Syntax Homogenization in Database Query Languages (2020–2025)

Executive Summary

Over the past five years, database query languages have trended toward greater standardization and cross-platform consistency, reducing vendor-specific dominance in query syntax. The SQL standard's latest evolution, **SQL:2023**, introduced features like property graph queries and enhanced JSON support, codifying many functions (e.g. GREATEST, LEAST, LPAD/RPAD, LTRIM/RTRIM, ANY_VALUE) that had long existed as vendor extensions . Major database engines are gradually adopting these features, improving portability: for example, **native JSON data types**—standardized in SQL:2023—are now supported in PostgreSQL, MySQL, Oracle, and others, aligning their behavior more closely . Meanwhile, the new ISO-standard **Graph Query Language (GQL)** (published 2024) has received broad industry support. Leading graph database vendors like Neo4j and TigerGraph have committed to aligning their query languages (e.g. Cypher) with GQL , signaling a convergence of syntax and semantics in the graph data domain. Even emerging open-source graph systems (NebulaGraph, etc.) rapidly implemented GQL .

Standardization efforts in semi-structured data access have also borne fruit: **JSONPath**, an IETF standardized JSON query syntax (RFC 9535, 2024), aims to unify the myriad of previously incompatible JSONPath implementations . Already, new JSONPath libraries in multiple languages (C#, Ruby, Rust) conform to the RFC . In relational databases, vendors that implemented JSON queries based on earlier SQL standards (e.g. the SQL/JSON path language from SQL:2016) are expected to adjust toward the RFC for consistency. This will address historic inconsistencies in JSON filtering behavior across engines. For instance, differences in handling of array slicing or filtering in JSONPath, once common among dozens of implementations, are slated to diminish as engines converge on the standard's definitions .

In the search and analytics arena, **Elasticsearch** and its open-source fork **OpenSearch** have both offered SQL query interfaces to query index data. These began from a common codebase, and initially had very similar syntax. After the 2021 fork, OpenSearch's SQL plugin expanded capabilities (notably adding support for JOIN queries earlier on) , while Elastic's SQL (part of the proprietary stack) lacked joins until mid-2025 when a "lookup join" was introduced in preview . Despite some divergence, the core SQL dialect in OpenSearch remains intentionally compatible with Elasticsearch's, enabling users to run similar queries on both. Both support standard SQL constructs (SELECT-FROM-WHERE, aggregations, etc.), and as of 2025 both now offer at least one form of join operation – indicating a reconvergence of features.

More broadly, **vendor-specific SQL extensions appear to be receding in new development**. Modern applications increasingly rely on widely supported SQL features and ORMs/abstraction layers to remain database-agnostic. As the SQL standard has incorporated formerly proprietary features (window functions, common table expressions, MERGE statements, etc.), the need for custom syntax has declined. Developers can achieve functionality using standard SQL in ~80% of cases across major platforms . Indeed, many new database systems strive for compatibility with established dialects (e.g. PostgreSQL or MySQL) rather than inventing unique syntax, facilitating portability. **Cross-engine query portability has improved yearly**: all major RDBMS now support core ANSI SQL-92 features, and by 2025 even advanced features like recursive CTEs or JSON functions are broadly implemented. The advent of cloud data warehouses and distributed SQL databases (Snowflake, Google BigQuery, CockroachDB, etc.) has not splintered SQL – on the contrary, these systems emphasize standard SQL compliance (with extensions kept optional) as a selling point to minimize user lock-in .

Key Findings

- **SQL:2023 Adoption:** The latest SQL standard (approved June 2023) added property graph queries and standardized many functions long present in vendor dialects. Major databases are at varying stages of implementing these features. Many new functions (e.g. GREATEST/LEAST) were already in popular engines (Oracle, MySQL, Postgres) and are essentially “homogenized” by being standardized . Support for the new **SQL/PGQ (Property Graph Queries)** is nascent: Oracle 23c implements a subset of SQL/PGQ and PostgreSQL developers have a prototype in progress , indicating multi-vendor interest. JSON enhancements in SQL:2023 (native JSON type, JSON_SERIALIZE, IS JSON, etc.) mirrored capabilities that PostgreSQL, Oracle, and MySQL had independently added; now these engines can converge on consistent syntax/semantics for JSON storage and validation . Overall, while full SQL:2023 compliance is rare so soon after publication, its features align with existing practice in many engines, suggesting relatively quick adoption of the most practical elements.

- **Graph Query Language (ISO GQL):** Published in April 2024, GQL is the first ISO database language since SQL. It was developed collaboratively by multiple vendors and academics . There is strong evidence of vendor alignment: Neo4j (with Cypher) and TigerGraph (with GSQL) have publicly committed to support ISO GQL . AWS’s graph service (Neptune) and openCypher project are also on board, with Neo4j and Amazon stating that Cypher and GQL will co-evolve and remain largely identical in core syntax . GQL’s MATCH…RETURN pattern matching syntax is deliberately based on Cypher’s, easing transitions . Already, NebulaGraph announced native GQL support in 2024, positioning itself as the first distributed graph DB to implement the standard . This industry consensus implies that, moving forward, query portability among graph databases will greatly improve. A query expressed in GQL should execute on any GQL-compliant system, reducing the fragmentation that previously existed (Cypher vs Gremlin vs PGQL, etc.). There may still be *temporary* vendor-specific extensions (Neo4j will carry features not in GQL v1 as extensions, e.g. MERGE and LOAD CSV, until future

GQL versions include them), but the trend is clearly toward convergence on one standard language for property graphs.

- **JSONPath Consistency:** Prior to standardization, JSONPath implementations varied widely, causing inconsistent behavior across databases and programming libraries . For example, filtering expressions or edge cases (like handling of null vs missing, negative array indices, etc.) might yield different results in PostgreSQL's SQL/JSON, MySQL's JSON_EXTRACT, or popular libraries like Jayway JsonPath. In February 2024, **RFC 9535 (JSONPath)** was published to define a single, **language-independent JSON query syntax** . The IETF working group explicitly reconciled differences among ~50 implementations, aiming for the "rough consensus" behavior . The result is a well-specified grammar and semantics for JSONPath. While it's early, there are signs of adoption: new compliant libraries in C#, Ruby, and Rust appeared immediately , and even prior to final publication, PostgreSQL had implemented a SQL/JSON path language aligned closely to the draft standard (as part of SQL:2016) . Moving forward, one expects relational databases that support JSON queries to align their implementations with RFC 9535. This means, for instance, a JSONPath filter query ($..?(@.price < 10)) should return the same results whether executed in Postgres, Oracle, MySQL or a standalone library, eliminating the historical inconsistency. **However**, full convergence will take time – dozens of legacy implementations exist, and not all will update immediately . In 2025, minor discrepancies likely persist (e.g. function extension syntax or certain edge-case behaviors), but the existence of the RFC and a growing compliance test suite are guiding engines toward unified JSONPath behavior .

- **Elasticsearch vs. OpenSearch SQL:** OpenSearch, forked from Elasticsearch 7.10 in 2021, inherited Elastic's SQL query interface and has maintained a similar syntax. Both allow using SQL SELECT statements to query search indexes. Initially, their SQL dialects were identical (since OpenSearch's SQL started as the open-source plugin from Elastic). OpenSearch, through the **Open Distro SQL plugin**, extended this dialect by introducing support for operations that Elastic's native SQL did not support. Notably, by 2022 OpenSearch SQL offered **JOIN** capabilities (inner, left outer, cross join) with certain restrictions , whereas Elastic's SQL historically forbade joins due to distributed scalability concerns . OpenSearch's decision to implement joins (limited to two indices and with other constraints) indicates a deliberate move to broaden SQL coverage in line with traditional RDBMS capabilities. In contrast, Elastic focused on alternative approaches (like **Elasticsearch EQL** for sequence queries and **Transforms/Enrich pipelines** for join-like use cases) and only in 2025 introduced a preliminary *lookup join* in Elasticsearch 8.18+ . Thus, for a few years the two dialects diverged in capability. As of 2025, they are somewhat reconverging: both support a form of join (though implemented differently), and both continue to support a similar set of SQL functions, aggregations, and search-specific clauses. The **syntaxes remain largely convergent** – an OpenSearch SQL query will usually run on Elasticsearch SQL without modification, except when using features unique to one side (for example, OpenSearch's multi-index join or Elasticsearch's newer pipeline and ESQL features). The open-source community

and AWS (which leads OpenSearch) strive to keep query syntax familiar to Elastic users, smoothing migration. Going forward, if Elastic broadens SQL capabilities (as hinted by ongoing enhancements in “ES|QL”) and OpenSearch follows suit, users may see a largely unified SQL query experience across these search engines. Minor differences (function names, support for certain data types, etc.) continue to exist but are shrinking relative to the broad common subset.

- **Vendor Extensions and New Codebases:** There is a discernible shift in database development philosophy: new systems tend to embrace standard or popular dialects instead of inventing proprietary syntax. **Vendor-specific extensions are shrinking in prevalence** within modern code for several reasons. First, the SQL standard itself has expanded to cover capabilities that used to require vendor-specific syntax – e.g. recursive queries, window functions, pivoting, MERGE statements, and JSON manipulation are all part of standard SQL now. This reduces the need for developers to resort to vendor-specific solutions and allows databases to provide standard-compliant methods. For instance, 15 years ago one might use Oracle’s proprietary CONNECT BY for recursion or Transact-SQL’s TOP syntax for limiting results; today, standard SQL recursive CTEs (WITH ...) and FETCH FIRST N ROWS/LIMIT are available in Oracle, SQL Server (now supports ANSI FETCH/OFFSET), and others . Second, **portability is a valued feature** in an era of cloud and hybrid deployments – organizations want the freedom to switch databases without total query rewrites. This incentivizes using common SQL. Surveys of best practices encourage developers to “stick to Standard SQL when possible” to maximize portability, and indeed many do so. ORMs and database abstraction layers further hide differences, generating dialect-specific SQL only when necessary. As a result, new application code often minimizes direct use of idiosyncratic constructs.

  Furthermore, new database engines themselves often adopt the surface syntax of popular databases to lower the learning curve. For example, **distributed SQL/NewSQL databases** like CockroachDB or YugabyteDB claim compatibility with PostgreSQL’s dialect and drivers, effectively propagating Postgres’s quasi-standard syntax (which itself closely tracks ANSI SQL with only minor extensions). MySQL forks and derivatives similarly stick to MySQL’s dialect. Cloud data warehouses (Snowflake, BigQuery, Redshift) pride themselves on SQL compliance, differing mostly in optional extensions (e.g. Snowflake’s variant data type or BigQuery’s ML functions) but not in core query syntax. This means the space of commonly used SQL features is actually **contracting to the intersection** supported by all major players. As one discussion succinctly noted, all vendors still have their dialect “flavors”, but “it’s all pretty close to the ANSI standard” in practice – especially for fundamental operations (SELECTs, joins, subqueries, basic functions). Some proprietary extensions are even being phased out in favor of standard equivalents: for instance, Oracle in recent versions encourages standard LISTAGG over its older WM_CONCAT, and Microsoft SQL Server added ANSI-standard TRIM function in addition to its legacy LTRIM/RTRIM. In **new codebases**, developers report that by using a common subset of SQL (sometimes estimated around 80% of full SQL

capabilities) they can remain database-agnostic with minimal sacrifices. This all contributes to a gradual homogenization of SQL usage.

- **Year-over-Year Portability Trends (2020–2025):** Each year from 2020 to 2025 has seen incremental improvements in cross-engine SQL portability. **Feature convergence**: MySQL 8 (2018) and PostgreSQL 11 (2019) both added common table expressions and window functions, eliminating major query portability gaps by 2020. By 2021–22, nearly all active RDBMS could execute queries with CTEs, window functions, and standard aggregates identically. **JSON support** became ubiquitous: if one wrote a query using JSON_EXTRACT/JSON_VALUE in 2020, it might have only worked on MySQL or SQL Server; by 2025, Oracle and Postgres also support JSON querying using SQL standard syntax (JSON_TABLE, JSON path expressions), bringing the behavior closer . **Analytics and BI tools** have also standardized on SQL-92+ syntax, putting pressure on databases to support at least that baseline. Another trend is the use of **PostgreSQL as a de facto standard** for advanced SQL: many new tools (e.g. data science query engines, federated query systems) implement a subset of Postgres SQL for interoperability. Even NoSQL and NewSQL systems often provide SQL interfaces (e.g. Google Spanner uses ANSI SQL, Cassandra has CQL resembling SQL, MongoDB Atlas offers an SQL option, etc.), further reinforcing SQL uniformity. According to industry experts, SQL's longevity and ubiquity forced it to "keep up with the times" by evolving rather than being replaced . This has meant that skills and queries are more transferable year by year – a developer proficient in SQL on one platform in 2020 can more easily adapt to another in 2025 because the core syntax and constructs have more in common now than they did a decade ago. Empirical evidence of improved portability can be seen in multi-database frameworks (like Hibernate, Django ORM, etc.) which find it easier to accommodate multiple backends using standardized SQL features, needing fewer vendor-specific code branches than in the past. In summary, from 2020 to 2025, cross-engine query portability has steadily improved as engines converge on standard SQL features and community demand for open, flexible data solutions grows.

In conclusion, the period 2020–2025 has reinforced the trend that database query syntax is becoming less vendor-dominated and more homogenized. SQL standards and related initiatives (ISO GQL, IETF JSONPath) are both responding to and catalyzing this convergence. While complete uniformity remains unattained – and some dialectal quirks and extensions persist – the common ground shared by different databases' query languages has expanded. This makes it easier to write portable queries and to transfer skills between platforms, benefiting developers and organizations seeking to avoid lock-in. Importantly, this homogenization is driven not by any single vendor's dominance, but by collaborative standardization and broad industry buy-in, marking a shift from vendor-specific query languages toward **truly interoperable querying** across systems.

SQL:2023 Features and Adoption Across Engines

The SQL standard, maintained by ISO/IEC, has periodically incorporated new language features to keep pace with database technology. **SQL:2023**, finalized in June 2023, is a significant update introducing capabilities that reflect modern data needs . Key enhancements include:

- **Property Graph Queries (SQL/PGQ):** SQL:2023 added a standardized syntax for graph pattern matching within SQL queries . This allows treating relational data as a property graph and querying connections via a MATCH clause inside GRAPH_TABLE functions . It's effectively an embedding of graph query semantics (similar to Cypher) into SQL.

- **Enhanced JSON Support:** A native JSON data type is now part of the standard, along with functions like JSON_SERIALIZE, JSON_SCALAR and an IS JSON predicate to validate JSON text . These features build on JSON functionality first introduced in SQL:2016. The standardization of a JSON type acknowledges what many vendors had done ad hoc – e.g. PostgreSQL's jsonb or Oracle's JSON type – and sets a common baseline for JSON handling .

- **New Functions:** SQL:2023 formalized several scalar and aggregate functions that were previously only vendor-specific. Examples are GREATEST and LEAST (to return the max/min of a list of values), string padding functions LPAD/RPAD, string trimming functions LTRIM/RTRIM (distinct from the earlier standard TRIM), and ANY_VALUE (which returns an arbitrary value from a group when strict aggregation isn't needed) . These functions have existed in databases like Oracle, MySQL, and PostgreSQL for years; by adding them to the standard, SQL:2023 essentially codified existing common practice .

The question is: *are these SQL:2023 features broadly implemented across engines?* In late 2023 and 2024, vendors have begun to announce support for parts of SQL:2023:

- **Functions and JSON:** The newly standardized functions (GREATEST, etc.) were already implemented in many engines prior to 2023 as extensions. For example, Oracle, MySQL, and PostgreSQL each have had GREATEST and LEAST for a long time, as well as LPAD/RPAD (Oracle and MySQL) and LTRIM/RTRIM (almost universal). Therefore, compliance in this area is naturally high – vendors can now claim standard support for these by aligning syntax if needed. One outlier was Microsoft SQL Server, which historically lacked GREATEST/LEAST and still uses LEN() or DATENAME etc. for certain things instead of standard functions. It wouldn't be surprising if upcoming SQL Server versions add these functions now that they are standardized, to improve compatibility. As of 2025, most mainstream RDBMS support or have easy equivalents for all the newly standard functions (indeed, **many "new" SQL:2023 functions were implemented long ago under vendor-specific status**). The presence of these in the standard narrows the gap between dialects.

- **JSON Type and Operators:** PostgreSQL was ahead of the curve with a binary JSON type (jsonb) since 2014; Oracle introduced a true JSON type in Oracle Database 21c (2021) and improved it in 23c. MySQL has a JSON type as of 2015. These implementations differ slightly (e.g. Postgres' jsonb is a binary format with indexing support, Oracle's JSON is text storage with functional indexing). With SQL:2023, there is now an official **JSON type** and syntax for inserting/querying JSON. Oracle's engineers note that Oracle 23c's JSON type is aligned with the emerging standard (likely intentionally, as Oracle was involved in SQL:2023) . Postgres might adjust its implementation details to match standard-defined behaviors (for example, ensuring IS JSON checks the same criteria as the standard). Microsoft SQL Server remains a laggard – it supports JSON **functions** but still has no dedicated JSON type (JSON is stored as NVARCHAR). If SQL:2023 spurs them, they might introduce a JSON type in a future release for compliance. In summary, **JSON features of SQL:2023 are either already present in major engines or on their roadmaps**, suggesting broad implementation in the near future. An industry analysis by TigerData in late 2023 described SQL:2023 as "a significant step forward in integrating JSON data" and noted PostgreSQL's leadership in this area .

- **Property Graph Queries (SQL/PGQ):** This is the area with least immediate adoption, as it's entirely new to relational engines. However, there are concrete moves: **Oracle Database 23c** (released as "23c Beta" in 2023, and an update nicknamed 23**_ai_** in 2024) introduced SQL/PGQ features. Oracle's implementation uses a CREATE PROPERTY GRAPH DDL and a GRAPH_TABLE() function to run graph pattern queries, exactly following the SQL:2023 syntax . Oracle's 23c docs describe that its SQL PGQ is currently a subset of Oracle's pre-existing PGQL language, with plans to expand it continually . This indicates Oracle's commitment to the standard (unsurprising, since Oracle was a contributor on the ISO committee). **PostgreSQL** also has shown interest: developers like Peter Eisentraut shared a prototype of SQL/PGQ at a 2024 conference . A patch series for PostgreSQL 17 (expected 2024) is under discussion to add property graph query support . This is non-trivial for an open-source RDBMS, but the existence of a patch suggests at least partial support may land in the next couple of releases, especially since it keeps PostgreSQL competitive and up-to-date with standards. Other vendors are quieter publicly: IBM DB2 and Microsoft have not announced anything specific about SQL/PGQ, though IBM has a graph store/query feature in DB2 (Graph Store) that might eventually align with the standard. Smaller database systems like **Memgraph** or other hybrid graph-relational platforms could adopt SQL/PGQ as a way to differentiate by standard compliance. In general, graph querying in SQL:2023 is optional (not a core requirement of the standard), so adoption will depend on each vendor's strategy. But given Oracle and Postgres leading the way, we can expect **broad interest in implementing SQL/PGQ** in the next few years, to cater to users' graph analytics needs without forcing a separate graph database.

- **Other Improvements:** SQL:2023 also included various minor syntax simplifications and extensions (e.g. allowing <column> IS DISTINCT FROM <value> without verbose

syntax, or other ease-of-use tweaks). These tend to be low-hanging fruit that engines can implement easily. We have already seen many databases incorporate such things *ahead* of formal standardization because they arose from real-world use. For example, IS DISTINCT FROM (null-safe equality) has been in PostgreSQL for years and in the SQL standard since SQL:2003; an analogous construct NULL-safe equals (<=>) exists in MySQL. If any new shorthand or syntax ease from SQL:2023 wasn't already in a product, vendors will add them to not fall behind in ergonomics.

Overall, **SQL:2023's features are on track to be broadly implemented** across major engines, albeit on different timelines. The fact that most new features were inspired by existing vendor functionality means databases are not starting from scratch. Adoption is uneven in 2024 – e.g., property graph queries are only in a couple of systems so far – but the momentum is clearly toward embracing these standard features. Vendors often use standards compliance as a selling point; for instance, Exasol (an analytical RDBMS) publishes documentation of its support for each SQL standard feature, and we can expect updates listing which SQL:2023 optional features it supports . Likewise, open-source projects like MariaDB, which historically lagged on some standards (like window functions), might use SQL:2023 as an opportunity to catch up or advertise compliance. An Oracle blog announcing SQL:2023's publication noted that having these features in the standard "gives a boost" to adoption of the technologies – vendors can now confidently implement them knowing they are officially sanctioned rather than proprietary gimmicks.

In summary, **SQL:2023's major additions (graph queries and JSON enhancements) are aligned with industry trends and are seeing early implementation in multiple engines, while its formalization of common functions immediately raised the baseline of "standard SQL" to include what many databases already do**. This contributes to homogenizing SQL syntax across vendors, as proprietary usages are replaced with standardized syntax.

Alignment of Graph Vendors on ISO GQL

The graph database domain historically lacked a standard query language, leading to fragmentation: Neo4j's Cypher, Oracle's PGQL, Apache TinkerPop's Gremlin, and various others all competed. In late 2010s, an international working group began crafting **ISO GQL (Graph Query Language)** to unify the industry. In April 2024, ISO/IEC published GQL as a new standard (ISO/IEC 39075) . This is a landmark, making GQL the first ISO database query language since SQL itself . The critical question is whether graph database vendors are actually aligning on this standard syntax and semantics.

Industry Collaboration: Notably, the GQL standard was developed with direct participation from major vendors. The editorial lead was a Neo4j product manager (Stefan Plantikow) . Neo4j – the creator of Cypher – thus had a heavy influence on GQL. Other contributors included Oracle, TigerGraph, AWS, and members of the Linked Data Benchmark Council (LDBC) . This collaborative origin already set the stage for adoption, as those vendors were essentially

designing the language they intend to use. According to IDC's analysis, the industry had "waited quite some time" for a standard like GQL and its formalization should remove a barrier to broader graph technology adoption .

Neo4j and Cypher: As the dominant graph database vendor, Neo4j's stance is pivotal. Neo4j has publicly embraced GQL. Jim Webber, Neo4j's Chief Scientist, explained that GQL will feel familiar to both SQL users (in parts of its aggregate and result set handling) and Cypher users (in its pattern matching). He indicated that if you know Cypher, GQL is a straightforward next step. In an AWS blog co-authored by Neo4j and Amazon engineers, they went further: *"Cypher is the best and fastest path to GQL"* . They assure users that **Cypher and GQL have been on a deliberate convergence course**, and that Neo4j's Cypher will evolve to fully align with GQL over time . For example, GQL uses certain keywords differently (INSERT instead of Cypher's CREATE, a new FOR instead of UNWIND) . Neo4j committed that it will support both forms (so existing Cypher queries don't break) and gradually encourage the GQL way . Essentially, Neo4j is treating GQL as the future of Cypher. They even mentioned that GQL's first version is about as powerful as SQL-92 (meaning it covers core capabilities but not every advanced feature yet), and that Cypher's extra features not in GQL v1 (like the MERGE clause, FOREACH updates, or bulk CSV loading) will remain as temporary vendor extensions until they can be proposed and added to GQL in subsequent revisions . This approach mirrors how SQL was adopted: initial standards with vendor extensions gradually phased out as the standard matured. The takeaway is **Neo4j is fully aligning with GQL**, ensuring its massive user base will indirectly become GQL users.

TigerGraph and Others: TigerGraph, a competitor to Neo4j, also supports GQL. TigerGraph's SVP of Engineering wrote that the graph industry has seen "a plethora of vendors" each with their own language, and **GQL emerged to provide a standard** . TigerGraph sees GQL as foundational, akin to what SQL did for RDBMS . This is a strong endorsement. Their quote underscores that standardization will foster a richer ecosystem and "prosperity of graph databases" . Practically, TigerGraph's native language GSQL is somewhat different from Cypher, but TigerGraph has worked on an **openCypher compatibility** mode as well. With GQL, TigerGraph will likely implement a GQL parser/executor or adapt GSQL to meet GQL semantics. The Register reported that *"in a sign of harmony among software vendors, TigerGraph, Neo4j's main rival, is also supporting the standard."* This implies that despite competing in the market, both companies realize a common language benefits everyone by expanding the graph market.

Amazon Neptune: Amazon's graph database (Neptune) historically supports openCypher and Gremlin as query languages. Amazon was an active participant in GQL's creation . In the April 2024 AWS Database blog, AWS's database team co-wrote with Neo4j leadership to announce the GQL standard . They explicitly reassure customers that their Cypher queries and skills are safe and that Neptune will support GQL. The blog strongly hints that Neptune will adopt GQL (likely via openCypher's transition, since Neptune uses a fork of openCypher for its property graph queries). Amazon's support is significant because Neptune's backing of GQL means one of the largest cloud vendors will push the standard, potentially influencing other graph services.

Other Graph Vendors: Several other players are worth noting:

- **Oracle:** Oracle contributed PGQL (its Property Graph Query Language) to the standardization effort. PGQL and GQL share similarities (both influenced by Cypher). Oracle's converging too: Oracle 23c's graph feature can use PGQL, but one can expect Oracle will support GQL when finalized. Oracle even distinguished "SQL/PGQ" from "PGQL", indicating SQL side support vs an existing language. In Oracle's public statements, they view GQL as complementary to SQL/PGQ: PGQL was Oracle's prior solution for graph, but GQL is the new ISO separate language, and Oracle will likely implement a GQL query interface to their graph data store.

- **Microsoft**: Azure Cosmos DB's Gremlin API and other niche graph DBs (like IBM's JanusGraph-based offerings) have not made clear announcements. JanusGraph itself uses Gremlin; if Gremlin and GQL will coexist or if JanusGraph would adopt GQL is uncertain. But given that the standard is out, any graph DB project would face pressure to at least allow GQL queries to avoid being "proprietary." We might see multi-language support (as Neptune does with both Gremlin and openCypher/GQL).

- **Open Source Graph DBs:** Aside from Neo4j (which has an open core) and JanusGraph, newer projects like NebulaGraph have seized the opportunity to claim first-mover advantage. As cited earlier, NebulaGraph announced **"native GQL support"** in April 2024, positioning itself as fully compliant with the ISO standard . They emphasize that their engine was redesigned to support GQL at an architectural level, not just as a translation layer . This suggests NebulaGraph is betting on GQL as a differentiator and hoping to attract users by being an early adopter. Such competition likely motivates others (e.g. Amazon, Neo4j) to keep pace in implementation quality.

GQL Syntax and Semantics Alignment: The core of GQL is heavily based on **Cypher** and its subset **openCypher** . This means many vendors that already implemented Cypher (Neo4j, Memgraph, RedisGraph, etc.) are quite aligned by default. Differences do exist between Cypher implementations, but the GQL working group aimed to reconcile those. The GQL standard includes not just pattern matching ((a)-[r]->(b) notation) but also more SQL-like constructs for set operations, subqueries, etc., to make it a full-featured language. According to experts, GQL's feature set in v1 is comparable to SQL-92 – it has the fundamentals of graph querying, but some advanced Cypher constructs were deferred. To handle that, GQL allows **vendor extensions** (similar to SQL) . The expectation is these extensions will diminish over time as GQL evolves. This mirrors how vendors align on SQL: new standard editions incorporate formerly vendor-specific features.

In practice, **we already see cross-vendor queries becoming possible**. For instance, a simple GQL query pattern like:

```
MATCH (p:Person)-[c:Connection]->(q:Person)
WHERE p.age > 30
```

RETURN p.name, q.name;

should be executable on Neo4j (post-GQL support), TigerGraph (if it supports GQL directly or via Cypher compatibility), and NebulaGraph 5.0, with identical results. Under old circumstances, one would have had to translate that to Cypher for Neo4j (MATCH (p:Person)-[:Connection]->(q:Person) ...) vs TigerGraph's GSQL (which uses a different SELECT-like syntax). With everyone aligning to GQL, the **portability of graph queries greatly increases**.

It's important to note the time horizon: GQL was published in 2024, so full adoption is in early stages in 2025. But the commitment is clearly there from the leading players, which strongly indicates that in coming years graph vendors will indeed coalesce around ISO GQL. Graph query *semantics* (like how certain pattern matching edge cases are handled, or how nulls/absences are treated in graph context) historically differed, but the standard defines these now. IDC's quote in The Register sums it up: *"The formalization of GQL should remove a key barrier to graph database adoption"* by easing portability . Already, CIOs and developers can be confident that investing in writing GQL queries won't tie them to one product . We've seen a "harmony" among erstwhile competitors unprecedented in earlier times – a testament to the industry recognizing that expanding the graph pie is beneficial to all, and standardization is the way to do it.

In conclusion, **graph database vendors are aligning strongly on ISO GQL**. The heavy hitters (Neo4j, TigerGraph, AWS) are not only supportive but actively driving it, and smaller players are racing to implement it. While vendor extensions and parallel legacy languages (like Gremlin) won't vanish overnight (there's no indication Gremlin will be abandoned immediately, for example), the trajectory is set toward GQL becoming the **lingua franca** of property graph querying. This alignment dramatically improves the outlook for query portability and skill reuse in the graph database arena, which historically lagged the relational world in standardization.

JSONPath (RFC 9535) Consistency Across Engines

JSONPath, informally described as "XPath for JSON", provides a way to navigate and filter JSON structures. Prior to 2024, there was no single authoritative specification. Stefan Gössner's original 2007 proposal seeded many implementations , and over 50 incompatible variants sprouted across languages . This led to a situation where the *same* JSONPath query string could produce different results depending on the database or library used – undermining the idea of JSONPath as a portable query mechanism.

Examples of Past Inconsistency: A concrete example is instructive. Consider the JSONPath expression $.store.book[0] intended to retrieve the first book in a JSON document. Something straightforward like this was uniformly handled. But more complex expressions, like filters, diverged. For instance, JSONPath filtering syntax (one popular form: ?(@.price < 10) to filter array elements with price < 10) was not implemented by all libraries, or the syntax to combine conditions varied (some used && vs and, etc.). Another sticking point: array slicing (e.g.

$..book[0:2]). The Gössner original didn't include Python-style slicing, but some implementations added it (like Jayway's did with [0:2]), while others didn't, or used different notation. Another difference: how to interpret the ".." recursive descent operator and whether it returns containers or leaf values differed between libraries. Even fundamentals like whether JSONPath queries return a single value or a list could vary (some return the first match vs all matches).

In databases:

- **PostgreSQL** introduced JSONPath support in version 12 (2019) as part of its adherence to the SQL/JSON standard. This means Postgres's implementation was based on the **SQL:2016 JSON path language**, which is similar but not identical to Gössner's JSONPath. For example, Postgres uses a syntax like $.books[*] ? (@.price < 10) for filtering (as per SQL standard) and supports filtering, arithmetic, wildcard, etc. Postgres aimed to meet the SQL standard conformance .

- **MySQL** from 5.7 (2015) supported JSON functions. MySQL uses JSONPath in functions like JSON_EXTRACT, but its syntax had some limitations. MySQL's JSONPath, for instance, did not originally support wildcards at arbitrary depths (** or ..), and required quotes around keys with special chars, etc. It was essentially a subset tailored to what MySQL needed for its functions.

- **SQL Server** (from 2016) has JSON querying via JSON_VALUE/JSON_QUERY, but its "path" syntax is a simple dot notation (like $[0].name for array index) without the more advanced operations; it doesn't support filters or deep scans at all. So one could say SQL Server's JSONPath is extremely limited and non-standard – it's more JSON Pointer-like.

- **Oracle** implemented JSON filtering in Oracle 12c (2014) with a syntax using JSON_EXISTS and passing a path string; Oracle's path syntax had its own quirks (using ? for condition like $.book[?(@.price < 10)] if I recall correctly, but with slight differences from others).

- **MongoDB** (though not SQL, worth noting) historically had no JSONPath, instead using its own query object syntax, so not relevant here.

The inconsistency across engines was recognized as a problem. If a user wrote a JSONPath expression in an application and then tried to use it in Postgres vs MySQL vs a Java library, they could get different behaviors or errors. A GitHub issue in 2020 noted *"there is not a single DB implementation that supports [JSONPath] (at least we couldn't find any)"* – meaning fully supports the concept as known. It pointed out that usage is primarily in client code.

RFC 9535 (2024): This IETF effort (co-edited by Glyn Normington and others) took a comprehensive look at existing behaviors. The group's charter explicitly was to base the

standard on “common semantics of existing implementations” and where they differ, choose the best approach with minimal disruption . Essentially, they gathered a **comparison of JSONPath implementations** (the JSONPath Comparison Project by Christoph Burgmer was instrumental) and identified consensus behavior vs outliers. The RFC is careful to define things like:

- The output must be a list of all matching nodes (JSONPath always returns zero, one, or multiple nodes as an array – removing ambiguity where some libraries returned first match or scalar).

- Specific notation for filters, scripted expressions, escape characters, etc. are standardized (e.g. @ refers to current node, && and || for boolean, etc.).

- How to handle JSON arrays and objects in edge cases, like the order of results (which can be considered undefined if the document itself is an object with unordered keys, but the RFC addresses nondeterministic ordering issues).

- What happens if no match – some implementations returned null vs empty list; the standard opts for an empty sequence as the result (consistent with Xpath-like semantics, I believe).

- Handling of special cases like JSON numbers (some libraries supported numeric comparisons with implicit conversions, others didn’t).

By and large, RFC 9535 has *settled* the JSONPath syntax and expected behavior.

Consistency Improvements: Now, to the question: Is JSONPath behavior consistent across engines? **As of 2025, we are in a transition.** Immediately after RFC publication, some engines or tools started adopting it:

- There’s mention that *“the standard already has the following implementations: JsonPath.Net in C#, jpt in Ruby, serde_json_path in Rust”* which conform to RFC 9535 . These are likely libraries updated or created to match the spec. Their existence means if you use those libraries (e.g. in an application), you get consistent behavior with the spec.

- For databases: PostgreSQL’s JSONPATH was pretty close to what ended up in RFC (since both drew from the SQL standard and cross-implementation feedback). It might require minor tweaks to fully align (the RFC might have chosen slightly different escape rules or function names). PostgreSQL devs have been tracking the IETF work – indeed one message from a Postgres hacker mused about the new RFC’s function extension mechanism , indicating awareness.

- MySQL and Oracle will likely implement more complete JSONPath now that a standard exists. Oracle 23c might include support for the RFC style JSONPath in its JSON operators (especially since Oracle’s SQL:2021 (the draft preceding 2023) was aligning

with an earlier draft of JSONPath).

- There is not yet a lot of public info of “MySQL 8.x now compliant with RFC 9535” or similar – but MySQL 8 did improve JSON features gradually. It may take a version or two for MySQL to fully support the more complex constructs allowed by the RFC (like arithmetic in JSONPath filters, etc., which MySQL’s JSON functions didn’t support initially).

One sign of convergence: independent of RFC 9535, the SQL:2016 standard had already unified JSON querying for SQL engines – so Oracle, Postgres, DB2 and MySQL have all tried to implement the **same** SQL/JSON Path language from SQL:2016, which is very similar to JSONPath. That standardization at least made those four closer to each other (e.g. all of them accept $.name syntax and ?(@.field > 10) filters in some form). The IETF RFC is in large part aligning with that SQL standard’s semantics too (since there was input from people in both processes). Thus, going forward, we expect **nearly uniform JSONPath support**. A query like $.store.book[*].author or even a filtered query $.store.book[?(@.price < 10)].title should yield the same authors or titles list whether you run it in Postgres using its jsonpath operators, in MySQL via JSON_SEARCH/JSON_EXTRACT, or in a standalone library.

One caveat: not every engine will implement *all* aspects of JSONPath (some may choose to support a subset if performance or complexity is a concern). But because JSONPath is now clearly defined, any deviations will be documented as such, rather than accidental. For example, if an engine doesn’t support the [^] union operator (just hypothetical), that would be a conscious omission. The RFC also allows an **extension mechanism** (for custom filter functions), which means engines could extend JSONPath in proprietary ways but still claim compliance for the base spec . That’s analogous to SQL: vendors support the core standard but might have extra functions.

As of 2025, **some inconsistencies still linger simply because not everything has updated**. There are indeed dozens of older libraries that may never update (but those aren’t “engines” in the sense of database engines, more like language-specific libs). Among databases, the big ones will likely converge:

- **PostgreSQL** 15/16 returns results for JSONPATH queries that match consensus and will adjust to any minor spec differences.

- **Oracle** in 23c has a JSON implementation that they claim conforms to IETF JSON (RFC 8259) and presumably now JSONPath standard (Oracle’s documentation for JSON in 23c references “RFC 8259 compliance” and likely will mention RFC 9535 in future).

- **SQL Server** still doesn’t implement the full JSONPath filtering (its JSON_VALUE only does simple path). It’s unclear if Microsoft will extend that. If they do, they now have a clear template (the RFC). If not, SQL Server remains an outlier by not supporting

complex JSONPath at all.

- **MySQL** has room to grow in JSONPath expressiveness. It might gradually incorporate more standard syntax in its functions (ensuring, say, that it supports all syntax in the RFC).

- **NoSQL engines**: Some, like Couchbase, have their own JSON querying (N1QL) which is SQL-like, and others like ArangoDB have JSONPath-like queries. They too might move to standard JSONPath for API consistency.

In conclusion, the publication of RFC 9535 is bringing **consistency to JSONPath across engines**. The full benefits will be realized over the next couple of years as database vendors and library maintainers implement or adjust to the standard. The consensus among the JSONPath working group was that while convergence might be slow (with 50+ impls, “convergence is likely to be a slow process”), the standard provides a permanent reference that everyone can target . Already, having a test suite (Compliance Test Suite) and reference implementations makes it easier for engine developers to validate their JSONPath behavior .

Thus, **by 2025, we see the early phase of JSONPath behavior becoming consistent** across databases: new engines or versions align with the RFC’s defined semantics, eliminating many historical quirks. Some engines that haven’t updated yet might still exhibit old behavior (so a cautious developer would still test JSONPath queries on each target engine). But the trajectory is clear – JSONPath is going from a nebulous concept to a true standard query syntax that can be relied on for consistent results across different environments. This standardization mirrors the earlier path of SQL: initially vendor-specific, eventually standardized – now JSONPath is on that road to homogenization, which ultimately improves multi-engine compatibility in handling semi-structured JSON data.

Convergence of OpenSearch and Elasticsearch SQL Syntax

Elasticsearch (a Lucene-based search and analytics engine) introduced an SQL query interface in version 6.3 (2018) , enabling users to query data in Elasticsearch using familiar SQL syntax. This was developed as part of Elastic’s X-Pack features to broaden adoption by SQL-fluent analysts. Meanwhile, Amazon, prior to the fork, had open-sourced a plugin for SQL (called **Open Distro SQL**). In 2021, Elastic’s licensing changes led Amazon to fork Elasticsearch 7.10 into **OpenSearch**. OpenSearch incorporated the SQL plugin as a core feature. We thus have two parallel SQL implementations: Elastic’s (closed-source, evolving in 7.x and 8.x) and OpenSearch’s (open-source, based on the last open Elastic code plus new enhancements by the community).

The key question: *Do OpenSearch and Elasticsearch SQL syntaxes converge?* In other words, are they staying compatible or diverging?

Initial State – Common Ancestry: At the moment of the fork (early 2021), OpenSearch 1.0's SQL was effectively identical to Elasticsearch 7.10's SQL. Both supported a subset of ANSI SQL: SELECT-FROM-WHERE queries, aggregations (with GROUP BY), ORDER BY, LIMIT, and certain SQL functions. They both translate SQL into Elasticsearch's native Query DSL under the hood. For basic queries, the syntax was exactly the same. For example, a query like:

```
SELECT host, COUNT(*)
FROM logs
WHERE status = 500
GROUP BY host
ORDER BY COUNT(*) DESC
LIMIT 10;
```

would work identically on Elasticsearch SQL and OpenSearch SQL, yielding the top 10 hosts by number of 500 errors, given an index logs. Both use SELECT … FROM index convention (with index names in place of tables), both allow WHERE conditions using lucene query syntax for full-text if needed (like MATCH(text)). So initially, convergence was 100%.

Divergence in Features (2021–2023): Post-fork, Elastic and OpenSearch had different development trajectories:

- **Joins:** By design, *Elasticsearch SQL did not support JOINs* originally, as joins are non-trivial in a distributed index context . Elastic encouraged denormalization or use of parent-child join features outside SQL. **OpenSearch SQL, however, implemented a form of JOIN.** The Open Distro SQL team added support for one-hop join queries – specifically inner join, left outer join, and cross join between two indices . This was an "enhanced feature" highlighted in Open Distro documentation . Of course, they imposed constraints: only two indices can be joined, each index must be aliased, no multi-level nested joins, etc. . For example, you could do:

```
SELECT s.name, sc.school_name
FROM students s
JOIN schools sc ON s.school_id = sc.id;
```

- in OpenSearch (assuming students and schools are two indices). Elasticsearch's SQL would reject such a query historically, as confirmed by many sources and user discussions . Elastic had other ways (like use of a "join" field type or pipeline transforms) but not via SQL. Thus by 2022, **OpenSearch SQL had diverged by offering join syntax that Elastic SQL lacked**.

- **Functions and Data Types:** Both engines supported a similar set of scalar functions (string functions like UPPER(), numeric functions like ABS(), date functions like YEAR(), etc.) since those were part of the original plugin. Over time, OpenSearch added a few that Elastic might not have (for instance, OpenSearch supports some array and JSON

functions as per AWS docs , given OpenSearch can query JSON fields via SQL). Elastic's SQL until recently did not support nested/nested object querying or JSON field extraction in the SELECT list, etc. However, Elastic has been developing its stack – one notable development was **EQL (Event Query Language)** for sequence queries, and a new **ES|QL** in 2023–2025 that seems to blend pipelining. There's a hint that Elastic is moving beyond pure SQL into a more search-oriented superset (for example, allowing WHERE text LIKE 'foo%' to use full-text search, etc.). If Elastic changes its SQL (or evolves it into "ES|QL"), that could cause divergence. But at least until 2023, Elastic maintained standard SQL-ish syntax for what it did support.

- **Piped Processing vs SQL:** OpenSearch introduced **PPL (Piped Processing Language)** as an alternative to SQL, similar to Splunk's syntax. Elastic, on the other hand, introduced **KQL (Kibana Query Language)** and the **Graph UI**, but not an official separate pipeline language for search. These are separate query modalities, though, and don't affect the SQL syntax directly. So one could say this is a difference in ecosystem, not in the SQL dialect itself.

Re-convergence / Compatibility: There's evidence that despite OpenSearch forging ahead on features like JOIN, Elastic eventually circled back. In 2025, Elastic announced native join support (specifically a "lookup join") in technical preview . It is restricted (the joined-to index must be an "lookup" index with a single shard, functioning like a broadcast join scenario) . But crucially, Elastic's syntax for this join is basically SQL join syntax (they mention possibly using JOIN USING or equality with qualifiers in future) . This means Elastic's SQL dialect is gaining capability closer to OpenSearch's. By Elastic 8.18, a user can perform a join in SQL (with some configuration). So ironically, **Elastic is now converging toward the extended SQL functionality that OpenSearch had earlier**.

On the whole, from a user perspective in 2025:

- The **core SQL syntax** (SELECT, WHERE, GROUP BY, HAVING, ORDER, LIMIT) is the same in OpenSearch and Elastic. Both support a wide range of expressions and functions similarly. Documentation from both sides echo each other in these basics.

- The **differences** are in edge capabilities. For a long time, "JOIN" was the glaring difference – OpenSearch allowed it (with limitations) , Elastic did not (and one had to do workarounds) . Now Elastic allows a form of join in preview . The join implementations differ under the hood, but to the user writing SQL, they likely will look similar (maybe Elastic only supports a special syntax like SELECT ... FROM A JOIN B /*+ LOOKUP*/ ON ... – but likely they aim to make it standard SQL syntax). So that gap is closing.

- Another possible difference: OpenSearch might support more SQL data types mapping (for instance, it may allow DATE type and CAST to date in queries; Elastic SQL might also, but I recall early versions had some limitations in date handling, requiring CAST).

- **SQL coverage:** The AWS/OpenSearch SQL documentation boasts support for “nested queries, subqueries, set operations” etc. . Elastic’s SQL in recent releases also supports subqueries and some unions. If anything, OpenSearch might push the envelope a bit more because they have had active development and user contributions. But Elastic invests heavily in performance and integration (like making SQL work with security features, new data types, etc.). So it’s not that one is strictly superset of the other; they are mostly overlapping circles with a few unique bits each, which over time might converge further.

Portability between the two: Users who migrated from Elasticsearch to OpenSearch after the fork expected that their existing SQL queries would run the same – and indeed they do. Conversely, if someone wrote queries with the new OpenSearch-only features (like join), those wouldn’t run on Elastic until recently. If now Elastic supports similar join, then even those may become portable. There are small syntax specifics: for example, OpenSearch SQL uses backtick quoting for certain identifiers and has certain reserved words – but Elastic SQL likely the same (since derived from same code). OpenSearch might have introduced synonyms for certain functions (the AWS doc mentions function synonyms added in SQL for familiarity). For instance, OpenSearch SQL might allow both ISNULL(x) and the standard x IS NULL. These are minor.

It’s worth noting that both engines aim to be “compatible with many SQL client tools”. They both offer a JDBC driver and ODBC driver that speak their SQL dialect to the engine. I’ve seen references that OpenSearch’s JDBC driver was forked from the Elastic JDBC driver. They presumably kept the same protocols and query language so that tools like Tableau or PowerBI can connect to either and run the same queries. So there is an intentional effort to **not** diverge drastically in SQL syntax – because that would break compatibility with third-party tools and user expectations.

Long-term: Elastic seems to be evolving the concept of SQL into something more search-native (the blog posts mention “ES|QL enhancements” as if merging EQL and SQL capabilities). OpenSearch, being community-driven, will probably adopt improvements that are useful to users, whether they originate in Elastic’s ideas or from user requests. If Elastic’s ES|QL ends up introducing non-standard syntax, OpenSearch might or might not copy it. For example, Elastic might add a syntax to do full-text search scoring in SQL (like WHERE MATCH(text_field, 'foo bar') > 0.8). If they do, OpenSearch might implement a similar function or clause for parity. Historically, Open Distro did incorporate features from Elastic’s future (they had alerting, index state management, etc. that Elastic later built differently). For SQL specifically, the common base and user desire for cross-compatibility means they have incentive to remain as convergent as possible.

In sum, **OpenSearch and Elasticsearch SQL syntaxes have remained largely convergent**, with the exception of certain advanced features that one introduced ahead of the other. By 2025, even those differences are diminishing: OpenSearch’s head start on JOINs is matched by Elastic’s new join support , and both continue to expand SQL functions (window functions, if any

– interestingly AWS docs show OpenSearch supports ROW_NUMBER() OVER(...) ; Elastic's SQL added some analytic functions as well in 7.x). Both are dialects of SQL aimed at a search engine context, so they do share inherent limitations (e.g., neither supports full arbitrary multi-table join or multi-statement transactions – those aren't relevant in a search engine). They both also lack certain SQL-92 features (no integrity constraints, of course, and some data types absent). But those are inherent to the domain rather than dialect divergence.

One could argue that OpenSearch's SQL might gradually drift if the community adds unique features that Elastic doesn't have. However, so far Amazon's strategy with OpenSearch seems to be maintaining as much compatibility as possible with Elastic v7 APIs (to allow easy migration). The SQL plugin is part of that. So they wouldn't intentionally break or change SQL syntax in incompatible ways; they would only extend it in hopefully non-conflicting ways. And now Elastic's catching up on at least the major extension (JOIN). The existence of third-party layers like **tSQL** or **tQL** (in the search results) which claim compatibility with "Elasticsearch SQL and OpenSearch SQL" further underscores that the two are considered effectively the same dialect by users, with only minor version differences.

To conclude, **OpenSearch and Elasticsearch SQL syntaxes are largely aligned and convergent.** Queries written for one will usually run on the other. The introduction of SQL JOIN in OpenSearch created a temporary divergence, but Elastic's move to implement similar functionality has re-converged their capabilities to an extent . Both systems appear committed to offering a familiar, nearly ANSI-standard SQL interface for search analytics, and this common goal keeps their dialects from straying far apart.

Decline of Vendor-Specific Extensions in New Databases

Historically, each database vendor introduced proprietary extensions to SQL, either to provide features not in the standard or to differentiate their product. This led to the notorious "SQL dialects" – for example, Oracle's PL/SQL and specific functions, Microsoft's T-SQL, or even the divergent procedural languages and syntaxes in each system. The question here is whether **vendor extensions are shrinking in new codebases** – in other words, are newer database products or modern development practices moving away from vendor lock-in at the query syntax level?

Trend in New Database Systems: Many modern databases launched in the last decade deliberately piggyback on an existing well-known syntax instead of inventing anew:

- **NewSQL / Distributed SQL** (CockroachDB, YugabyteDB, Google Spanner): these systems chose to be PostgreSQL-compatible in syntax and drivers. CockroachDB, for example, advertises that it speaks the Postgres query language and supports Postgres drivers. This is interesting because it means Cockroach didn't create "CockroachSQL"; they realized that adopting a popular dialect (Postgres) lowers adoption friction. Similarly, Amazon's Aurora offers modes that are wire-compatible with MySQL or Postgres. Azure's Cosmos DB, when introducing an SQL API, made it somewhat akin to SQL

(though for JSON data). The overarching pattern is *reuse, not reinvent*. By aligning with either an ANSI-standard or a de facto standard (like Postgres's dialect), these new systems inherently minimize vendor-unique syntax. They might extend in implementation (e.g. Spanner doesn't allow all types of joins or has different performance considerations, but the syntax is standard).

- **Cloud Data Warehouses**: Snowflake's query syntax is very close to Oracle/ANSI SQL, with minimal differences. Snowflake did add some extension (like FLATTEN table function for semi-structured data, and allowing SELECT * REPLACE semantics), but by and large if you know standard SQL you can use Snowflake. Google BigQuery started with a somewhat quirky SQL (it had a legacy SQL that was not fully ANSI), but by 2016 they introduced **Standard SQL** mode which conforms to ANSI SQL-2011. BigQuery's standard SQL intentionally removed many differences, and today BigQuery's dialect is highly standard (with extensions mainly for JSON, geospatial, and ML functions that are clearly namespaced or optional). Amazon Redshift is basically PostgreSQL 8.0-derived; it has a few extensions for distribution keys, etc., but query-wise, it's mostly ANSI SQL with Postgres quirks. Azure Synapse (formerly SQL Data Warehouse) is actually based on SQL Server, so it's T-SQL, which is a bit vendor-specific in procedural aspects but the SELECTs are ANSI-compliant.

So, newer DBs are not introducing radical new syntaxes; instead they're trying to **embrace the standard or established open dialects**. This leads to fewer brand-new vendor-specific languages. Compare this to, say, the 1980s/90s when each vendor had a completely proprietary extension language (like Informix-4GL, Oracle PL/SQL, etc.).

Standards Catching Up: The SQL standard committees have actively worked to encompass features so that vendor innovations become standard features, reducing the need for divergence:

- **Example – MERGE:** Different vendors had different upsert syntax (Oracle's MERGE, MySQL's ON DUPLICATE KEY, etc.). The standard SQL:2003 defined MERGE and now most vendors support that exact syntax or a close variant. Over time, developers might prefer using MERGE if it's widely supported, instead of vendor-specific upsert hacks.

- **Example – Analytics:** Window functions were once only in Oracle and later in SQL Server (2005) and DB2. Now even MySQL and MariaDB (as of 2018) implemented the standard window function syntax. There's no need for a proprietary alternative when the standard one is everywhere. So code written with ROW_NUMBER() OVER (PARTITION BY ...) will run on nearly any modern DB (Oracle, DB2, SQL Server, Postgres, MySQL 8, etc.) – ten years ago that statement wouldn't hold (MySQL lacked it).

- **Example – Recursive queries:** Instead of Oracle's old CONNECT BY or SQL Server's proprietary CTE with UNION ALL (which they did align to standard mostly), now the standard WITH RECURSIVE is implemented by Oracle (in 11g they added it), by DB2,

by Postgres, etc. So new code uses the standard WITH rather than Oracle-specific syntax because it's supported almost everywhere.

Therefore, one could say **the gap that vendor extensions fill is smaller**. Vendors continue to innovate, but often through contributions to standards or at least by adopting common frameworks. For example, many vendors had their own JSON syntax around 2015; but once SQL:2016 JSON came, they pivoted to that, or at least to a convergent approach. Microsoft is a bit of an exception with JSON (they still don't have a JSON data type), but otherwise they participate too (they implemented many ANSI things in recent versions – e.g., STRING_AGG function with a clause is very similar to the standard).

ORMs and Portable Code: In application development, the use of ORMs (Object-Relational Mappers) and database abstraction layers has increased. These tools encourage writing database-agnostic code: you write in a host language and the library generates SQL tailored to each database backend. The ORM will only use features that it can emulate on all supported backends, or it will have different code paths. For app developers, this reduces their incentive to use vendor-specific SQL directly, since that might break the ORM's abstraction. Additionally, the prevalence of microservices and polyglot persistence sometimes means teams want the flexibility to switch databases if needed; writing standard SQL or using abstraction helps achieve that. A Reddit commentary noted that "it is possible to limit engine proprietary code to a minimum" using layers like Liquibase for schema and SQLAlchemy for queries . That mindset is more common now than decades ago when stored procedures and DB-specific optimizations were heavily used.

Open Source Influence: With the rise of open-source databases (MySQL, Postgres, MariaDB) and their broad adoption, proprietary databases can no longer rely on lock-in via syntax as much. If Oracle's query language were completely unique, developers might avoid it in favor of something more standard. We saw that even Microsoft, which historically was happy with T-SQL differences, started adding compatibility: e.g., SQL Server 2016 added support for STRING_SPLIT (like a table-valued function) because developers used to rely on such functionality in other DBs or via common code. Microsoft also made their SQL Server on Azure support a "hyperscale" mode that's PostgreSQL (Hyperscale Azure is basically Postgres), again highlighting that even Microsoft sees value in offering the Postgres dialect in addition to T-SQL.

Shrinking Extensions in Practice: A concrete metric is to look at how new projects choose their database and how they write SQL:

- Many new projects default to either Postgres or MySQL for relational storage due to open source, cost, etc. These databases (especially Postgres) are very standards-compliant, and developers writing for them usually write ANSI SQL with maybe a few Postgres conveniences (like ILIKE for case-insensitive, or :: casting syntax). But even those conveniences, if the project cares about portability, they avoid. For example, in documentation for frameworks you often see advice: don't use MySQL's STR_TO_DATE, use standard CAST/TO_DATE if available; don't use T-SQL TOP

without ORDER BY, use FETCH FIRST N ROWS if possible.

- The idea of **database-agnostic applications** is more mainstream. A Stack Overflow discussion asked about writing portable SQL and the consensus was you can stick to a subset that is common (the estimate given was ~80% of features) . Data types might differ, but if you stick to basic ones (INT, VARCHAR, etc.), you'll be fine across DBs. This means new codebases often intentionally avoid exotic vendor types (like Oracle's nested table or Postgres' array types) unless they really need them, because they know it reduces portability.

- Another anecdotal trend: in communities like Ruby on Rails or Django or Go's database/sql, the focus is on supporting multiple databases with one codebase, which forces using the lowest common denominator of SQL. Rails, for instance, historically supported MySQL, Postgres, and SQLite out of the box, so its migration DSL avoids anything that all three can't do. This results in a least-common-denominator approach (e.g., it won't use partial indexes or specific syntax that's not in all). This indeed "shrinks" vendor-specific extension usage at least in the code the framework generates or encourages.

That said, **counterpoints** (which we'll cover more in the counter-evidence section) include: certain vendor-specific extensions are still very entrenched in specialized use. For example, PL/SQL (Oracle's procedural extension) is still heavily used for in-database logic at companies that rely on Oracle. T-SQL has features (like the MERGE in older version was different and had bugs, or ON UPDATE triggers with specific syntax) that are used by MSSQL-focused shops. But these are often legacy or limited to companies committed to one database. New startups or new code often avoid tying themselves that way, precisely to maintain flexibility.

Extensions in New Codebases might refer also to new versions of code. If an existing codebase is being actively developed, do developers remove vendor-specific SQL and replace with standard constructs? Sometimes, yes: for example, after MySQL 8 came out with CTE and window support, some projects refactored their queries that previously relied on non-standard tricks (like using variables for running totals) to now use standard window functions, making the query more portable to, say, Postgres. Another example: a project that used Oracle's syntax might adapt to use more standard syntax if they plan to support Postgres as well (for cost-saving migration). There is indeed a movement in many organizations to migrate from commercial to open source databases; as part of that, code gets rewritten to shed vendor-specific parts. Oracle to Postgres migrations are very common now, with companies like EDB providing tools to convert PL/SQL to PL/pgSQL. So one could say the prevalence of Oracle-specific or DB2-specific SQL in new development is waning, as companies either stay and use standard features or move away.

Extension Areas Still Remaining: The only area where vendor-specific languages remain is often in **procedural extensions and admin syntax**. SQL standard has SQL/PSM (persistent stored modules) for procedures, but hardly anyone uses that exact syntax. Instead, Oracle uses

PL/SQL, SQL Server uses T-SQL dialect for stored procs, Postgres has PL/pgSQL. Those are not unified. But interestingly, some new systems avoid stored procs entirely in favor of external logic (microservices trend etc.). So not using those at all is one way to avoid vendor lock-in. If one must, they accept the lock-in. But it's less fashionable now to put a lot of business logic in DB stored procs, compared to the 90s client-server era. That cultural shift means *in new codebases,* reliance on those vendor-specific procedural languages is less than before. It still exists in particular industries (banks on Oracle, etc.), but new SaaS products often limit stored proc use for maintainability and portability.

In summary, **yes, vendor-specific SQL extensions are generally shrinking in prominence in new development.** The converging forces are:

- **Standards evolution** providing standardized ways to do things that previously required vendor features.

- **Multi-database support** becoming a norm for frameworks and tools, encouraging the common denominator SQL.

- **Conscious avoidance of lock-in** by developers who have seen the difficulties of migrating a locked-in system.

- **Open source adoption** meaning the community gravitates to the features available in popular OSS DBs (which themselves adhere to standards closely).

- **Standard-setters incorporate widely-used vendor extensions**, effectively turning them into standard (as SQL:2023 did for functions like ANY_VALUE and GREATEST which originated in MySQL and Oracle respectively).

Where vendor-specific extensions still appear (e.g., in very new features such as Oracle adding a proprietary syntax for vector search maybe, or a specific function for blockchain data in SQL Server), those are niche and likely transient until a standard catches up if the feature proves broadly useful.

Thus, new codebases in 2025 are far more likely to be written in a database-agnostic SQL style than code from 2005. As one Reddit commenter put it: "All database vendors have some variation… That being said, it's all pretty close… variation in more complex functionalities" . New code tends to avoid those complex bits or handle them carefully. Another commented that sticking to ANSI standard yields about "80% compatibility" and that indeed the core (DML, DDL) is essentially standard across vendors . This illustrates that most of what typical application code does is achievable in a standard way on all major DBs. So developers naturally stick to that to maximize compatibility (and likely also because it's easier to Google solutions that work for everyone rather than something obscure and vendor-specific).

Year-Over-Year Improvements in Cross-Engine Query Portability

Assessing query portability between engines over the period 2020–2025, the trajectory has been one of steady improvement. Several dimensions contribute to this:

1. Feature Parity and Standard Adoption: By 2020, the major relational databases had nearly all converged on supporting the core of SQL:1999 and SQL:2003 (which introduced things like CTEs, window functions, etc.). But a few laggards existed (e.g., MySQL didn't have CTEs or window functions until version 8 in 2018). Once MySQL 8 became widespread in 2020–21, it eliminated a common portability headache – previously, a query using a CTE (WITH ...) would run on Oracle, Postgres, SQL Server, but not MySQL 5.7. After MySQL 8, that query runs on MySQL too. Similarly, window functions and JSON functions in MySQL 8 mean those queries are portable to Postgres, Oracle, etc., since those also have them (Oracle had since ~2012, SQL Server 2012 also added the last missing ones like LEAD/LAG). So around 2020, a significant milestone was achieved: **all major SQL engines supported the majority of SQL:2003 analytics features**.

Year by year, this got better with updates:

- 2021: MySQL 8 adoption grows, meaning devs target its features confidently. MariaDB (a MySQL fork) also implements many of the same features, ensuring that fork didn't fragment capability.

- 2022: Oracle and SQL Server added some SQL:2016 features. SQL Server 2022, for instance, added the STRING_SPLIT function returning a table, which while not exactly standard, addresses a common need in T-SQL that earlier required workarounds. Also, SQL Server 2022 finally added approximate count distinct (APPROX_COUNT_DISTINCT) which had been in some others – showing ideas cross-pollinating.

- Also around 2021–2023, we saw standardization of previously non-standard areas (like JSONPath in 2024, GQL in 2024 as discussed). These efforts, while new, mean that the coming years will see even more engines align, which directly improves portability for those query types.

2. Polyglot Query Engines: An interesting development is the rise of **query engines that can query multiple backends** (Presto/Trino, Spark SQL, etc.). These engines present a unified SQL interface that can query data from different sources (Oracle, Hive, Elasticsearch, etc.). They often implement a subset of SQL that's similar to one database (Trino follows more of a PostgreSQL/Oracle style). As users get used to writing one query that fetches from many sources, it forces them to use only the constructs supported by the engine – which typically are the standard ones. This indirectly encourages data stores to be queryable via those standard

constructs. For example, Trino can query Elasticsearch by internally translating SQL to ES queries; it expects certain things to work (like basic SELECTs, filters). If ES didn't have the SQL plugin, it wouldn't be possible. But ES did make one. Similarly, many NoSQL systems added **SQL-like layers** (e.g., Cassandra with CQL, which intentionally mimics SQL's DDL and DML to lower learning curve; or MongoDB's BI Connector which lets you use SQL to query Mongo via an engine that maps it to aggregation pipeline). This overall environment is making SQL a lingua franca not just within relational DBs, but across many data stores – and by necessity, a *standard-ish* SQL, because it has to work across heterogenous systems. So query portability is not just improving among relational engines, but even between SQL and non-SQL systems via these interfaces.

3. Fewer Breaking Differences: Some of the things that used to trip up cross-engine queries have been mitigated over time:

- **Case sensitivity and quoting rules** – still differ (MySQL by default case-insensitive for identifiers on Windows, etc.), but frameworks handle those and developers know to avoid using reserved keywords or odd casing. It's more about best practices now (like always quote identifiers or never quote them and stick to lower-case).

- **NULL handling and functions** – Standard SQL has COALESCE and NULLIF, which are well-supported everywhere (instead of, say, NVL which was Oracle-only – though NVL is still there, people tend to use COALESCE for portability). Similarly, use of CASE expressions (ANSI) is preferred over proprietary DECODE (Oracle) or IIF (SQL Server) – and indeed, developers increasingly do so. By 2025, any knowledgeable SQL user will use CASE instead of vendor-specific conditional functions, because they know it works on all engines and is part of SQL standard.

- **Date and time** – these were an area of pain historically (different date functions). Now, the standard EXTRACT(year FROM date) is widely supported. The standard INTERVAL types are somewhat supported (some like Postgres and Oracle support them, MySQL has INTERVAL in its own syntax but not fully as type, SQL Server lags on interval). But workarounds exist (like using integer of days, etc.). Even something like CURRENT_TIMESTAMP which was standard but some had different ways – nowadays all engines support CURRENT_TIMESTAMP or an equivalent.

- **Data types** – cross-engine creation of tables might differ (e.g., number precision, etc.). But ORMs often hide this, and many types converge (VARCHAR, INTEGER are everywhere; if you stick to those, you're fine). In 2020-2025, we haven't got a new basic type that's only on one engine – except maybe some NoSQL or extended types like geospatial or JSON (but JSON now standardized as type as discussed).

4. Cross-Engine Tools and Compatibility Layers: The ecosystem has more tools to facilitate cross-engine usage:

- Example: **Hibernate and JPA** in Java have dialects for each DB, but they allow one codebase to target multiple DBs with only config changes. They had to overcome differences by adjusting SQL generation per vendor. Over time, even that got easier because vendors implemented what the frameworks expect. The frameworks themselves also influenced vendors; for example, if a framework heavily uses CTEs and a certain database doesn't have them, that DB might implement them to not be left out.

- **Migration toward open source**: Many enterprises moved from, say, Oracle to Postgres to escape licensing costs in the last 5 years. To do so, they needed their queries to run on Postgres. That encouraged them to replace proprietary syntax with standard or Postgres-compatible syntax. Services like AWS DMS (Database Migration Service) have features to automatically translate some Oracle SQL to Postgres SQL. This trend effectively **converts vendor-locked code into portable code** as part of modernization. We can infer that year by year, more legacy code is refactored for portability.

Empirical example: Suppose in 2020 you had a library of queries for reports, and you wanted them to run on both Oracle and PostgreSQL. In 2020, you might face differences like:

- Oracle's LISTAGG vs Postgres's STRING_AGG function for string aggregation. By 2021, PostgreSQL 14 added STRING_AGG(... ORDER BY ...) to match Oracle's ordered aggregation, and Oracle 19c had LISTAGG with overflow handling which Postgres can emulate. If the standard will include a generic array_agg or something in the future, they both align. So a report query using LISTAGG in Oracle would need change for Postgres – but if you wrote it with the standard-ish approach (common table subquery and string_agg) it can now run on Oracle too (since Oracle supports LISTAGG which is standard now in SQL:2016).

- A query using T-SQL's OFFSET 0 FETCH NEXT N ROWS ONLY vs MySQL's LIMIT. These were unified in the standard (the FETCH/OFFSET approach is ANSI SQL:2008). MySQL added support for WITH TIES and OFFSET/FETCH in 8.0. SQL Server added OFFSET/FETCH in 2012. So by 2020, all engines have some support for a standard way to do pagination. Developers increasingly use the standard syntax over the older vendor ones. In 2025, if someone needs pagination, they might use OFFSET/FETCH knowing it works on SQL Server, Oracle (12c+), Postgres, etc. whereas in 2010 they'd have to write TOP/LIMIT per vendor.

Quantitative Perspective: If one were to measure the percentage of SQL queries that can run unmodified on multiple DBs:

- Back in early 2000s, complex queries often had vendor-specific syntax. Maybe only 50-60% of non-trivial queries were portable due to differences in join syntax (some old engines needed outer join operator differences), analytic functions absent in some, etc.

- By 2010s, thanks to SQL-92 and SQL-99 being widely implemented, that rose. A study or anecdotal evidence suggests maybe ~80% as one commenter estimated . Basic CRUD and moderate complexity selects were fine, only advanced ones needed rework.

- By 2025, that number could be even higher (for the subset of features commonly used). Most differences now are in seldom-used features or extremely engine-specific stuff (like Oracle's CONNECT BY, which is rarely used now in new code since recursive WITH is available).

Non-Relational Query Portability: A quick note that outside the SQL RDBMS world, GraphQL became popular for API queries, but that's a separate domain and not directly relevant to cross-engine SQL portability. However, it emphasizes how users expect to not worry about underlying source differences. Similarly, for analytics, tools like dbt (data build tool) let you write mostly standard SQL models and it compiles them to your target data warehouse's SQL dialect. dbt has macros to abstract differences, and they've found that differences are small enough to manage. As warehouses (BigQuery, Snowflake, Redshift, etc.) adopt more standard functions (or dbt provides shims), analysts can write largely portable SQL models.

All told, **year-over-year, the direction is clear**: more engines implementing more of the SQL standard (including recent expansions), and more consensus on best practices for writing portable SQL. As Dave Stokes wrote in 2024 reflecting on SQL's 50th anniversary, SQL thrived by "keeping up with what users need now" like JSON and graph support – which implies those features become available across platforms, not just one. With each passing year, the fraction of queries that are truly engine-dependent shrinks. For example, a specific proprietary syntax you learned in Oracle in 2010 might now either be standardized or have an equivalent in other engines by 2025, making your query portable with minimal changes.

In a direct answer: **Yes, cross-engine query portability has been improving year-over-year, especially from 2020 to 2025**, driven by standardization of previously proprietary features, conscious development of new databases to be SQL-compliant, and a general industry preference for openness and flexibility. There will always be some edge differences, but for the common use cases, writing a query once and running it on any major SQL platform is more feasible in 2025 than it ever was before.

Counter-Evidence: Persistent Vendor-Specific Variations

Despite significant progress in homogenizing database query syntax, it's important to acknowledge areas where **vendor-specific dialects and incompatibilities remain entrenched** or even have expanded. This counter-evidence underlines that complete convergence is a moving target:

- **Stored Procedures and Procedural SQL Dialects:** As noted, the ISO SQL/PSM standard for procedural language never gained full traction. Oracle's PL/SQL, SQL

Server's T-SQL, and PostgreSQL's PL/pgSQL remain quite different. A stored procedure written in PL/SQL will not run on SQL Server or Postgres without substantial rewriting, and vice versa. Many enterprises continue to use database-specific stored procs for critical logic (often for performance reasons or historical design). For example, Oracle PL/SQL supports things like associative arrays, CURSOR FOR loops, etc., that have no direct equivalent in T-SQL or PL/pgSQL. While new projects might avoid heavy stored-proc usage, **existing codebases (which still evolve) might double down on these extensions**. There's little evidence of convergence here; if anything, each vendor extended their procedural language in recent versions (Oracle adding JavaScript execution inside PL/SQL in 21c, for instance!). So portability at the procedural level is still very low.

- **Vendor-Specific Analytics/Extensions:** Cloud data warehouses, while SQL-based, do introduce new non-standard extensions in specialized areas. For instance:

    - Google BigQuery has SQL extensions for machine learning (CREATE MODEL statements) and geospatial analysis (e.g., ST_GeogPoint), which are not part of ANSI SQL. These are unique to BigQuery, so queries using them won't port to, say, Snowflake or Redshift. Snowflake has proprietary functions for semi-structured data (like GET_PATH for Variant types) and unique syntax like FLATTEN table function. These are niches but in those niches, portability suffers. A BI query using Snowflake's TO_VARIANT or BigQuery's ML.PREDICT will only run on that platform.

    - Oracle and MS SQL, while adopting many standards, still have their own analytic library extensions – e.g., Oracle's MODEL clause (for inter-row calculations) is proprietary (though very powerful), and if a query uses it, no other DB supports it. Microsoft's T-SQL has things like the CUBE/ROLLUP groupings (which are standard, yes, but also an older syntax) and some specialized window function extensions (e.g., row mode vs batch mode differences).

- **Lack of Universal Compliance:** No database is fully compliant with SQL:2016 or SQL:2023. Each skips certain optional features or deviates. For example, PostgreSQL still hasn't implemented the SQL standard MERGE (as of 15, though planned for 16 perhaps); it uses its own INSERT ... ON CONFLICT syntax for upsert, which is non-standard. So if you write an upsert using ON CONFLICT (Postgres-specific), that won't work on others; if you use standard MERGE, that won't work on Postgres until it implements it. MySQL's INSERT...ON DUPLICATE KEY UPDATE remains non-standard. Developers sometimes prefer the vendor-specific upsert if it's more convenient or efficient. Thus some new code may still use these non-standard constructs, perpetuating portability issues.

- **Differences in Default Behaviors and Optimizer Hints:** Even when syntax aligns, semantics can differ. For instance, the way SQL Server handles NULLs in unique

constraints (it treats multiple NULLs as duplicates by default in some cases) vs Oracle (which treats NULLs as distinct) can affect portability of constraints and thus query assumptions. Collation and case sensitivity differences mean an ORDER BY might yield different order in Oracle vs Postgres for text with different case or accents. These subtle differences can break applications when porting. Some have gotten better (SQL:2008 added standardized COLLATE clause; now many support it), but not completely.

- **Legacy Systems and Vendor Lock-in Features:** A lot of enterprise systems are built on specific DBMS and use proprietary features for performance or convenience:

    - IBM DB2 and Informix have their own extension sets that aren't widely adopted elsewhere. For example, Informix has a time series data extension with its own query syntax. DB2 supports the OLAP functionality but also has unique syntax like hierarchical queries using non-standard keywords prior to adopting ANSI recursive SQL.

    - Teradata (before getting merged into Actian) had its own dialect nuances. Netezza (IBM PureData) used standard SQL but had custom functions for analytics.

    - These may not be "new codebases", but code is continuously updated and often remains vendor-specific because the cost to generalize is high and there's no incentive if they're staying on that platform.

- **Graph Query Languages** – while we herald ISO GQL, as of 2025 it's new. In practice, Gremlin (Apache TinkerPop) and SPARQL (for RDF graphs) remain in use and are not aligned with GQL. If your application used Gremlin traversals, you can't just switch to GQL without a rewrite. Some graph DBs like JanusGraph, which use Gremlin, have not announced moves to GQL yet. Also, GQL and SQL/PGQ are separate; some vendors might favor one approach over the other, causing some split in graph querying. For example, Oracle supports SQL/PGQ inside SQL, but Neo4j likely will support GQL as a standalone. Those two are intended to be compatible at some level (GQL and SQL/PGQ share concepts), but differences might emerge in how they're used. Thus, in the short term, graph queries still lack portability – a Cypher query can't run on TigerGraph's GSQL until both converge to GQL, which will take time and new product versions. So for the moment, vendor-dominance in graph query languages still exists (Cypher for Neo4j, GSQL for TigerGraph, etc.), even though convergence is on the horizon.

- **JSONPath Implementation Gaps:** Even though RFC 9535 is out, not all databases immediately conformed. For example, SQL Server's JSON implementation still doesn't support JSONPath filter expressions as of 2025, and it may not incorporate the new standard soon. MySQL's JSON functions may not yet support all the fancy JSONPath features (like arithmetic in filters or regex matching) that the RFC allows. So if you write a complex JSONPath (now standard), it might run in Postgres 16 (assuming full

implementation) but not in MySQL 8.1 if MySQL hasn't caught up, or vice versa with certain syntax differences (MySQL uses $.** for recursive descent? Actually MySQL 8 doesn't support ** at all; the standard uses ..). So in practice, ensuring a JSONPath expression works the same on all engines in 2025 might still require limiting to the intersection of what's implemented. Full consistency probably a year or two away as engines adopt test suites.

- **SQL Extensions for Search/NoSQL:** SQL interfaces to NoSQL (like Cassandra's CQL, or SQL for Mongo via connectors) often only implement a subset of SQL. If you try to use advanced SQL features there, it won't work. For example, Cassandra CQL looks like SQL but doesn't support JOINs or subqueries (because of distributed design). So if you write portable SQL that includes joins, you can't expect it to work on Cassandra. This is more about not all "SQL-like" systems supporting full SQL, which complicates portability if your ecosystem includes those. One could argue those aren't relational engines, but they market themselves as such sometimes. And even within NewSQL: Google Spanner's SQL has some limitations (no foreign keys initially, etc.). So moving a query from Oracle to Spanner might require adjustment if it uses unsupported constructs. Essentially, not every engine implements 100% of standard SQL, so one must still be aware of lowest common denominator.

- **Cultural/Optimizational use of vendor features:** Developers and DBAs often use vendor-specific hints or functions to optimize performance. E.g., Oracle's /*+ INDEX(table idx_name) */ hint, or SQL Server's OPTION (HASH JOIN) hints, etc. Those are by nature not portable. If performance tuning relies on these, migrating that query to another system needs a whole new tuning approach. So in an operational sense, queries in production might still be tied to vendor-specific idioms.

In summary, **while cross-engine SQL uniformity has grown, significant counterpoints remain**:

- **Deeply embedded vendor-specific code** in legacy systems that continue to be updated (they don't get spontaneously rewritten to standard SQL, because "if it isn't broken, don't fix it"). So a lot of enterprise SQL code is still vendor-specific and will remain so until those systems retire.

- **Areas not fully covered by standards** (procedural logic, certain analytics, new domains like graph or ML) where vendors implement their own approach.

- **Lag in adoption of new standards** – standards solve consistency only if implemented. There's always a time lag and some vendors might choose not to implement certain optional features if they deem them low priority.

Therefore, it's not a completely linear march to homogeneous syntax. Some fragmentation persists and even new fragmentations appear when vendors innovate outside the standard's current scope. We should expect that **complete portability is still not automatic**: careful testing and often minor query adjustments are needed when moving between engines. Real-world experiences often involve subtle bugs when porting, e.g., the same query yielding slightly different results or performance, requiring vendor-specific tuning. The improvement in portability is clear, but these counter-examples emphasize that we haven't reached a one-size-fits-all SQL nirvana yet.

The **balance of evidence** is that vendor dominance in syntax is reducing, but not gone. Developers and architects must still be mindful of dialect differences, especially outside the core SELECT/INSERT/UPDATE statements. And as vendors continue to extend SQL for emerging needs (like built-in AI predicates or JSON manipulations), those extensions initially will be vendor-specific until standardization catches up again – a cycle that means at any given time, some part of "cutting edge" SQL is not uniform.

Ultimately, the counter-evidence does not overturn the trend toward homogenization, but it does illustrate the *limits* of that homogenization in 2025 and the areas requiring caution or further alignment efforts. Query portability has improved, yet writing a database-agnostic application is still not trivial in all cases – diligent attention to the aforementioned pitfalls is necessary to avoid being caught by vendor-specific behaviors . The presence of these vendor-specific variations underlines why standards work (SQL:2023, GQL, JSONPath, etc.) remains crucial and why database users still sometimes find themselves grappling with dialect nuances in practice.

Footnotes:

[^1]: Dave Stokes, *"SQL at 50: A Lesson in How to Stay Relevant Around Data,"* Dataversity (May 1, 2024) – Highlights new SQL:2023 features and their significance .

[^2]: ISO/IEC 9075:2023 (SQL:2023) – Introduced Property Graph Queries (SQL/PGQ) bridging relational and graph models , native JSON type and functions , and standardized various common vendor functions .

[^3]: *The Register* – Lindsay Clark, *"Graph databases speaking the same language after ISO gives GQL the nod,"* (Apr 24, 2024). Discusses ISO GQL standard and vendor support (Neo4j, TigerGraph) .

[^4]: AWS Database Blog – Philip Rathle & Brad Bebee, *"GQL: The ISO standard for graphs has arrived,"* (Apr 25, 2024). Joint letter by Neo4j and AWS on GQL, noting Cypher and GQL convergence .

[^5]: NebulaGraph Blog – *"NebulaGraph Enterprise v5.0: The First Distributed Graph Database to Offer Native GQL Support,"* (Apr 17, 2024). Announces early adoption of ISO GQL .

[^6]: Stefan Gössner et al., *"JSONPath: Query Expressions for JSON,"* IETF RFC 9535 (Feb 2024). The official JSONPath standard, aiming to reconcile differences among ~50 implementations .

[^7]: Glyn Normington, *"JSONPath: from blog post to RFC in 17 years,"* IETF Blog (Feb 21, 2024). Describes the standardization process; notes that convergence will be gradual and lists initial compliant implementations .

[^8]: BigData Boutique Blog – Shai Greenberg, *"SQL Joins in Elasticsearch and Kibana,"* (Nov 24, 2022). Confirms that as of 2022, **"SQL JOIN is not supported by Elasticsearch and OpenSearch"**, and discusses workarounds .

[^9]: Elastic.co (Elasticsearch Labs) – *"Native joins available in Elasticsearch 8.18,"* (Aug 13, 2025). Announces tech preview of **lookup join** in Elasticsearch, noting SQL join support had been elusive until now .

[^10]: Huawei Cloud Docs – *"Using SQL to Search for Data in OpenSearch,"* (2021). Shows OpenSearch SQL supports INNER JOIN, LEFT JOIN, CROSS JOIN with limitations .

[^11]: Reddit – u/SamAndFrodo4Ever, *"Standard SQL supported by all major vendors, can there be?"* r/SQL thread (1 year ago). Discusses SQL portability: *"it's all pretty close to the ANSI standard… more variation as you get into complex functionalities (dialects of same language)."* .

[^12]: Reddit – u/JeegReddit44, same thread as above. Estimates *"ANSI standard will get you ~80% compatibility… DML and DDL essentially standard, but each vendor adds extensions… Analytical functions and stored procedures… not included in standard and [are] most notable differences."* .

[^13]: Markus Winand, *"Modern SQL,"* modern-sql.com. Emphasizes writing portable, standard SQL and documents cross-vendor support for modern SQL features .

[^14]: Oracle Blogs – *"Property Graphs in Oracle Database 23c: The SQL/PGQ Standard,"* (2023). Notes Oracle 23c's implementation of SQL property graphs aligns with SQL:2023's PGQ (subset of PGQL, with plans to extend) .

[^15]: Oracle-Base (Tim Hall), *"SQL Property Graphs and SQL/PGQ in Oracle Database 23c,"* (2023). Provides examples of Oracle's SQL/PGQ usage with GRAPH_TABLE() queries .

[^16]: Christoph Burgmer, *"JSONPath Comparison,"* (archived Mar 2024) – Comparative test results of many JSONPath implementations, illustrating wide discrepancies and lack of consensus prior to RFC .

[^17]: PostgreSQL Mailing List – Peter Eisentraut, *"SQL Property Graph Queries (SQL/PGQ),"* pgsql-hackers (Feb 16, 2024). Introduces a prototype implementation of SQL:2023 graph queries in Postgres .

[^18]: TigerGraph Blog – Mingxi Wu, *“The Rise of GQL: A New ISO Standard in Graph Query Language,”* (2024). States that GQL addresses the proliferation of proprietary graph query languages and will be foundational for the industry .

[^19]: Exasol Documentation – *“Compliance to SQL Standard,”* (2023). Enumerates which SQL:2023 features are supported or partially supported in Exasol, exemplifying vendor transparency in adopting new standard features .

[^20]: *SQL:2023 – Wikipedia.* Summarizes new features in SQL:2023, including JSON enhancements and property graph queries .