Exploring Migration Strategies: Transforming Relational Databases to Document-Oriented NoSQL Systems

Nimisha Modi
Department of Computer Science,
Veer Narmad South Gujarat University, Surat

ABSTRACT
Software systems developed over the past decade were traditionally designed to incorporate relational databases for managing their data requirements. However, relational databases increasingly struggle to handle the growing volume, velocity, and variety of data in modern applications. In response, document-oriented NoSQL databases—such as MongoDB— have progressively emerged as a compelling alternative, offering greater flexibility and scalability for managing semi-structured data. This paradigm shift has necessitated the migration of existing systems to NoSQL architectures.

This paper presents a comprehensive review of migration strategies and methodologies for transitioning from relational databases to document-oriented NoSQL systems. It examines key components of the migration process, including schema conversion, data transformation, and query refactoring, while evaluating existing tools and techniques in terms of automation, scalability, and data integrity. The study identifies unresolved research challenges, including semantic preservation, schema evolution, and tool interoperability, and proposes future directions, such as AI-assisted migration planning. Overall, this paper highlights key research gaps to help researchers support the development of reliable, scalable, and semantically robust migration solutions.

General Terms
Document-oriented NoSQL Databases, SQL to NoSQL Migration, Data Transformation

Keywords
Migration Strategies, Database Schema Conversion, Data Transformation, Query Refactoring.

1. INTRODUCTION
Relational databases have dominated the DBMS market for decades. Their high level of abstraction and strong data independence are their most powerful characteristics. Relational databases implement well-defined transaction management to guarantee ACID (Atomicity, Consistency, Isolation, Durability) properties, making them reliable and secure for ensuring data integrity and consistency. As a result, they have long been considered the most trusted option for managing structured data. However, the limitations of relational databases have become more apparent due to massive changes in application requirements that rapidly raise the size and diversity of data. These emerging requirements are often difficult to meet using traditional relational models.

To address these emerging challenges, NoSQL databases have gained significant acceptance by offering more flexible and scalable data models. Among the various NoSQL types, document-oriented databases stand out due to their ability to store semi-structured data in JSON-like formats, which meet modern application needs. They provide high scalability, schema flexibility, and higher performance for handling large volumes of diverse and rapidly changing data. Additionally, many document databases, such as MongoDB, are available as free and open-source software, making them accessible to a wide range of users. Consequently, document databases have become popular alternatives to traditional relational systems, particularly in scenarios involving big data, real-time analytics, and cloud-native applications.

Given these advantages, many organizations with existing relational database systems are now seeking to migrate to document-oriented NoSQL databases. This migration is driven by the need to support evolving data structures, improve scalability, and reduce the complexity imposed by rigid relational schemas. By transitioning to document databases, businesses can more effectively accommodate unstructured or semi-structured data and enhance application performance. However, this transition is not trivial. It requires a deep understanding of both data models, careful planning for schema transformation, and the use of reliable tools to automate and validate the migration process. This underscores the importance of studying available migration methodologies specifically tailored for moving existing systems from relational to document-oriented database systems. Improper migration can result in data inconsistency, performance degradation, or system downtime.

This paper aims to review the existing tools, algorithms, and techniques used for migrating relational databases to document-oriented NoSQL databases, with a particular focus on MongoDB. By analyzing various approaches, this study seeks to identify the strengths and limitations of current migration strategies, highlight best practices, and uncover gaps that require further research. The goal is to provide a complete understanding of the migration landscape to assist practitioners in selecting or developing effective solutions for seamless data transition from relational to document-oriented database systems.

2. LITERATURE REVIEW
The migration from relational databases to document-oriented NoSQL systems has received increasing attention due to the limitations of traditional RDBMS in handling modern data demands. Several studies have explored schema transformation techniques, emphasizing the need for flexible mapping strategies such as direct, nested, and hybrid approaches [1][3][6]. Tools like DLoader and SQL2Mongo have been proposed to automate schema conversion and data extraction, offering varying levels of support for query translation and workflow orchestration [10][11].

Semantic preservation during migration remains a critical challenge, especially in domains requiring high data fidelity such as healthcare and finance [5]. Researchers have also

highlighted the importance of data transformation algorithms, including normalization, cleansing, and format conversion, to ensure consistency and performance in NoSQL environments [4][7]. Query refactoring has been addressed through techniques that eliminate joins, restructure aggregations, and translate predicates into NoSQL-compatible formats [9]. Middleware tools are increasingly used to connect SQL and NoSQL systems during migration, allowing for hybrid setups and minimizing operational interruptions [10].

Recent work has proposed unified metamodels and AI-assisted migration planning to improve automation and semantic coherence [3][6]. Additionally, snapshot–live stream migration techniques have been introduced to minimize downtime and enhance scalability during data transfer [4][12]. Despite improvements, there are still problems with testing tools, dealing with complex queries, and keeping up with changes in database schema. Continued research is needed to develop standardized frameworks and robust evaluation metrics for seamless and reliable migration [5].

A variety of tools have emerged to facilitate the migration process. DLoader, Talend, Apache NiFi, SQL2Mongo, and custom ETL scripts represent distinct approaches to handling schema conversion, data movement, and query adaptation. DLoader emphasizes schema mapping and data extraction, offering moderate automation and support for hybrid mapping strategies. Talend and Apache NiFi excel in workflow orchestration, enabling rule-based execution, error handling, and retry mechanisms that enhance reliability and scalability [11]. SQL2Mongo is tailored for MongoDB migrations, with strong capabilities in query translation and schema transformation. Custom ETL scripts, while highly flexible, require substantial manual effort and domain expertise, making them suitable for specialized use cases [11]. These tools vary in their approach to schema conversion. Some favor direct mapping for simplicity, while others support nested or hybrid structures to optimize performance. Automation levels also differ, with some tools offering graphical interfaces and others relying on scripting and manual configuration.

Drawing from the literature review, Section 3 presents an overview of various algorithmic strategies for key migration domains, highlighting the technical mechanisms that enable data and schema transformation.

3. KEY MIGRATION DOMAINS AND ALGORITHMIC STRATEGIES

Migration from relational databases (RDBMS) to document-oriented NoSQL systems involves a variety of algorithmic challenges. We categorize these challenges under three domains - database schema, data and query statement. Each area involves distinct techniques and tools to ensure a seamless and reliable transition. This section examines these domains in depth, based on a comprehensive review of literature.

The migration from relational to document-oriented databases includes several steps related to schema, data, and query changes. Figure 1 shows the overall migration framework with the main strategies and how they are connected.

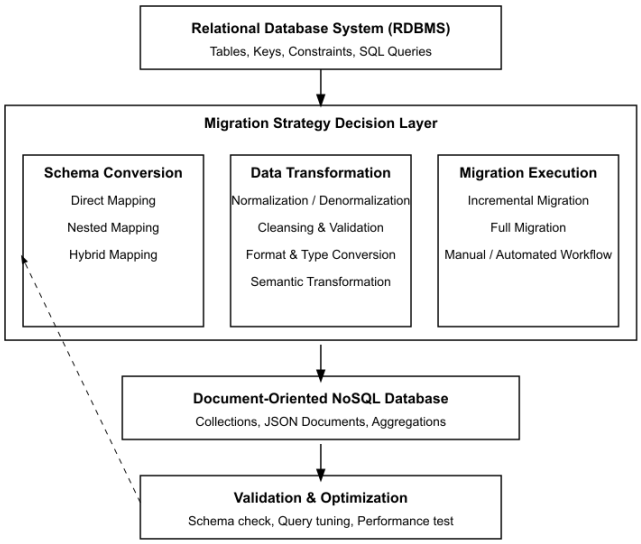

Figure 1: Framework for relational-to-document database migration

3.1 Schema Conversion Strategies

The process of converting a relational database schema into a document-oriented NoSQL schema is a foundational step in migration. This transformation requires algorithmic strategies that reinterpret the rigid, tabular structure of relational databases into the flexible, hierarchical format of document stores. Schema conversion algorithms aim to preserve data integrity, optimize query performance [1][5][6], and reflect real-world relationships in a more intuitive format.Based on the literature, these algorithms can be broadly categorized into three main approaches.

3.1.1 Direct Table-to-Document Mapping

Direct mapping approaches perform a straightforward translation where each relational table is mapped to a corresponding document collection. Each row within the table is transformed into a separate document. Foreign key relationships are typically handled by referencing related documents using unique identifiers.

3.1.2 Nested Document Mapping

Nested mapping algorithms are designed to exploit the hierarchical nature of document databases by embedding related data directly within parent documents. One-to-many and many-to-many relationships are represented using nested arrays or embedded sub-documents.

3.1.3 *Hybrid Embedding and Referencing Technique*

Hybrid approaches combine embedding and referencing strategies based on query frequency, data volatility, and access patterns. Frequently accessed relationships are embedded for performance, while less critical or highly dynamic associations are referenced for modularity. This framework clarifies schema conversion techniques and the algorithmic decisions shaping NoSQL schema efficiency. Each method has strengths and limitations, with the choice depending on application requirements and workload. Table 1 summarizes key technical aspects of these strategies.

Table 1: Schema Conversion Strategies

	Direct Table-to-Document	Nested Document Mapping	Hybrid Embedding & Referencing
Strategy	Each table becomes a document; rows become individual documents. Foreign keys are referenced.	Related data embedded as sub-documents or arrays within parent documents.	Mix of embedded and referenced data based on access patterns; guided by heuristics or ML models.
Pros	Simple to implement; maintains clear traceability to the original schema.	Reduces need for joins; improves read performance for queries that retrieve related data together.	Balanced performance and modularity; adaptable to diverse access patterns.
Cons	Fragmented data access patterns; multiple lookups for related data.	Increases data redundancy that may result in inconsistency; also increase document size	Complex to implement; requires workload analysis and dynamic mapping logic.
Best Use Case	Small systems with well-defined, isolated relationships.	Read-heavy applications with frequent access to related entities.	Complex applications with mixed access patterns and evolving schemas.

Table 2: Data Transformation Techniques

	Normalization & Denormalization	Data Cleansing & Validation	Format & Type Conversion	Semantic Transformation
Strategy	Reorganize data to reduce redundancy (normalize) or embed for performance (denormalize).	Identify and correct inconsistencies, errors, or duplicates in source data.	Convert data types and formats to meet NoSQL constraints.	Adjust data meaning based on domain rules or business semantics.
Pros	Enables consistency or improves read performance depending on strategy.	Ensures high data quality and integrity.	Ensures compatibility and preserves query ability.	Preserves business logic and improves semantic clarity.
Cons	Normalization complicates embedding; denormalization can introduce redundancy.	May be time-consuming; can require domain-specific rules.	Potential for data loss if mapping is incorrect.	Requires deep domain knowledge; prone to misinterpretation.
Best Use Case	Systems prioritizing consistency (normalize) or performance (denormalize).	Legacy systems with poor data hygiene.	Applications requiring strict data format alignment.	Domains with rich semantics (e.g., healthcare, finance).
Real-World Example	E-commerce platform normalizes customer and order data for inventory consistency, while denormalizing product reviews for faster reads.	Healthcare migration cleans patient records by removing duplicates and correcting inconsistent date formats before EMR transfer.	Financial services convert SQL timestamps to ISO 8601 and normalize currency formats for NoSQL analytics compatibility.	Insurance claims map raw codes to standardized categories and merge address fields for improved reporting clarity.

3.2 Data Transformation Techniques

Data transformation is a critical phase in the migration from relational databases to document-oriented NoSQL systems. While schema conversion addresses structural compatibility, data transformation ensures that the actual content conforms to the semantics, formats, and constraints of the target system. This process involves cleansing, reformatting, and restructuring data to line up with the flexible and often denormalized nature of document databases. The literature [4][11] identifies several strategies that facilitate this transition.

Data transformation strategies can be categorized based on their role in adapting relational data for use in document-oriented NoSQL systems. These categories reflect algorithmic approaches to data normalization, quality assurance, format alignment, and semantic restructuring.

3.2.1 Normalization and Denormalization Strategies

Normalization is applied during data extraction to identify and separate redundant or repetitive data to ensure consistency before transformation. Conversely, Denormalization is often applied during migration to embed related data within documents to reduce the need for joins and improve read performance in NoSQL systems. These algorithms typically analyze dependency graphs and relational keys to determine optimal strategies for grouping and flattening data.

3.2.2 Data Cleansing and Validation Algorithms

Data cleansing and validation are critical steps to ensure data integrity and prevent the propagation of errors into NoSQL systems, where the lack of strict schemas makes anomalies harder to detect once data is migrated. These processes detect and correct common problems such as missing values, inconsistent formats, and duplicate records. Some common techniques for handling validation include rule-based validation, statistical methods for outlier detection, and pattern matching using regular expressions.

3.2.3 Format and Type Conversion

Format and type conversion algorithms adapt data structures and types to meet the requirements of the target NoSQL database. Examples include converting SQL DATETIME values to ISO 8601 strings, adjusting numeric precision, and standardizing character encodings (e.g., UTF-8). These conversions are typically guided by transformation rules or mapping tables based on the constraints of the target database's schema.

3.2.4 Semantic Transformation Based on Domain Logic

Semantic transformations go beyond structural changes to reinterpret data based on domain-specific rules or business logic. For example, categorical values might be mapped to standardized labels, or multiple fields combined into a single composite attribute. These transformations are often guided by domain models to maintain meaning, consistency, and relevance in the target system.

Together, these algorithmic strategies ensure that the migrated data is not only structurally compatible but also semantically coherent and optimized for performance in a document-oriented environment. The choice and combination of transformation techniques depend heavily on the nature of the source data, the design of the target schema, and the operational requirements of the application. Table 2 describes these techniques at various dimensions.

3.3 Query Refactoring and Optimization

Query refactoring is an essential process in the migration from relational to document-oriented databases, as both systems use different query languages. While relational databases rely heavily on Structured Query Language (SQL), with support for joins, aggregations, and transactions, NoSQL databases work with hierarchical data and use imperative query frameworks optimized for document structures. Along with translating queries into the target paradigm, refactoring algorithms must focus on optimizing them to maintain correctness, efficiency, and semantic accuracy. The literature [2][9] outlines several algorithmic strategies for this purpose.

3.3.1 Join Elimination via Embedded Structures

These algorithms analyze SQL queries to identify join operations and determine whether they can be eliminated through schema denormalization. When relationships are embedded within documents—as established during schema conversion—joins can be replaced with direct access to nested fields. NoSQL databases often replace costly join operations by embedding related data directly within documents. However, too much embedding can cause data duplication and make updates harder. Theoretical methods use cost-based optimization and graph algorithms to analyze query patterns and data relationships. This helps decide the best balance between embedding and referencing to speed up queries while controlling redundancy and update costs.

3.3.2 Refactoring Aggregation Logic

SQL queries frequently utilize GROUP BY, HAVING, and aggregate functions such as SUM, COUNT, and AVG. Refactoring algorithms translate these constructs into the aggregation frameworks provided by NoSQL systems, such as MongoDB's aggregation pipeline. These algorithms restructure query logic into stages like filtering, grouping, and projecting, while preserving the original semantic intent.

3.3.3 Predicate and Filter Translation

Predicate and filter translation algorithms convert SQL WHERE clauses and conditional expressions into equivalent NoSQL query filters. They handle logical operators (AND, OR, NOT), comparison operators, and pattern matching constructs. Special attention is given to indexing strategies and query optimization in the target system to maintain performance and avoid inefficiencies.

3.3.4 Migration of Stored Procedures and Functions

Business logic embedded in stored procedures or user-defined functions must be re-implemented using NoSQL-compatible scripting or application-layer logic. Refactoring algorithms decompose procedural code into modular operations that can be executed within the constraints of the NoSQL query engine. In some cases, external orchestration tools or middleware are employed to replicate procedural behavior effectively.

3.3.5 Query Optimization in NoSQL Environments

In addition to translation, query optimization techniques examine execution strategies to detect inefficiencies and recommend enhancements for better performance. Techniques include index utilization, query rewriting, and caching strategies tailored to the NoSQL environment [9]. Optimization is especially critical in systems that lack traditional query planners or cost-based optimizers.

These query refactoring and optimization techniques work together to bridge the gap between relational and document-based queries. Query refactoring ensures that the migrated application continues to function correctly and efficiently, despite the fundamental differences in query paradigms between relational and document-oriented systems. By removing joins, reshaping aggregations, translating filters, moving procedural code, and tuning for NoSQL, they help keep queries accurate and fast after migration. Success depends on an understanding of source and target query models, as well as the ability to adapt logic to new data access patterns.

4. MIGRATION METHODOLOGY

Building on the foundational components of schema conversion, data transformation, and query refactoring, this section explores the overarching methodology that governs the migration process from relational databases to document-oriented NoSQL systems. Migration methodology refers to the structured approach used to plan, execute, validate, and

optimize the transition, ensuring minimal disruption, data integrity, and long-term scalability.

4.1 Strategic Planning and Dependency Mapping

Effective migration starts with a thorough evaluation of the source system, taking into account schema complexity, data volume, and the relationships between tables. Dependency analysis plays a critical role in determining the optimal sequence for migrating tables, views, and procedures. Graph-based models and specialized migration tools are often employed to preserve referential integrity and accurately map dependencies during this process. Planning also involves identifying business-critical components, estimating acceptable downtime windows, and assessing system constraints to reduce risks and ensure a smooth migration flow [6][8][9].

4.2 Choosing the Right Migration Strategy: Incremental vs. Full

Migration strategies typically fall into two categories - incremental and full migration [11].

Incremental Migration involves transferring data and functionality in stages, allowing the system to remain operational throughout the process. This approach is ideal for large or complex systems where minimizing downtime is essential. It requires synchronization between the source and target databases to maintain consistency and ensure data integrity at each step.

Full Migration refers to moving all data and operations in a single, comprehensive step. This strategy is suitable for smaller systems or environments where temporary downtime is acceptable. Full migration demands thorough preparation, including complete backups, pre-migration testing, and a reliable rollback plan to mitigate risks in case of failure.

4.3 Ensuring Data Integrity: Validation and Quality Checks

Post-migration validation is essential to confirm that data has been accurately and completely transferred. Validation techniques include record-level comparisons, schema conformity checks, and referential integrity tests. Quality assurance also involves verifying semantic consistency, particularly when data has been transformed or denormalized during migration. These validations help to confirm that the transferred data preserves its original context and remains functional for its intended use. [6][11].

4.4 Automating the Migration Pipeline

Automation tools play a vital role in streamlining the migration process by coordinating tasks such as schema conversion, data transformation, and query adaptation. Workflow orchestration platforms like Apache NiFi and Talend enable rule-based execution, error handling, and retry mechanisms, reducing manual effort and minimizing human errors. Automation speeds up migration and supports scalability and repeatability across different projects.

4.5 Preparing for Failure: Rollback and Recovery Plans

A robust migration methodology includes contingency plans for handling failure scenarios. Rollback mechanisms may involve checkpointing, transactional wrapping, or maintaining parallel systems during migration. These strategies ensure that data integrity is preserved and that the system can revert to a stable state if needed. Effective recovery planning minimizes risk and helps maintain operational continuity, safeguarding against unexpected disruptions.

4.6 Optimizing Performance and Resource Utilization

Migration performance is influenced by factors such as data volume, system load, and network bandwidth. Optimization techniques such as parallel processing, load balancing, and dynamic resource allocation is used to enhance throughput and reduce latency. Monitoring tools are employed to track migration progress and identify bottlenecks in real time, enabling proactive adjustments and ensuring efficient resource utilization.

Migration methodology is not merely a technical exercise—it is a strategic framework integrating planning, execution, validation, and optimization. A well-defined methodology ensures that the transition from relational to document-oriented systems is not only technically sound but also aligned with business goals, operational constraints, and long-term scalability. By combining structured processes with adaptive strategies, organizations can achieve seamless and resilient migration outcomes.

5. RESEARCH GAPS AND FUTURE DIRECTIONS

Despite significant advances in schema conversion, data transformation, and query refactoring, several challenges remain in achieving reliable and semantically consistent relational-to-NoSQL migration. The study of existing literature reveals several underexplored areas that require further research. The following points highlight key research gaps along with corresponding future directions.

5.1 Lack of Standardized Migration Frameworks

Existing migration workflows depend on diverse tools such as Talend, Apache NiFi, SQL2Mongo, and DLoader, each following its own mapping and transformation logic. While Talend supports ETL-based schema integration [13] and NiFi provides flow-oriented data orchestration [14], tools like SQL2Mongo [3] and DLoader [11] address only specific aspects of relational-to-NoSQL migration. The lack of a unified framework or standardized ontology leads to inconsistent schema interpretation, limited interoperability, and redundant development efforts. Establishing a common migration reference model remains a key research priority [3].

Future work should focus on developing a unified migration metamodel and standardized ontology to harmonize relational and document-oriented schemas, enabling consistent schema interpretation and bidirectional synchronization [6].

5.2 Schema Evolution and Version Control

Document-oriented databases are inherently schema-flexible, yet this adaptability complicates versioning and maintenance. Maintaining backward compatibility as document structures evolve often introduces semantic drift and operational inconsistency [5][12].

Research is needed to implement version-aware migration mechanisms that automatically track, reconcile, and validate schema evolution across releases, thereby reducing semantic drift and ensuring long-term maintainability.

5.3 Semantic Preservation and Business Logic Alignment

Existing migration algorithms effectively handle structural and query transformations but often fail to retain domain semantics and embedded business logic [9][12]. Critical contextual information may be lost when stored procedures, triggers, and constraints are externalized into application code. The lack of formal semantic mapping or ontology-driven reasoning remains a significant theoretical gap.

A comprehensive effort is needed to provide ontology-driven, semantic-preserving frameworks to retain business logic and domain meaning, ensuring migrated data remains contextually accurate.

5.4 Complex Query and Transactional Migration

Most current refactoring tools manage only simple select–project–join queries and struggle with procedural constructs, nested subqueries, or multi-transaction workflows [9][10].

Achieving full functional parity between SQL and NoSQL query paradigms requires extending current translation layers or adopting middleware abstractions capable of reinterpreting complex transactional semantics.

5.5 Lack of Automation, Tool Interoperability, and Evaluation Metrics

Current migration processes often require manual intervention and rely on heterogeneous tools, which limits automation and reproducibility. Differences in tool-specific mapping and transformation logic reduce interoperability across systems [3][6]. Additionally, standardized metrics for evaluating migration quality, performance, and semantic accuracy are lacking.

Future frameworks should automate schema mapping, validation, and rollback, integrate with CI/CD pipelines for reproducibility, and include standardized benchmarks to measure performance, semantic fidelity, and maintainability.

5.6 Underutilization of AI and Machine Learning

While AI and machine learning techniques have been applied in areas like query optimization and system tuning, their use in relational-to-NoSQL migration remains limited. Current tools rarely leverage AI for schema mapping, dependency inference, or cost prediction [3][6].

Integrating AI and machine learning into migration workflows can automate decision-making, optimize schema transformations, predict performance, and recommend the most efficient mapping strategies.

In addition to the above, future research should focus on hybrid middleware solutions that support gradual migration and coexistence of SQL and NoSQL systems. Middleware can enable dynamic query routing, live synchronization, and transaction consistency for near-zero downtime migrations [10]. Establishing standard evaluation metrics is also essential: open, community-driven benchmarks covering performance, scalability, semantic accuracy, and maintainability will allow fair comparison of migration approaches [1][6]. Finally, integrating knowledge from data governance, software engineering, and knowledge representation can embed compliance rules and domain constraints directly into migration workflows, ensuring both technical and organizational alignment [6][12]. Addressing these areas will support the development of intelligent, automated, and semantically aware migration frameworks, capable of delivering reliable, scalable, and contextually accurate transformations.

6. Conclusion

The migration from relational to document-oriented databases presents a multifaceted challenge encompassing schema design, data transformation, query adaptation, and system orchestration. While current tools and techniques offer a strong foundation for facilitating this transition, several theoretical and practical issues remain unresolved. These gaps highlight the need for continued research and innovation to support the dynamic requirements of modern data systems.

7. REFERENCES

[1] Alotaibi, O., & Pardede, E. (2019). Transformation of schema from relational database (RDB) to NoSQL databases. Data, 4(4), 148. https://doi.org/10.3390/data4040148

[2] Amir Guliyev, & Kazazi, F. (2023). Migration of data from RDBMS to NoSQL and possibility of implementing a single query language for NoSQL databases. *International Journal for Research in Applied Science and Engineering Technology, 11*(X), 2004–2008. https://www.ijraset.com

[3] Bansal, N., Soni, K., & Sachdeva, S. (2022). Journey of database migration from RDBMS to NoSQL data stores. In S. Sachdeva et al. (Eds.), *BDA 2021* (Lecture Notes in Computer Science, Vol. 13167, pp. 1–19). Springer Nature Switzerland AG. https://doi.org/10.1007/978-3-030-96600-3_12

[4] Bhagwat, G., Rajiv, Y. A., & Kawale, P. (2022). Data migration from SQL to NoSQL using snapshot–live stream migration. International Journal of Scientific Research in Science and Technology, 9(1).

[5] Chillón, A. H., Klettke, M., Ruiz, D. S., & Molina, J. G. (2022). A taxonomy of schema changes for NoSQL databases. *arXiv Preprint*, arXiv:2205.11660. https://arxiv.org/abs/2205.11660

[6] Fernández Candel, C. J., Sevilla Ruiz, D., & García-Molina, J. J. (2022). A unified metamodel for NoSQL and relational databases. *Information Systems, 104*, Article 101898. https://doi.org/10.1016/j.is.2021.101898

[7] Ghotiya, S., Mandal, J., & Kandasamy, S. (2017). Migration from relational to NoSQL database. *IOP Conference Series: Materials Science and Engineering, 263*, 042055. https://doi.org/10.1088/1757-899X/263/4/042055

[8] Kuderu, N., & Kumari, V. (2016). Relational database to NoSQL conversion by schema migration and mapping. *International Journal of Computer Engineering Research Trends, 3*(9), 506–513.

[9] Modi, N., & Patel, J. (2025). Evaluating query processing architectures in relational and document databases. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 12*(2), 2732–2736.

[10] Namdeo, B., & Suman, U. (2022). A middleware model for SQL to NoSQL query translation. *Indian Journal of Science and Technology, 15*(16), 718–728. https://doi.org/10.17485/IJST/v15i16.2250

[11] Rajaram, K., Sharma, P., & Selvakuma, S. (2023). DLoader: Migration of data from SQL to NoSQL databases. In *Proceedings of the International Conference on Cognitive and Intelligent Computing (ICCIC 2021)* (Vol. 2). Springer. https://doi.org/10.1007/978-981-19-2358-6_19

[12] Saur, K., Dumitras, T., & Hicks, M. (2015). Evolving NoSQL databases without downtime. *arXiv Preprint*, arXiv:1506.08800. https://doi.org/10.48550/arXiv.1506.08800

[13] Minu Balakrishnan, Abinayaa Azhahappan, Ashwin Ramdass, Rithika Sathiyappan; Integration and migration of data in ETL using Talend. AIP Conf. Proc. 4 March 2024; 3035 (1): 020009. https://doi.org/10.1063/5.0195142

[14] Wnęk, K., & Boryło, P. (2023). A Data Processing and Distribution System Based on Apache NiFi. Photonics, 10(2), 210. https://doi.org/10.3390/photonics10020210