatic materialization is also promising.

Unifying semantic data graphs and property graph query (see 5.2) is worth exploring further. GoogleSQL already implements graph query and SQL in the same framework, sharing a unified type system, function library, and other details. While standard SQL/PGQ is a subset of GQL, GoogleSQL supports the full GQL language inside `GRAPH_TABLE`. In reverse, GoogleSQL also works inside GQL; expressions in GQL can use any SQL expression, including subqueries. We see SQL and GQL not as two languages, but instead as one common language containing multiple query operators.

Pipe syntax[16] makes it easy to extend SQL with new operators. A `MATCH` operator could bring GQL's functionality directly into SQL as a fully composable operator. Maybe it's not necessary to switch to GQL syntax to use graph operators at all. For example, while join columns allow discrete edge traversals like `employee.manager.manager.name`, maybe GQL-style paths like `employee -[manager]->* top_manager` could be supported directly in SQL, making recursive edge traversals easy too.

We hope to extend SQL to support OLAP-style query features that are easily expressed in languages like MDX but difficult in SQL. This includes querying "context-shifted"[15] and period-over-period measures like "previous year sales".

Extending DML and allowing writable semantic graph views is also interesting, and could follow the work in [14].

Attaching user-defined functions (UDFs) and procedures to graph nodes seems interesting. This could make a semantic graph look like an object-oriented database supporting actions on the objects. AI agents could use these functions against a transactional database, encapsulating simple actions that can be applied safely.

8 CONCLUSION

This paper introduced several useful SQL extensions:

- Virtual columns, encapsulating business logic.
- Join columns, which make foreign-key joins easy to use at query time.
- Horizontal aggregation, simplifying aggregation over arrays or joined data.
- Measure columns, encapsulating logic for aggregate metrics and ensuring they're aggregated without double-counting.
- *Semantic data graphs*, which use these features to build semantic data models and make them queryable in SQL.

These SQL improvements are independent and broadly useful. Simplifying joins helps everyone, regardless of how much additional data modeling they do.

Semantic data graphs defined for SQL are very similar to property graphs defined for GQL. Sharing a common graph model brings SQL and GQL together, letting users query their data graph using whichever syntax is most convenient.

Defining semantic data models inside SQL has several advantages over typical approaches using semantic layers built above SQL. Supporting SQL queries solves the analysis cliff and data siloing problems inherent with non-SQL data modeling languages and APIs. Models can be built incrementally and are backwards compatible, making adoption easier. Exposing the semantic model as virtual tables and columns means that even tools without any graph query support can still query the semantic model with standard SQL.

Enabling wider use of semantic data models, incrementally from SQL, brings numerous benefits. Queries using encapsulated business logic are shorter, simpler, and more reliably correct. This helps business users, and anyone else writing queries – particularly AI agents generating queries from natural-language prompts.

ACKNOWLEDGEMENTS

This work builds on previous work across many teams at Google: the GoogleSQL team and our query engine partners, the Spanner graph query team, the F1 unified data modeling team, Julian Hyde and John Fremlin for measures in SQL, Dave Otaguro, Karl Schnaitter and the Qubos team for work on data modeling theory and use cases, and user teams who've helped define the problems to solve.

REFERENCES

[1] [n. d.]. GQL overview (from Spanner). Retrieved 2025-11-22 from https://cloud.google.com/spanner/docs/reference/standard-sql/graph-intro
[2] [n. d.]. Postgres generated columns. Retrieved 2025-11-22 from https://www.postgresql.org/docs/18/ddl-generated-columns.html
[3] [n. d.]. Snowflake semantic_view reference. Retrieved 2025-11-22 from https://docs.snowflake.com/en/sql-reference/constructs/semantic_view
[4] [n. d.]. SQL Server computed columns. Retrieved 2025-11-22 from https://learn.microsoft.com/en-us/sql/relational-databases/tables/specify-computed-columns-in-a-table?view=sql-server-ver17
[5] [n. d.]. ZetaSQL. Retrieved 2025-11-22 from https://github.com/google/zetasql
[6] Michael Carey, Don Chamberlin, Almann Goo, Kian Win Ong, Yannis Papakonstantinou, Chris Suver, Sitaram Vemulapalli, and Till Westmann. 2024. SQL++: We Can Finally Relax!. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 5501–5510. doi:10.1109/ICDE60146.2024.00438
[7] Amol Deshpande. 2025. Beyond Relations: A Case for Elevating to the Entity-Relationship Abstraction. arXiv:2505.03536 [cs.DB] https://arxiv.org/abs/2505.03536
[8] Jens Dittrich. 2025. A Functional Data Model and Query Language is All You Need. arXiv:2507.20671 [cs.DB] https://arxiv.org/abs/2507.20671
[9] Andrew Eisenberg and Jim Melton. 1999. SQL: 1999, formerly known as SQL3. *SIGMOD Rec.* 28, 1 (March 1999), 131–138. doi:10.1145/309844.310075
[10] Avrilia Floratou et al. 2024. NL2SQL is a solved problem... Not! *(CIDR 2024)*.
[11] Bart Samwel et al. 2018. F1 query: declarative querying at scale. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1835–1848. doi:10.14778/3229863.3229871
[12] Gokul Nath Babu Manoharan et al. 2016. Shasta: Interactive Reporting At Scale. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1393–1404. doi:10.1145/2882903.2904444

[13] James C. Corbett et al. 2017. Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 331–343. doi:10.1145/3035918.3056103

[14] Shashank Gugnani et al. 2025. JSON Relational Duality: A Revolutionary Combination of Document, Object, and Relational Models. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD-/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 431–443. doi:10.1145/3722212.3724441

[15] Julian Hyde and John Fremlin. 2024. Measures in SQL. In *Companion of the 2024 International Conference on Management of Data (SIGMOD/PODS '24)*. ACM, 31–40. doi:10.1145/3626246.3653374

[16] Jeff Shute, Shannon Bales, Matthew Brown, Jean-Daniel Browne, Brandon Dolphin, Romit Kudtarkar, Andrey Litvinov, Jingchi Ma, John Morcos, Michael Shen, David Wilhite, Xi Wu, and Lulan Yu. 2024. SQL Has Problems. We Can Fix Them: Pipe Syntax In SQL. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4051–4063. doi:10.14778/3685800.3685826

[17] Michael Stonebraker and Andrew Pavlo. 2024. What Goes Around Comes Around... And Around... *SIGMOD Record* 53, 2 (Jun 2024), 21–37. https://db.cs.cmu.edu/papers/2024/whatgoesaround-sigmodrec2024.pdf

[18] Michael Stonebraker, Lawrence A. Rowe, Bruce G. Lindsay, Jim Gray, Michael J. Carey, Michael L. Brodie, Philip A. Bernstein, and David Beech. 1990. Third-generation database system manifesto. *SIGMOD Rec.* 19, 3 (Sept. 1990), 31–44. doi:10.1145/101077.390001

[19] Royce Wilson, Celia Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. 2020. Differentially Private SQL with Bounded User Contribution. 230-250 pages. doi:10.2478/popets-2020-0025

A EXAMPLES

A.1 Information schema

Many databases provide an `INFORMATION_SCHEMA` catalog with tables describing database metadata. Figure 1 shows the schema fragment describing tables, indexes and foreign keys.

Schema normalization requires spreading data across many tables. Queries often need to join several of these tables to reconstruct useful objects. All tables have multi-part primary keys with at least three parts (e.g. `table_catalog, table_schema, table_name`), so all joins require complex multi-column join conditions. These complicated joins make this schema verbose and annoying to query.

For example, reading a table's indexes, with details about the columns they point at, requires a query joining four tables, listing 10 columns in join conditions (or 20 if using `ON` instead of `USING`):

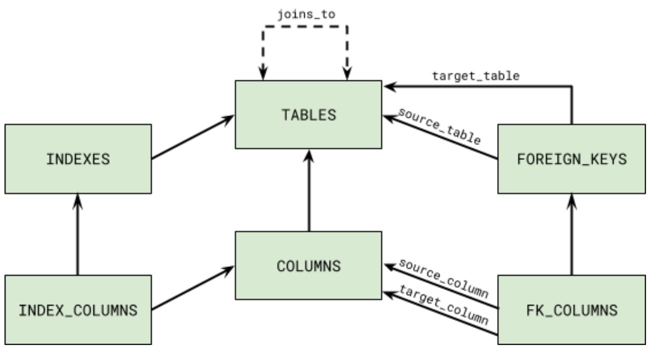

Figure 1: A simplified subset of an `INFORMATION_SCHEMA` schema, showing some tables and their relationships. The `joins_to` edge is a virtual many-to-many self-join through `FOREIGN_KEYS`.

```
SELECT table_name, index_name, column_name, column_type
FROM tables
JOIN indexes USING (table_catalog, table_schema, table_name)
JOIN index_columns USING (table_catalog, table_schema,
                          table_name, index_name)
JOIN columns USING (table_catalog, table_schema,
                    column_name)
```

Encapsulating these relationships as join columns in a graph model can significantly reduce the noise (and potential for mistakes!). Join columns encapsulate all those details, and allow other simplifications like multi-step joins:

```
SELECT table_name, index_name, column_name, column_type
FROM tables AS t
JOIN t.indexes AS i
JOIN i.index_columns.column AS c  # A two-step join
```

The `INFORMATION_SCHEMA` schema is a good example of the complexity of modeling real-world objects. The data does not fit neatly into flat tables. Normalization creates many tables, with complex relationships, including cyclic relationships.

When an application wants to read "the schema for table X", they need to join data from many tables. The full result cannot be read with a single query returning a flat result table because there are multiple plural sub-objects (columns, indexes, and foreign keys). Any flattened representation creates unwanted cross-products.

Returning a structured data object (like a protocol buffer or JSON object) works better because nested sub-objects can be listed in arrays. This is a good use case for SQL views providing those dynamically constructed objects, as in [14].

Concrete objects can represent slices of this data (like one table's schema) but cannot include the full graph structure, which allows traversal to all connected objects (e.g., joinable tables). Inlining all connected objects is impractical because connections expand infinitely. (The graph has cycles.) Treating the database as a graph allows each query to traverse or extract a relevant subgraph.

With a graph data model and our SQL extensions, it's simple to write a single query extracting many facts about a table:

```
SELECT
  table_name,

  # Count the columns, and list them.
  COUNT(UNNEST(t.columns)) AS num_columns,
  ARRAY_AGG(UNNEST(t.columns.column_name)) AS column_names,

  # Build an array with an object for each joined row.
  ARRAY_AGG(STRUCT(column_name, column_type)
    FROM UNNEST(t.columns)) AS column_names_and_types,

  # Count the columns that are primary keys.
  COUNT(UNNEST(t.columns) WHERE is_primary_key)
    AS num_pk_columns,

  # Count the indexes and foreign keys.
  # They're joined independently, avoiding cross-products.
  COUNT(UNNEST(t.indexes)) AS num_indexes,
  COUNT(UNNEST(t.foreign_keys)) AS num_foreign_keys,
  # Count distinct column_names, with a multi-step join.
  COUNT(DISTINCT UNNEST(
    t.indexes.index_columns.column_name)) AS indexed_columns

FROM tables t
GROUP BY table_catalog, table_schema, table_name
```

Our graph data model (Figure 1) also adds a self-join relationship. The `joins_to` join column encapsulates the many-to-many link between `TABLES`, joining through `FOREIGN_KEYS`. In SQL, this can be queried with a `JOIN`.

```
SELECT t1.table_name, t2.table_name
FROM tables AS t1
JOIN t1.joins_to AS t2
```

A query to find the shortest join path between Table1 and Table22 could be expressed in SQL using `WITH RECURSIVE`, but is simpler to express as a graph query using GQL:

```
MATCH p = ANY SHORTEST                  # Find shortest path p,
 (t1:tables {table_name: "Table1"}) # from a tables node t1,
   -[:joins_to]->+                      # traversing :joins_to
                                        #   edges 1+ times,
 (t2:tables {table_name: "Table2"}) # to a tables node t2.
RETURN t1.table_name, t2.table_name,
       path_length(p) AS num_join_steps
```

These queries illustrate why it's useful to have a semantic graph model, and to share the same model for both SQL and GQL queries, allowing queries to be written in whichever form (relational or graph) is most natural.

A.2 TPC-H schema

This example describes the definition and use of a semantic data graph over the well-known TPC-H schema shown in Figure 2.

We'll assume the `FOREIGN KEY` relationships are defined and that we've inferred join columns for each of them. By convention, we reuse the table names as join column names, using singular names for N:1 join columns (e.g. `Orders.customer`) and plural names for 1:N join columns (e.g. `Customer.orders`).

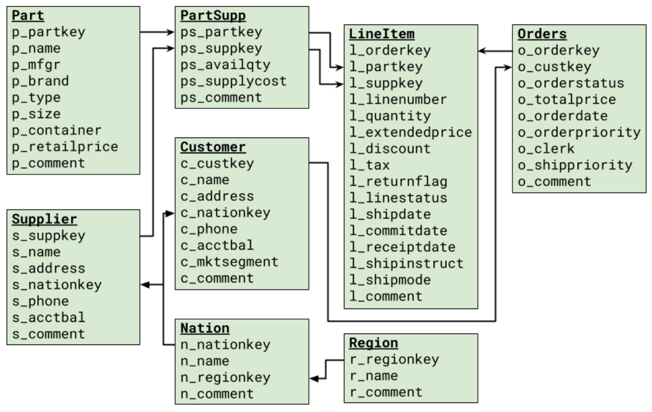

Figure 2: The TPC-H schema, showing tables, columns, and foreign key join relationships

Additional virtual columns can be added in the model using DDL syntax:

```
# Some virtual columns defined with expressions on Lineitem.
ALTER TABLE Lineitem
ADD COLUMN DiscountedPrice AS
             l_extendedprice * (1 - l_discount),
ADD COLUMN ChargedAmount AS DiscountedPrice * (1 + l_tax);

# Add measure columns for some aggregate metrics.
# These are "grain-locked" to aggregate relative to one
# table, and never double-count.
ALTER TABLE Orders
ADD COLUMN SumTotalPrice AS MEASURE(SUM(o_totalprice));

ALTER TABLE Lineitem
ADD COLUMN SumDiscountedPrice AS
             MEASURE(SUM(DiscountedPrice));
```

Figure 3 shows how queries can be simplified using those semantic model features.

Here is another example query using several semantic model features. It's written as a query on `Customer`, using join columns and horizontal aggregation to collect attributes and metrics from several related tables.

```
SELECT
  # Read attributes from this table.
  c_name,
  # Read attributes from linked dimension tables.
  nation.n_name,
  nation.region.r_name,

  # Aggregate metrics from this table and child tables.
  # Note that because we didn't write a JOIN in FROM, we
  # can collect metrics from several tables without
  # any double-counting.
  SUM(c_acctbal),
  COUNT(UNNEST(orders.lineitems) AS NumLineitems,
  MAX(UNNEST(orders.o_totalprice)),
  AVG(UNNEST(orders.lineitems.DiscountedPrice)),

  # A filtered horizontal aggregate.
  SUM(l_quantity FROM UNNEST(orders.lineitems)
      WHERE l_returnflag='R') AS NumReturnedItems
FROM
  # We scan Customer, producing one row per customer.
  Customer
WHERE
  # Filter on an attribute reached through a join column.
  nation.region.r_name = "EUROPE"
  # Filter on a horizontal aggregate of a child table.
  AND MIN(UNNEST(orders.o_orderdate)) < "2020-01-01"
GROUP BY
  # This GROUP BY prevents aggregation across customers.
  # We still get rows for customers with no orders, with
  # zeros or NULLs for their aggregates.
  c_custkey
```

Standard SQL query

```
# TPC-H Q5 before (plus an aggregate metric from TPC-H Q1)
# Note that within all the join conditions, there's a
# hidden extra condition that looks like a triangular
# join adding a condition checking that the customer's
# and supplier's nations are the same.
SELECT
  n_name,
  # Metrics have complex, overlapping formulas.
  sum(l_extendedprice * (1 - l_discount)) AS revenue,
  sum(l_extendedprice * (1 - l_discount) * (1 + l_tax))
    AS sum_charge
FROM
  customer,
  orders,
  lineitem,
  supplier,
  nation,
  region
WHERE
  c_custkey = o_custkey
  AND l_orderkey = o_orderkey
  AND l_suppkey = s_suppkey
  AND c_nationkey = s_nationkey  # Note the extra condition!
  AND s_nationkey = n_nationkey
  AND n_regionkey = r_regionkey
  AND r_name = 'AFRICA'
  AND o_orderdate
      BETWEEN '1994-01-01' AND '1994-12-31'
GROUP BY n_name
ORDER BY revenue DESC;
```

Query with semantic graph and SQL extensions

```
# TPC-H Q5 after
# Note the simplification of FROM and WHERE.
# It's also clearer in SELECT and WHERE that we're
# referencing the supplier's nation, not the customer's.

SELECT
  supplier.nation.n_name,
  # Metrics logic is encapsulated using virtual columns.
  sum(DiscountedPrice) AS revenue,
  sum(ChargedAmount)
    AS sum_charge
FROM


  lineitem



WHERE



  `order`.customer.c_nationkey = supplier.s_nationkey


  AND supplier.nation.region.r_name = 'AFRICA'
  AND `order`.o_orderdate
      BETWEEN '1994-01-01' AND '1994-12-31'
GROUP BY n_name
ORDER BY revenue DESC;
```

```
# A query aggregating columns from both orders and lineitem.
# Aggregations incorrectly double-count after we add joins
# that duplicate rows.
SELECT
  c_name,
  SUM(l_extendedprice * (1 - l_discount)),
  SUM(o_totalprice)  # Wrong: double-counts
FROM customer AS c
LEFT JOIN orders AS o ON c.c_custkey = o.o_custkey
LEFT JOIN lineitem AS li ON o.o_orderkey = li.l_orderkey
GROUP BY c_name
```

```
# A query aggregating measure columns.
# Measure columns are "grain-locked" to compute aggregates
# relative to a specific table, never double-counting.
SELECT
  c_name,
  AGG(SumDiscountedPrice),
  AGG(SumTotalPrice)  # Correct aggregation
FROM customer AS c
LEFT JOIN c.orders AS o
LEFT JOIN o.lineitem AS li
GROUP BY c_name
```

Figure 3: Comparing queries on TPC-H schema using standard SQL and semantic graph SQL