Converging Access Patterns: How SQL, KV, Document, Graph, and Search Are Unifying Across DBMS

Executive Summary

Access patterns are converging across data stores. Modern database systems increasingly support multiple data models and query interfaces within a single platform. Top cloud-managed databases are evolving beyond single-model APIs: for example, Azure Cosmos DB offers SQL (document), MongoDB, Cassandra (wide-column), Gremlin (graph), and Table (key-value) APIs under one service . This multi-model approach lets developers use familiar languages (SQL, MongoQL, Gremlin, etc.) on the same backend, reducing polyglot persistence complexity. Similarly, multi-model databases like ArangoDB, OrientDB, and Couchbase combine document, key–value, graph, and even search capabilities in one engine . These unified offerings address the rise of diverse data (JSON, graph relationships, text) in modern applications while simplifying operations.

API gateways are bringing multi-API access to production. Technologies such as DataStax **Stargate** sit in front of databases (Apache Cassandra, in Stargate's case) to expose *REST, GraphQL, document (JSON), and gRPC* APIs alongside the native query language . This allows one database cluster to appear as different interfaces – e.g. using Cassandra as a document DB via JSON API or as a graph via GraphQL. In practice, Stargate is used at scale: it's built into DataStax AstraDB (the managed Cassandra service), and companies like Netlify and PrepLadder have leveraged Stargate's GraphQL/REST endpoints to speed up development . Open-source adoption has grown (7× downloads increase in a year) , indicating real-world interest in API gateways for database flexibility. While not yet ubiquitous for all databases, in Cassandra's ecosystem Stargate demonstrates that multi-API gateways *are* viable in production.

The Redis protocol has become a de-facto standard for key–value stores. Originally the interface for Redis, its simple command protocol (RESP) is now implemented by many systems to ensure compatibility with Redis clients and skills. Emerging high-performance caches and KV stores (e.g. DragonflyDB, KeyDB, **rocksdb-server**/KVRocks) explicitly advertise **Redis-compatible APIs**, allowing drop-in replacement of Redis . For instance, engineers at Abnormal Security deployed a RocksDB-based KV service using the Redis protocol so they could reuse existing Redis client libraries and tools . They note that the Redis protocol even has built-in support for clustering/sharding, easing scale-out . Likewise, cloud offerings (AWS ElastiCache, Azure Cache) support "Redis" as a service, and projects like Netflix Dynomite use

a Redis-compatible interface for distributed KV. This broad adoption suggests Redis's wire protocol is the closest thing to a lingua franca in the key–value/cache world – analogous to SQL for RDBMS . While not every vendor uses Redis's protocol (some use Memcached or custom APIs), the trend leans toward Redis compatibility as the default for new KV systems.

In graph databases, the new ISO Graph Query Language (GQL) standard is poised to unify and surpass prior languages (Gremlin, Cypher). ISO GQL was officially published in late 2023 as the first new ISO database query language since SQL , marking a milestone in graph DB maturity. It builds heavily on the success of **Cypher** (Neo4j's declarative pattern-matching query language) – which had already become a de facto standard used beyond Neo4j (e.g. openCypher in AWS Neptune) . GQL adopts Cypher's ASCII-art pattern syntax and aligns with SQL's style, providing a standardized, vendor-neutral graph query language . Early signs show industry uptake: for example, in 2025 the analytics platform Siren adopted ISO GQL as its query interface, calling GQL *"the* industry standard" and crediting it with unifying full-text search, SQL-style relational queries, and graph traversals in one query plane . With GQL, new graph applications can avoid fragmentation – developers no longer face a stark choice between Gremlin's imperative traversals and various dialects of Cypher. Given that Neo4j (the market leader) plans to converge Cypher into GQL , and other vendors like Memgraph and Oracle are on board, ISO GQL is likely to quickly overtake Gremlin and proprietary languages in new deployments. Gremlin remains in use (especially with Apache TinkerPop/JanusGraph), but the momentum in new projects is shifting toward declarative GQL/openCypher due to ease-of-use and standardization.

SQL interfaces for search engines have emerged and see real use in production scenarios, though not as the primary query method. Systems like Elasticsearch and its open-source fork OpenSearch now "speak SQL," allowing users to query indexed documents with SELECT/FROM/WHERE syntax. This SQL API is not just a toy – it enables integration of search-backends with BI tools and the skills of SQL-trained analysts. Elastic's SQL feature, for instance, can expose full-text search on petabytes of data through a familiar tabular query paradigm . Users can send SQL to Elasticsearch via REST, ODBC, or JDBC, and get results in standard tabular format . This is used for interactive analytics dashboards (Elastic's Canvas) and by applications that prefer SQL abstraction over JSON DSL . Amazon's OpenSearch Service likewise provides a SQL workbench and translates SQL to the native query DSL . In practice, organizations have leveraged these SQL layers to let non-developers query log data or metrics indexed in search engines using familiar SQL (instead of learning Lucene syntax). That said, the SQL support typically covers read-only queries and has limitations (not all text-search features map neatly to SQL). But its existence in production environments – bridging two previously separate worlds – underscores the trend of **SQL as the universal "query glue"**, even for NoSQL search stores.

SQL and NoSQL document stores are converging on common JSON querying standards (SQL/JSON and JSONPath). Over the past decade, all major relational databases added native JSON support, blurring the line with document databases. The SQL:2016 and SQL:2023 standards introduced **SQL/JSON** features: a JSON data type, functions like JSON_VALUE/JSON_QUERY, and a **JSONPath** syntax for navigating within JSON documents .

This JSONPath (a string syntax akin to XPath for JSON) is used in SQL conditions and functions to extract nested fields ($.store.book[0].title etc.) . It has become widespread across relational DBMS: e.g., Oracle, MySQL, and SQL Server all accept JSON path expressions in queries , and PostgreSQL supports JSONPath through its @@ operator and jsonb_path_query functions (following the SQL standard) . In parallel, an IETF standard for JSONPath (RFC 9535) was published in 2024, solidifying a common syntax for JSON queries across platforms . On the NoSQL side, document databases have their own query methods but are increasingly aligning too – for example, Couchbase's N1QL is a SQL-based query language for JSON, and Azure Cosmos's SQL API uses SQL-like syntax to query JSON items by properties . We now even see queries that join SQL and JSON: for instance, SQL Server's OPENJSON and **FOR JSON** allow relational rows to be converted to JSON and vice versa, enabling hybrid use of SQL and document patterns in one system . In practice, JSON usage is *bridging SQL and NoSQL*: developers can choose a relational DB and still store/query schemaless JSON when flexibility is needed, using standard JSONPath expressions across products . Likewise, some NoSQL document stores (like MongoDB via BI adapters or emerging SQL-on-Mongo tools) allow SQL access to their JSON data. Thus, SQL/JSON and JSONPath have become common tools in both camps, reducing the need to choose one paradigm over the other. It's increasingly **common to see SQL and document stores sharing JSON query techniques**, making semi-structured data a first-class citizen in SQL databases and allowing document databases to integrate with SQL-based tooling.

Counter-Evidence and Limitations: Despite clear convergence trends, there remain important caveats:

- **Separate DB services remain the norm in clouds:** While platforms like Cosmos DB are multi-model, major providers (AWS, Google) still offer specialized databases for each workload (relational, key-value, graph, search) rather than one engine exposing all APIs. This means many "top managed DBs" do *not* yet provide *every* interface natively. For example, AWS has distinct services (Neptune for graph, OpenSearch for search, DynamoDB for key-value, etc.) instead of a single multi-model database – integration is left to the user. The multi-model ideal of one database for all use cases is still emerging. Even Cosmos DB's approach has trade-offs: under the hood it implements MongoDB and Cassandra APIs by mapping them to its core engine, which may not support 100% of each protocol's features. Likewise, Oracle and PostgreSQL added JSON and graph features, but not with the full functionality of dedicated document or graph DBs. Thus, *full* consolidation is not uniform – organizations often still employ polyglot persistence for best-of-breed capabilities.

- **Complexity and maturity issues:** Using one engine for multiple models can introduce complexity and sometimes compromises. A 2023 analysis noted that many so-called multi-model databases "fall short" of delivering equally robust support for all models . For example, a database might excel at document queries but offer only basic graph traversal performance, or vice versa. Multi-model systems are maturing, but one system rarely "fully supports all data models with high performance" yet . This is why some experts argue that current solutions don't yet meet the "true multi-model" vision of

seamlessly handling relational, document, graph, and full-text in one engine without trade-offs . Therefore, in practice, teams must still evaluate if a multi-model DB meets their specific needs or if multiple specialized stores are safer.

- **Stargate and similar gateways are not universally adopted**. While Stargate has traction in the Cassandra ecosystem, most databases do not have an equivalent API gateway layer. Many production deployments are not comfortable adding an extra network hop/layer for data access unless needed. Some use cases prefer direct native drivers for performance. Additionally, running a gateway at scale demands its own maintenance. So, the concept is promising but not yet pervasive outside of Cassandra/DataStax's realm. We have yet to see API gateway frameworks for, say, relational databases or MongoDB in wide use (aside from vendor-specific GraphQL endpoints).

- **Redis protocol dominance has limits**: Although Redis API is common for in-memory and cache use cases, it doesn't cover *all* key-value stores. High-throughput disk-based KV stores (Cassandra, Riak, etc.) use different interfaces and focus on different use cases. Some scenarios require the simpler get/put of Memcached or the rich data structures of Redis – not one size fits all. Moreover, the Redis protocol itself evolves (with new commands/modules), and compatibility can lag in third-party systems. Thus, while Redis API is a strong standard, it's not literally universal across all vendors (e.g., Aerospike has its own client API, and cloud DynamoDB uses HTTP JSON). There is also potential fragmentation with Redis forks after licensing changes. Still, as a de facto standard for cache and ephemeral stores, it's very influential.

- **Graph query adoption is still in transition**: ISO GQL being brand new means many existing applications continue with Gremlin, Cypher, or SPARQL. Gremlin in particular remains entrenched in open-source graph stacks and some AWS Neptune customers. It will take time (and vendor implementations) for GQL to truly surpass all older languages. Moreover, GQL's success depends on industry consensus; if some major players don't implement it or if developers stick to what they know (Cypher), the "unification" could be slow. As of mid-2025, GQL is at the starting line: early adopter platforms like Siren exist , but widespread support (in Neo4j's products, Neptune, TigerGraph, etc.) is still unfolding. Therefore, claims of GQL *surpassing* Gremlin/Cypher might be premature, though the trajectory points that way.

- **SQL for search is not a full replacement for search DSLs**. The SQL interfaces on Elasticsearch/OpenSearch cover basic querying and aggregations, but advanced text search features (complex relevance scoring, nested queries, regex, etc.) often require the native query DSL. Many power users of search engines still prefer the JSON DSL or domain-specific query languages for full control. Additionally, performance considerations arise – the SQL layer might add overhead or not expose all tuning parameters. In production, SQL is commonly used for quick analytics or integration with BI tools, but application developers building search experiences typically use the native

search APIs for richer functionality. So while SQL **is** used in production on search data, it complements rather than replaces the specialized search query languages.

- **JSON in SQL vs. NoSQL trade-offs**: Relational databases can now store and query JSON, but that doesn't always equal the flexibility or performance of native document stores. For instance, scaling massive JSON workloads or doing ad-hoc schema-less queries might still be easier in MongoDB or Couchbase which are purpose-built for JSON. Conversely, document stores often lack the robust JOINs or analytical SQL functions of RDBMS. Thus, the convergence via SQL/JSON is helpful, but each side retains some advantages. A SQL database with JSON might still struggle if you treat it exactly like a schema-free NoSQL at large scale (indexing every field, etc.). Meanwhile, NoSQL systems adding SQL interfaces (like Couchbase's N1QL or Mongo's SQL BI connectors) can incur translation overhead. In short, SQL+JSON features provide common ground, but "one database for everything" might not always yield the best performance or simplicity.

Outlook: In the near term, we can expect further consolidation of access patterns: more databases will advertise multi-model capabilities and multi-API endpoints, as developer demand grows for flexibility. SQL is solidifying its place as an interlingua – from querying JSON in PostgreSQL to querying Elasticsearch – and this will only expand as vendors standardize interfaces. Industry initiatives like ISO GQL and IETF JSONPath show a clear push toward standards that span multiple platforms, which will lower barriers to adopting new database tech (developers can carry skills across systems). We may also see **more cross-pollination** such as graph queries inside SQL databases (the SQL/PGQ extensions) and full-text indexes in traditionally non-search systems, further erasing the boundaries between "SQL" and "NoSQL" engines.

However, specialization won't disappear overnight. Different workloads (transactional vs. analytical, document vs. relational vs. graph) still benefit from optimized engines. The likely future is a middle ground: polyglot persistence simplified by common APIs. For example, a cloud provider might offer a unified gateway that routes SQL, Gremlin, and key–value calls to the appropriate back-end service – giving the illusion of one multi-model DB without forcing one engine to do it all. API gateways like Stargate hint in this direction, as does the integration of technologies (such as using PostgreSQL for both relational and JSON and time-series via extensions). The multi-model trend is clearly accelerating, and **the lines between databases are blurring**. In the next few years, technical professionals can expect the convenience of using standard languages (SQL, GQL, etc.) and protocols (Redis, JSONPath) across many storage technologies – consolidating access patterns even if the underlying storage engines remain diverse. The end result aims to combine the strengths of each model with less friction: letting teams pick the right data model for a problem without having to juggle completely distinct databases and languages for each. The trajectory is set toward more unification of APIs, guided by standards and driven by the practical need to reduce data silo complexity in modern architectures . The consolidation of access patterns is well underway, promising greater

developer productivity and agility – so long as we remain mindful of the current limitations and continue to address them with careful architecture and evolving technology.

Footnotes (Chicago Style):

1. Azure Cosmos DB – Multi-Model APIs and Data Models

2. Rapydo Blog – Examples of Multi-Model DBs (ArangoDB, Cosmos DB, Couchbase)

3. DataStax Stargate Press Release – Stargate exposes JSON, CQL, GraphQL, REST, gRPC on Cassandra

4. DataStax Stargate – Usage in AstraDB and Developer Quotes (Netlify, PrepLadder)

5. Abnormal Security Engineering Blog – Using Redis protocol on RocksDB (justification and benefits)

6. GitHub (Dapr issue comment) – Redis protocol seen as de facto standard for KV stores

7. Neo4j Blog – ISO GQL introduction and Cypher as de facto standard prior to GQL

8. Siren Press (CFOtech) – Siren adopts ISO GQL, calls it industry standard, unifies query paradigms

9. Elastic.co – Elasticsearch SQL feature overview (SQL syntax on search data, via ODBC/JDBC)

10. Microsoft Docs – SQL Server JSON support bridging relational and NoSQL concepts

11. Oracle Documentation – SQL/JSON Path analogous to XPath, enabling JSON queries in SQL

12. DEV Community (Arctype) – JSONPath usage in PostgreSQL (@@ operator, functions like size())

13. ChaosSearch Blog – “Multi-Model Databases: Hype vs. Reality” (current solutions’ limitations)

14. Rapydo Blog – Multi-model DB definitions and vendor landscape (need for unified engines)