Evaluating Query Processing Architectures in Relational and Document Databases

Nimisha Modi[1], Jayshri Patel[2]
[1]*Assistant Professor, Department of Computer Science, VNSGU, Surat.*
[2]*Assistant Professor, Department of Computer Science, VNSGU, Surat.*
doi.org/10.64643/IJIRTV12I2-182656-459

***Abstract*— Query processing architectures in relational and document-oriented databases differ significantly in how they handle core operations like selection, sorting, and joins. Relational databases rely on fixed schemas, cost-based optimizers, and strong ACID guarantees to efficiently process complex queries with multiple joins. In contrast, document databases offer flexible schemas and horizontal scalability, using pipeline-based query execution with denormalized data models optimized for semi-structured data. This comparative analysis explores trade-offs in schema design, indexing strategies, query optimization, scalability, and consistency, providing insights to help choose the best database system for specific application needs.**

***Index Terms*—Query Processing, Relational Databases, Document Databases, Schema Flexibility, Database Scalability.**

I. INTRODUCTION

Today, most applications depend heavily on data, particularly when dealing with large volume and requiring fast responses. Choosing the right database model is crucial to achieve performance, productivity, and reliability in software systems. Among the various data models, two prominent approaches are Relational Database Management Systems (RDBMS) and Document oriented NoSQL Databases. NoSQL databases are designed to handle large volumes of structured, semi-structured, and unstructured data, making them well-suited for applications such as social media platforms where data grows rapidly. Based on the CAP theorem, NoSQL databases are categorized according to the trade-offs they make between consistency, availability, and partition tolerance [3]. The main types of NoSQL databases include Key-Value stores, Document databases, Columnar databases, and Graph databases. In this study, we concentrate only on Document Databases due to their adaptability in managing semi-structured data and their increasing prevalence in morden web applications.

Relational databases (RDBMS) have been the unrivaled
choice for database solutions for application developers for decades. They store data in tables with fixed schemas and use Structured Query Language (SQL) for data manipulation. MySQL, PostgreSQL, Oracle, and Microsoft SQL Server are popular relational databases. These databases maintain consistency and reliability through strong ACID (Atomicity, Consistency, Isolation, Durability) properties, making them ideal for applications like financial systems, ERP platforms, healthcare management systems, and government or regulatory systems where consistency and auditability are critical.

Document-oriented databases have gained popularity for managing rapidly changing or complex data structures. Unlike the normalization-centric approach of RDBMSs, document databases encourage denormalized data models by embedding related data inside documents to enhance read performance [6]. These systems store, retrieve, and manage data as documents—typically in JSON (JavaScript Object Notation) or BSON (Binary JSON) format—supporting hierarchical and nested structures with schema flexibility. Examples include MongoDB, CouchDB, and Amazon DocumentDB. These databases scale easily by distributing data across multiple servers and are well-suited for applications such as content management platforms, user profile storage, and IoT devices.

This paper presents a comparative analysis of query architectures in relational and document databases. It explores core query processing mechanisms such as selection, projection, sorting, and joins. The paper examines trade-offs related to schema design, consistency, and query execution. By drawing on case studies and real-world examples, this work highlights key differences and helps guide the decision-making process when choosing between these two database paradigms.

II. RELATIONAL DATABASES: QUERY PROCESSING ARCHITECTURE AND ASSOCIATED TRADE-OFFS

Query processing in RDBMS involves translating high-level SQL queries into optimized execution plans through parsing, logical and physical planning, and cost-based optimization. Query processing starts with parsing the query to check its syntax. After verifying the syntax, it creates a query plan expressed in relational algebra. The optimizer applies transformation rules such as predicate pushdown, join reordering, and projection pruning to improve efficiency[10].

Physical plans are generated using cost-based estimation techniques, considering table statistics, index availability, and cardinality estimates. Joins can be executed using nested loops, merge joins, or hash joins, depending on the chosen strategy. [12] Execution typically follows an iterator model, with pipelining and buffer management used to support efficient result streaming. RDBMSs excel at handling complex queries involving multiple joins and aggregations, supported by mature indexing strategies and robust transaction controls.

Relational databases excel in providing robust transactional guarantees, structured schema enforcement, and mature query optimization capabilities, but these strengths come with notable trade-offs[1].

A. Data Integrity vs. Schema Rigidity

The fixed schema model of relational databases enforces strict data integrity and type safety which is essential for applications requiring consistent and validated data. However, this rigidity necessitates costly schema migrations and limits flexibility in handling evolving or semi-structured data. [8] This can slow development cycles and increase maintenance overhead.

B. Complex Query Support vs. Performance Overhead

Relational engines support sophisticated operations such as multi-way joins, complex aggregations, and nested queries, empowered by cost-based optimizers and indexing.. These capabilities introduce substantial optimizer complexity and may occasionally result in suboptimal query plans, especially in environments with rapidly changing data statistics. The overhead can impact response times for high-throughput workloads [1].

C. Vertical Scaling vs. Limited Horizontal Scalability

Traditional RDBMSs are optimized for vertical scaling with powerful hardware, which simplifies consistency and transactional support. Horizontal scaling, however, is non-trivial and often requires complex application-level partitioning or middleware support, which can limit scalability and increase system complexity.

D. ACID Compliance vs. Distributed Performance

Strong ACID guarantees [1][12] ensure reliability and data correctness; however, they can reduce throughput and increase latency in distributed environments, particularly when networked storage is involved. This trade-off makes RDBMSs less suitable for large-scale, geo-distributed applications that demand low latency.

III. DOCUMENT DATABASES: QUERY PROCESSING ARCHITECTURE AND ASSOCIATED TRADE-OFFS

Document-oriented databases process queries differently, reflecting their flexible schema and hierarchical document structure. Queries are often expressed in JSON-like syntax using match-filter pipelines (e.g., $match, $group, $project) rather than declarative SQL. The planning stage generates an abstract execution plan, considering whether indexes can accelerate filtering or sorting operations[10].

Execution follows a pipeline model, where documents pass through sequential transformation stages. Each stage applies a filter or transformation, supporting both stateless and stateful behaviors. Operators like $lookup provide limited join functionality between collections, although performance often degrades with complex joins. Document databases benefit from flexible indexing on nested fields, while auto-sharding and replica sets offer built-in horizontal scalability[5]. These databases emphasize schema flexibility [8] and scalability, but they also involve trade-offs that impact query processing and overall system behavior.

A. Schema Flexibility vs. Query Predictability

A flexible, schema-less design enables rapid development [8] and evolution of data models without downtime. However, the lack of enforced schemas can lead to inconsistent data shapes, complicating query optimization and validation. Query performance may vary significantly depending on document structure variability.

B. Simplified Data Modeling vs. Join Limitations

Document databases often rely on Denormalization and embedded documents to avoid expensive join operations. While this approach improves read performance for common access patterns, it can result in data duplication, increased storage costs, and challenges in maintaining data consistency across related entities[5][10].

C. Horizontal Scalability vs. Consistency Guarantees

Document databases are designed for horizontal scaling through auto-sharding and replication, offering high availability and fault tolerance. However, many implementations relax consistency guarantees in favor of eventual consistency, which may not be suitable for applications requiring strict transactional integrity.

D. Query Pipeline Flexibility vs. Optimization Complexity

The aggregation pipeline model supports powerful data transformations and analytics but lacks a mature cost-based optimizer. Query planning is often heuristic or rule-based, which can lead to suboptimal execution plans and unpredictable performance in complex queries.

E. Distributed Performance vs. Transaction Support:

Although recent versions of document databases have introduced multi-document ACID transactions, these features may incur latency and performance penalties compared to single-document operations—posing trade-offs between consistency and throughput.

IV. COMPARATIVE ANALYSIS AND EVALUATION OF QUERY PROCESSING ARCHITECTURES

To better appreciate the differences in query processing, it is essential to first examine the core architectural distinctions between relational and document databases [[3].

A. Database Schema:

Relational databases enforce a strict, predefined schema. All records must adhere to the table structure defined by column types and constraints. This rigidity enables the optimizer to rely on precise metadata, helping enforce referential integrity. However, adapting schemas requires migrations, which can be disruptive for large datasets.

Document databases support flexible schemas. Documents in the same collection can vary in structure, accommodating changes without costly migrations [5]. Developers can iterate quickly by evolving document structures in line with application logic. Over time, this flexibility can lead to document heterogeneity and complicate analytics [1].

B. Query Language and Semantics:

Relation databases use SQL, a declarative language that allows the optimizer to derive efficient execution strategies. SQL's composability, such as views and Common Table Expressions (CTEs), facilitates complex query constructs. Plus, a well-developed ecosystem supports query tuning and static analysis.

Document databases employ JSON-like or pipeline-based queries. These are more imperative and provide less opportunity for algebraic optimization. They are intuitive for developers familiar with JSON and scripting languages. But lack of an algebraic foundation limits the optimizer's ability to restructure queries [6].

C. Join and Aggregation Capabilities:

Relation databases offer full join support and optimized aggregation using GROUP BY, window functions, and subqueries. Cost-based optimizers can choose between hash, merge, or nested-loop joins based on cardinality stats. Window functions enhance advanced analytics within queries.

Document databases prefer denormalized structure and avoid joins. Limited join functionality (e.g., $lookup) and aggregation pipelines support document reshaping. $lookup enables cross-collection joins but is less efficient than relational joins. Aggregations in document stores tend to operate on one document at a time, limiting inter-document correlation.

D. Indexing and Access Paths:

Relation databases provide rich indexing options including B-tree, bitmap, and covering indexes, integrated with a cost-based optimizer. Multi-column and expression indexes further optimize complex predicates. Statistics drive selection of index- vs. full-scan strategies.

Document databases support indexing on nested fields and arrays but rely on heuristic-based query planning. Indexing nested fields helps with document lookups but does not integrate deeply into query planning.

Cardinality estimation is often missing, leading to unpredictable access path choices.

E. Execution Strategy:

Relation databases utilize an iterator model with pipelined operators and sometimes JIT compilation for performance. Operators process tuples on the fly, reducing I/O and memory. JIT can compile hot execution paths for faster loops over data.

Document databases use a pipeline execution model where documents flow through transformation stages sequentially. Each document is processed stage-by-stage, enabling streaming transformations. But without adaptive reordering, pipelines may become bottlenecks for heavy workloads.

F. Query Optimization:

Relation databases rely on advanced cost-based optimizers and query planners leveraging metadata and statistics. Heuristics such as join reordering, predicate pushdown, and materialized view selection are mature. Cost estimates take into account I/O, CPU, and memory usage.

Document databases often depend on rule-based optimizations and lacks advanced cost models, though recent efforts aim to close this gap. Heuristics include pushing filters early and using available indexes. Advanced systems are beginning to add cardinality estimation and adaptive stages.

G. Scalability and Distribution:

Relation databases are designed for vertical scaling. Supports horizontal partitioning and replication in enterprise deployments [2]. Hard partitioning strategies (like range or hash sharding) require careful key selection to balance workload. Replication typically offers strong consistency guarantees.

Document databases are built for horizontal scaling via sharding, supporting distributed processing and elastic scaling. Auto-sharding distributes both storage and query load seamlessly. Rebalancing shards on the fly simplifies scaling at the cost of added network complexity.

H. Aggregation Performance:

Relation databases optimized aggregation engines can leverage indexes and parallelism. Multi-core execution parallelizes GROUP BY and hash aggregation efficiently. Aggregates can be pre-computed in materialized views or summary tables [11].

For Document databases, aggregation pipelines are flexible but may lack deep optimization, impacting large dataset performance. Pipelines can execute map-reduce style but typically lack parallel planning. Performance suffers on large collections without manual tuning [4].

V. CONCLUSIONS

This comparative study highlights the core architectural differences and trade-offs between relational and document databases in the context of query processing. Relational databases provide mature mechanisms for executing complex queries, ensuring strict consistency and maintaining data integrity through well-established ACID guarantees. These strengths, however, come at the cost of schema rigidity and challenges in achieving horizontal scalability. In contrast, document databases offer schema flexibility, horizontal scalability [7], and rapid development cycles, making them suitable for modern, dynamic applications; yet, they often trade off advanced join capabilities, robust query optimization, and strict transactional consistency [6].

The gap between relational and document databases is narrowing with the emergence of hybrid and multi-model systems. One of the emerging trends in enterprise architecture is the adoption of polyglot persistence [11], where relational databases are used for structured, transactional workloads—such as billing, auditing, and user account management — while document databases manage semi-structured or rapidly evolving data like logs, user-generated content , and real-time feeds [9]. This hybrid approach allows organizations to optimize system performance and maintainability by selecting the most appropriate storage model for each use case.

As database systems continue to evolve, the boundaries between paradigms are becoming increasingly blurred. Relational systems such as PostgreSQL[2][13] now support JSON storage and indexing, while document databases like MongoDB [12] are improving their ACID compliance and query planning frameworks. Future developments are expected to further unify capabilities, offering developers more flexible and powerful platforms for data management across diverse workloads.

Future research should explore the integration of AI-driven query optimizers and conduct benchmarking under emerging workloads such as IoT and real-time analytics. These advancements hold significant implications for developers and architects, aiding in the informed selection of database systems best suited to their specific application needs.

REFERENCES

[1] Palanisamy, S., & Suvitha Vani, P. (2020). A survey on RDBMS and NoSQL databases MySQL vs MongoDB. International Conference on Computer Communication and Informatics (ICCCI), 1–7. https://doi.org/10.1109/ICCCI48352.2020.9104047

[2] ElDahshan, K., Selim, E., Ebada, A. I., Abouhawwash, M., Nam, Y., & Behery, G. (2022). Handling big data in relational database management systems. Computers, Materials and Continua, 72(3), 5149–5164. https://doi.org/10.32604/cmc.2022.028326

[3] Dave, M. (2012). SQL and NoSQL Databases. International Journal of Advanced Research in Computer Science and Software Engineering, 2(8), 20–27.

[4] Shichkina, Y., & Ha, M. (2020). Creating collections with embedded documents for document databases taking into account the queries. Computation, 8(2), 45. https://doi.org/10.3390/computation8020045

[5] Mason, R. T. (2015). NoSQL databases and data modeling techniques for a document-oriented NoSQL database. In Proceedings of Informing Science & IT Education Conference (InSITE).

[6] Carvalho, I., Sá, F., & Bernardino, J. (2023). Performance evaluation of NoSQL document databases: Couchbase, CouchDB, and MongoDB. Algorithms, 16(2), 78. https://doi.org/10.3390/a16020078

[7] Čerešňák, R., & Kvet, M. (2019). Comparison of query performance in relational and non-relational databases. In Proceedings of the 13th International Scientific Conference on Sustainable, Modern and Safe Transport (TRANSCOM 2019), High Tatras, Slovak Republic.

[8] Hamouda, S., Zainol, Z., & Anbar, M. (2019). A flexible schema for document oriented database (SDOD). In Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2019) (pp. 413–419). https://doi.org/10.5220/0008353504130419

[9] Soni, D. N. (2023). Digital transformation of libraries: Challenges and strategies in information and library science. International Journal of Research in Humanities & Social Sciences, 11(2).

[10] Khan, W., Kumar, T., Zhang, C., Raj, K., Roy, A. M., & Luo, B. (2023). SQL and NoSQL database software architecture performance analysis and assessments—A systematic literature review. BigData and Cognitive Computing, 7(2).

[11] Kaur, K., & Rani, R. (2015). A smart polyglot solution for big data in healthcare. IT Professional, 17(6), 48–55. https://doi.org/10.1109/MITP.2015.111

[12] MongoDB. (2025). The official MongoDB. https://www.mongodb.com

[13] PostgreSQL Global Development Group. (2023). JSON types — PostgreSQL Documentation. https://www.postgresql.org/docs/current/datatype-json.html