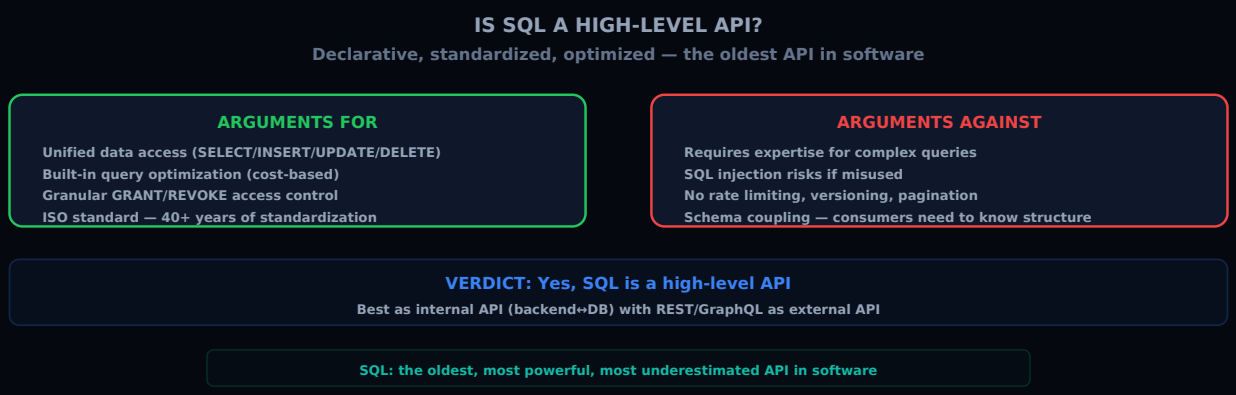
Is SQL a High-Level API?

Sylvain ARBAUDIE · May 18, 2025

SQL ARCHITECTURE API OPINION



The Uncomfortable Question

In modern software architecture, everything goes through APIs. REST, GraphQL, gRPC — applications communicate via well-defined, documented, versioned interfaces. But there is one API that predates all others, that is ubiquitous, and that nobody really calls an "API": **SQL**.

SQL is a standardized declarative language (ISO/IEC 9075) that allows interacting with structured data. You declare what you want, not how to get it. The database engine handles execution. That is, by definition, an application programming interface — an API.

So why does nobody treat SQL as an API?

SQL as API: Arguments For

Unified Data Access

SQL offers a single access point to complex data. Whether reading a row, joining ten tables, aggregating millions of rows, or bulk-modifying data, the interface is the same: SQL statements sent via a network protocol (the MariaDB / MySQL protocol, for example).

This is exactly what a REST API does: expose a unified access point to resources, with clear semantics (GET = read, POST = create, etc.). SQL does the same with SELECT, INSERT, UPDATE,

DELETE.

Built-in Optimization

When you call a REST API, it is your backend code that decides how to execute the request. When you send a SQL query, it is the database optimizer that chooses the best execution plan.

The SQL optimizer analyzes the query, evaluates table statistics, considers available indexes, and generates an optimal execution plan. This is an abstraction layer that few APIs offer: you say "what," the system decides "how."

Granular Access Control

SQL integrates a native access control system. GRANT and REVOKE allow fine-grained control over who can do what on which data. This is the equivalent of an authorization system built into the API.

```
GRANT SELECT (product_name, price) ON catalog.products TO 'api_user'@'%';
```

This user can read the product name and price, but not internal costs or margins. Access control is at the column level.

Standardization

SQL is an ISO standard. Despite variations between implementations (MariaDB, MySQL, PostgreSQL, Oracle), the core of the language is shared. A developer who knows SQL can interact with any relational database.

This is an advantage that few APIs have. REST is not a strict standard (it is an architectural style), GraphQL is a more recent standard, gRPC is tied to Protocol Buffers. SQL has more than 40 years of standardization.

SQL as API: Arguments Against

Expertise Required

SQL is powerful but complex. Writing a simple SELECT is accessible to any developer. Writing a performant query with complex joins, recursive CTEs, and window functions requires significant expertise.

A good REST/GraphQL API abstracts this complexity: the consumer does not need to know how the data is physically organized. With SQL, the consumer must understand the schema, the relationships, and the performance constraints.

SQL Injection

The risk of SQL injection is SQL-as-API's Achilles' heel. If the consumer builds queries through string concatenation, the data is at risk.

```
# DANGEROUS — SQL injection possible
query = f"SELECT * FROM users WHERE name = '{user_input}'"

# SECURE — parameterized query
query = "SELECT * FROM users WHERE name = %s"
cursor.execute(query, (user_input,))
```

REST/GraphQL APIs do not have this fundamental problem: the interface is separated from the data by design.

No Connection Management

SQL does not handle the concept of HTTP sessions, rate limiting, standardized pagination, or API versioning. The MariaDB / MySQL protocol manages connections, but there is no equivalent to HTTP headers, 4xx/5xx status codes, or HTTP caching mechanisms.

These features must be implemented on top of SQL, via proxies (MaxScale, ProxySQL) or application layers.

Schema Coupling

Exposing SQL directly couples the consumer to the physical database schema. If you rename a column, add a table, or modify a relationship, all of the consumer's SQL queries must be updated. A REST or GraphQL API isolates the consumer from these changes via an abstraction layer.

The Verdict: Yes, But...

SQL is functionally a high-level API. It meets the essential criteria: standardized interface, declarative data access, built-in optimization, access control.

But it is an API that requires expertise to use correctly, that exposes security risks if misused, and that does not provide modern management mechanisms (versioning, rate limiting, pagination).

The best approach is probably hybrid:

- **SQL as an internal API** between backend services and the database, with views and stored procedures as an abstraction layer
- **REST/GraphQL as an external API** for consumers who do not need to know the physical schema

SQL is not dead. SQL is not a tool of the past. It is an API — the oldest, the most powerful, and the most underestimated in the software ecosystem.

This article was originally published on Medium.