International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 11 | Nov - 2025 SJIF Rating: 8.586 ISSN: 2582-3930

A Comparison of Developer Performance Across Raw SQL, ORM-Inspired Interfaces, and a Cosmos-Specific ODM

Sanjyoti Kumar Tarai[1], Rishabh Sharma[2*], Priyanshu Sharma[2], Ayush Godiyal[2], Lakshay Kumar Signh[2]

[1.] Assistant Professor,
Dr. Akhilesh Das Gupta Institute of Technology and Management, New Delhi, India
sanju1463@gmail.com

[2.] Department of Information technology,
Dr. Akhilesh Das Gupta Institute of Technology and Management, New Delhi, India
[*]rishabhsharma17aug2004@gmail.com

Abstract

Anyone who's worked with Azure Cosmos DB knows the frustration. The platform has its own SQL dialect that doesn't quite match what you're used to. You need to constantly think about partitions when writing queries. And there's no schema enforcement which sounds great until you realize how easy it is to accidentally break things. sure, we have tools like Prisma and Mongoose that make database work easier. They're genuinely great at what they do. But here's the problem: they were designed for completely different databases. Prisma targets relational databases, Mongoose was built for MongoDB. Neither one really gets what makes Cosmos DB different, so developers are left figuring things out on their own. We wanted to see if we could do better. This paper looks at four different ways to work with Cosmos DB: writing raw SQL queries, using Prisma-style interfaces, following Mongoose's patterns, and using a new ODM we built specifically for Cosmos. Our version has Zod validation for catching schema issues, typed field references so you don't misspell things, automatic parameterization to prevent injection attacks, and a visual Explorer that actually understands Cosmos DB's structure. We got several developers together and had them complete the same tasks using each method. We measured everything how accurate they were, how long tasks took, how much mental energy they burned, and how many errors they made. The difference was pretty clear. With the Cosmos-specific ODM, developers made far fewer mistakes. No more typos in field names, no more schema mismatches, way fewer syntax errors. Tasks got done faster, and the queries people wrote were more reliable and predictable. What this really tells us is that one-size-fits-all tools don't work for specialized platforms. Cosmos DB has its own personality, its own quirks. It needs tools that understand those quirks. When developers have the right tools for the job, everything gets easier they write cleaner code, make fewer mistakes, and ship more reliable features.

1. Introduction

Azure Cosmos DB has become a go-to choice for cloud-native applications, and it's easy to see why. Global distribution, elastic scaling, and a flexible document model make it perfect for modern distributed systems. But these same features that make Cosmos DB powerful also create real headaches for developers. You're dealing with a custom SQL dialect, you need to constantly think about partition keys when accessing data, and there's no enforced schema to catch your mistakes before they become problems.

Research has shown that this complexity takes a toll. Studies demonstrate that query syntax challenges and missing structural guidance significantly increase both developer errors and mental strain[1,3]. In practice, this means developers write incorrect filters, mess up query conditions, forget to parameterize their queries, and end up with inconsistent document structures across their database.

Tools like Prisma and Mongoose try to help by offering fluent query interfaces and schema-driven abstractions that make database work more intuitive. These are genuinely useful tools. The catch is they were built with completely different databases in mind Prisma for relational databases, Mongoose for MongoDB. Neither understands what makes Cosmos DB unique. Research on NoSQL modeling has documented how schema drift and unvalidated writes lead to bugs and data inconsistencies in document databases[5],[11]. So most developers end up writing raw Cosmos SQL instead, which gives them complete control but also exposes them to more errors and a steeper learning curve.

This paper introduces a Cosmos-specific Object Data Mapper (ODM) designed from the ground up for the platform. It includes Zod-based schema validation to catch issues early, typed field references that prevent typos, automatic SQL parameterization for security, and an integrated Explorer that lets you inspect your data in real time. We tested this ODM against three other approaches: raw Cosmos SQL, Prisma-style fluent APIs, and Mongoose-style ODM patterns. Developers completed the same query construction and data manipulation tasks using each method.

Our goal was straightforward: figure out whether building tools specifically for Cosmos DB actually makes a difference. Can domain-specific abstractions improve productivity, reduce mistakes, and help developers build more reliable cloud-based applications? That's what we set out to answer.

2. Background & Related Work

Research on how developers interact with database systems touches on everything from query usability to schema management and tooling for semi-structured data. Usability studies have repeatedly found that unfamiliar or overly rigid query languages increase both cognitive load and error rates among developers[1]. Real world software projects show this same pattern developers frequently wrestle with the mismatch between their application code and database access layers[3].

Cosmos DB sits in a unique position within the database landscape. It pairs a flexible JSON document model with SQL based querying, all running on a partition-oriented execution model. This combination sets it apart from traditional relational systems and schema less NoSQL databases alike. Work on schema evolution in NoSQL systems has identified a recurring problem: overly flexible document structures lead to schema drift and inconsistencies that become harder to manage over time[5]. Broader studies of NoSQL modeling practices echo these concerns, documenting how insufficient structure creates bugs and data anomalies that surface later in development[12]. Various researchers have tried tackling these challenges using schema extraction techniques and taxonomy based classification systems for tracking schema changes[6],[9].

A different research thread examines ways to make query construction more intuitive. When researchers compared visual and form based query builders, they found that guided interfaces help developers avoid syntax errors and construct correct queries more reliably[1]. Natural language interfaces for databases offer similar advantages, reducing the need to master complex query syntax[4]. The underlying principle here is straightforward: when developers have structured, guided tools, they're simply more productive.

Unified or cross database query frameworks represent another major research area. These frameworks use unified query plans or middleware layers to provide consistent interfaces across different data models. However, they frequently miss the specific behaviors and constraints of individual platforms[8],[10]. This limitation becomes especially apparent with Cosmos DB. The platform's consistency models, partitioning constraints, and request-unit cost considerations create challenges that generic ORMs weren't built to handle. Prisma style relational builders and Mongoose style document mappers, while effective for their intended databases, don't map well onto Cosmos DB's operational semantics.

Looking across this research landscape, three patterns stand out. Query languages that feel unfamiliar or complex slow developers down and increase errors. Schema flexibility in NoSQL systems creates technical debt that accumulates over time. Generic query abstractions, regardless of how well-designed, struggle to accommodate platform-specific requirements. These gaps suggest an obvious need: Cosmos DB developers would benefit from purpose built tooling that offers type safety, schema validation, and query guidance matched to the platform's actual behavior and constraints.

3. Problem Statement

Developers working with Azure Cosmos DB face challenges that stem from how the platform is built its custom SQL dialect, distributed storage model, and the fact that there's no enforced schema. When you write raw Cosmos SQL, you get complete control over your queries. But that flexibility comes at a cost. You're responsible for managing syntax details, parameterization, field accuracy, and partition-aware query patterns all by yourself. Research has shown that this kind of unassisted query writing increases both syntax errors and the mental effort required[1]. Things become even trickier when you're dealing with semi-structured JSON documents. Inconsistent fields and schema drift introduce subtle bugs that can easily go unnoticed during development[5].

It makes sense that developers turn to general-purpose abstractions like Prisma-style fluent APIs and Mongoose-style ODMs. These tools genuinely simplify database interactions by providing higher-level interfaces that reduce boilerplate. But here's where things break down: neither Prisma nor Mongoose was designed with Cosmos DB's specific characteristics in mind. Partition keys, continuation tokens, RU consumption, Cosmos-specific query patterns these tools simply don't account for them. Research on NoSQL modeling practices backs this up, showing that generic abstractions often miss the nuances of individual platforms, which leads to queries that either behave unreliably or run inefficiently[12]. So while you do get less boilerplate code, you're still vulnerable to structural errors, weak type checking, and data inconsistencies that happen silently in the background.

What developers really need is tooling built specifically for Cosmos DB something that provides strong type safety, validates schemas properly, guides you through safe query construction, and includes integrated data inspection tailored to how Cosmos actually works. Without these capabilities, the typical development experience involves more errors, longer task completion times, and significant frustration when trying to diagnose data problems. This situation clearly calls for a domain-specific ODM and query builder that actually works with Cosmos DB's operational model instead of fighting against it.

4. Methodology

4.1 Study Design

A comparative, task-based study was chosen because it allows direct measurement of how developers perform when using each approach. Every participant completed the same set of tasks four time’s once using each method. The order of methods was randomized to reduce learning bias.

All experiments were executed using the Azure Cosmos DB Emulator to eliminate network inconsistencies and RU cost variability. Participants interacted with the provided APIs (raw SQL, Prisma-style mock, Mongoose-style mock, and the proposed ODM) through a standardized TypeScript environment with identical scaffolding, code templates, and debugging tools.

4.2 Participants

The study recruited developers with varying experience levels, including:

- beginners with limited database experience,
- intermediate developers familiar with SQL or Mongo-style queries, and
- experienced backend developers.

A mixed experience group was intentionally chosen to reflect real-world teams where skill levels vary. Each participant was given a short introduction to the dataset and API signatures but not to the query logic itself, ensuring the tasks evaluated genuine usability rather than memorized patterns.

4.3 Tasks

Participants completed five representative tasks that reflect common operations in real Cosmos DB applications:

1. Filtering Task
Write a query to retrieve documents where a numeric field satisfies a condition (e.g., age > 25).

2. Pagination Task
Retrieve a subset of results using TOP and continuation tokens.

3. Cross-Partition Task
Execute a query that requires searching multiple partitions safely.

4. Insert Task
Add a new document with correct structure, ensuring all required fields are present.

5. Update Task
Update an existing document in a type-safe manner without overwriting unintended fields.

Each task has a well-defined correct answer, allowing objective evaluation of correctness.

4.4 Measurements

The following metrics were recorded for each participant and for each approach:

A. Productivity Metrics

- Task Completion Time: measured from the moment the participant begins typing to successful execution.
- Number of Attempts: how many times the participant had to retry due to errors.
- Developer Steps: the count of meaningful actions required to complete a task (e.g., writing conditions, adjusting syntax).

B. Error Metrics

Errors were categorized into:

- Syntax errors (incorrect SQL format, invalid keywords)
- Field mismatch errors (incorrect or missing field names)
- Schema violations (malformed JSON or missing required fields)
- Unparameterized queries (unsafe string interpolation)
- Partition-related mistakes (missing or incorrect partition keys)

Each error was logged automatically through console output or Cosmos emulator error messages.

C. Cognitive Load Metrics

After each method, participants rated:

- task difficulty,
- confidence in correctness, and
- perceived clarity of the API.

Ratings were collected using a 5-point Likert scale.

4.5 Procedure

The experiment followed a consistent sequence:

1. Participants received a short briefing.

2. They completed all five tasks using one approach.
3. They filled out the cognitive load survey.
4. The process was repeated for the remaining three approaches in randomized order.
5. All collected logs, execution times, and error counts were analyzed.

This structured procedure ensures comparability and reduces bias due to learning effects or ordering.

5. Implementation

The system we built has two main components that work closely together: a Cosmos-native Object Data Mapper (ODM) focused on type safety and schema enforcement, and a fluent Query Builder that generates parameterized Cosmos SQL while steering developers away from common pitfalls. Our goal was to eliminate error-prone manual work and create a structured, Cosmos-aware programming interface that actually understands how the platform operates.

5.1 Schema Definition and Validation

We chose Zod schemas as the foundation for defining document structure in the ODM. This choice gives us runtime validation while simultaneously producing TypeScript types that provide compile-time safety. Using runtime schema validation tackles the well-documented problems around schema drift and inconsistent document structures that plague NoSQL systems[5],[9].

Model definitions use a configuration object that includes the Cosmos container name, partition key path, Zod schema for validation, and optional indexing or metadata settings. The schema acts as the single authoritative source for both validation and type inference. Once you create a model, every insert, update, and bulk write operation goes through validation before touching the database. This prevents malformed or incomplete documents from ever getting persisted.

5.2 Typed Field References

Each model exposes a .fields object that transforms the schema into strongly typed field references. This gives developers two important advantages. First, it eliminates magic strings—you can't mistype field names, which is a surprisingly common source of runtime errors. Second, it dramatically improves autocomplete in IDEs, which can now suggest valid fields automatically. This reduces both search time and mental effort during query construction.

The design takes inspiration from type-safe query abstractions in earlier systems[4], but we adapted it to match Cosmos DB's JSON-document structure and SQL-style query execution.

5.3 Fluent Query Builder

The Query Builder offers a chainable API for building Cosmos SQL queries without writing SQL strings manually. Developers work with expressive functions like eq(field, value), gt(field, value), and(expr1, expr2), ilike(field, pattern), and orderBy(field, direction). Behind the scenes, each function builds an abstract syntax representation of the query, which gets compiled into a parameterized Cosmos SQL string later.

This approach guarantees automatic parameter binding, consistent aliasing, injection protection, and error checking before execution. The Query Builder also understands partition keys. When queries would trigger cross-partition scans, the builder either warns the developer or applies settings explicitly. This prevents developers from accidentally racking up unnecessary RU consumption a frequent operational concern with Cosmos DB[13].

5.4 CRUD Operations and Query Execution

The ODM provides standard high-level methods: insert, insertMany, find, findById, update, and deleteById. Each method integrates validation and either the Query Builder or parameterized SQL internally. The result is a consistent

API surface that avoids the inconsistencies you typically see in generic ODMs that weren't designed for Cosmos-specific patterns.

Execution happens through the official Cosmos SDK, and responses include typed outputs matching the schema. When continuation tokens appear in responses, the ODM wraps them in a structured pagination object that makes multi-page query handling straightforward.

5.5 Integrated Explorer Interface

We included a graphical Explorer interface to support debugging and real-time data inspection. Unlike general-purpose tools that treat all databases the same way, our Explorer was built specifically around Cosmos DB's hierarchical structure. It shows databases and containers in a navigable tree, includes a Query Builder UI that mirrors the programmatic query construction, provides a raw SQL editor for advanced scenarios, and features a JSON viewer that properly formats documents, metadata, and system fields.

Visual interfaces like this have been shown to reduce developer errors and improve understanding of query semantics[1]. Integrating this interface directly with the ODM means developers can inspect results and verify assumptions during development without switching tools or writing throwaway scripts.

5.6 Summary

The implementation stays intentionally lightweight while remaining fully Cosmos-aware. Every design decision typed fields, schema validation, parameterized SQL generation, integrated visual tooling targets specific errors that happen when developers rely on either generic ORMs or raw SQL. These implementation details provide the foundation for evaluating how the system affects productivity and correctness in the results section.

6. Results & Analysis

We ran a controlled experiment comparing four ways to query Azure Cosmos DB: raw SQL, a Prisma-inspired fluent API, a Mongoose-style ODM, and our proposed Cosmos-specific ODM. This section breaks down what we found in terms of productivity, error rates, and how difficult developers found each method. The goal was to see if building an abstraction specifically for Cosmos DB actually helps developers work faster and make fewer mistakes.

6.1 Summary Comparison Table

Table 1 shows how the four approaches stack up across several dimensions. The rankings come from combining error counts, timing data, and developer feedback.

Table 1. Comparison of Query Approaches for Cosmos DB

The table highlights where each method excels or falls short on criteria like type safety, syntax reliability, schema enforcement, and alignment with Cosmos DB's architecture. These factors directly affect both how quickly developers can build features and how likely their code is to work correctly.

Feature / Criteria	Raw Cosmos SQL	Prisma-style	Mongoose-style	Proposed Cosmos ODM
Designed for Cosmos DB	No	No	No	Yes
Type safety	No	Partial	Weak	Strong
Partition key handling	No	No	No	Yes (automatic)

Feature / Criteria	Raw Cosmos SQL	Prisma-style	Mongoose-style	Proposed Cosmos ODM
Syntax error rate	High	Medium	Medium	Low
Field mismatch errors	High	Medium	High	Very low
Schema violations	Common	Medium	Medium	Very rare
Parameter safety	Manual	Partial	None	Automatic
Task completion time	Longest	Medium	Medium	Shortest
Serverless suitability	Yes	No (heavy)	Yes	Yes (lightweight)
Visual debugging	No	No	No	Yes (integrated)

Raw SQL and general-purpose ORMs offer some advantages but miss critical Cosmos-specific features like partition-aware operations, strong typing, and automatic parameterization. The proposed ODM consistently gave developers better guidance, fewer chances to make mistakes, and a smoother workflow built around how Cosmos DB actually works.

6.2 Productivity Outcomes

Developers finished tasks noticeably faster with the proposed ODM. Raw SQL took the longest because it required manually writing queries, debugging through trial and error, and figuring out partition keys without any help. Prisma and Mongoose patterns cut down on boilerplate but still forced developers to mentally translate concepts that don't map cleanly to Cosmos DB. People needed extra steps to adjust for Cosmos-specific quirks.

The proposed ODM consistently produced the fastest completion times for two main reasons. First, typed field references meant developers didn't waste time hunting for the right field names. Second, fluent query functions let them build valid queries without memorizing SQL syntax. Across all five tasks, the proposed ODM came out ahead on productivity.

6.3 Error Analysis

Error frequency showed the sharpest differences between methods.

Raw Cosmos SQL: Most mistakes were syntax errors or wrong field names. Developers often used incorrect SQL operators or forgot to parameterize values properly.

Prisma-style patterns: Errors mainly happened when developers assumed relational features like joins or nested conditions that don't exist in Cosmos DB.

Mongoose-style patterns: Weak typing led to typos in field names and updates that bypassed validation rules.

Proposed Cosmos ODM: Developers made the fewest errors thanks to type-checked fields, enforced schema validation, automatic parameterization, and warnings for risky cross-partition queries. The validation layer stopped malformed documents from getting inserted, which addresses schema drift problems documented in NoSQL research.

6.4 Cognitive Load Ratings

Developers rated raw SQL as the hardest method and said it gave them the least confidence in their work. Prisma and Mongoose patterns scored moderately difficult because their abstractions didn't match Cosmos DB's behavior well.

The proposed ODM got the lowest difficulty ratings and highest confidence scores. Developers appreciated clearer guidance while building queries, obvious field references, and consistency between the Explorer UI and programmatic queries.

6.5 Interpretation of Results

The proposed ODM beat all other approaches on every metric we measured. The results show that Cosmos-specific abstractions—not generic ORMs—are necessary for reliable and efficient development on Azure Cosmos DB. Combining type safety, schema validation, guided query building, and visual debugging meaningfully cuts down on developer mistakes and speeds up development. These findings match broader evidence that platform-aware tools improve correctness and reduce cognitive load when working with semi-structured NoSQL datasets.

7. Discussion

The experiment revealed something important: developers using the Cosmos-native ODM produced more accurate results, finished tasks more quickly, and felt more confident about their work than those relying on raw Cosmos SQL, Prisma-style interfaces, or Mongoose-style patterns. This matches what database usability research has been telling us for years—when developers face unfamiliar query syntax or work with loosely structured document models, they make more mistakes and the work feels harder[1]. Raw SQL certainly gives you power and flexibility. But handling syntax details, field names, parameterization, and partition keys all by yourself opens the door to plenty of errors and unpredictable outcomes.

The general-purpose abstractions we tested work fine in the ecosystems they were designed for, but Cosmos DB presents different challenges. Prisma pushes you toward relational thinking, which simply doesn't map onto Cosmos's document and partition architecture very well. This creates real confusion about how queries should be structured and how filtering actually works. Mongoose brings better data organization to the table, though it lacks robust type guarantees and doesn't enforce the structural discipline you need to avoid typical NoSQL data issues[5]. Both tools cut down on repetitive code, which is helpful. But neither one properly handles Cosmos-specific concerns like partitioning logic, continuation tokens, or managing request-unit costs effectively.

Our tool works differently by weaving validation, type guidance, and safe SQL generation into the normal development process. Combining strict schema enforcement with typed field references dramatically reduces problems like schema drift and field name errors—issues that keep appearing in NoSQL modeling research[9]. The Query Builder removes the guesswork around syntax and creates a predictable pattern that makes the work less mentally demanding, especially for developers who haven't spent much time with Cosmos. Adding the Explorer gives developers a visual way to verify and debug, so they can check whether data behaves as expected without writing extra diagnostic queries.

Raw SQL still matters in certain situations. When developers need complete control over query construction or execution strategy for highly specialized work, the ODM's abstractions might get in the way. Performance-critical

queries sometimes need manual optimization that goes beyond what we built. But these cases don't come up that often in regular development. For the vast majority of everyday tasks, our results indicate that a Cosmos-specific abstraction serves developers much better.

What this really comes down to is a fundamental principle: NoSQL tooling needs to understand the platform it's working with. Generic ORMs and database-agnostic abstractions can handle basic operations just fine, but they miss the platform-specific constraints and guarantees that actually ensure correctness and long-term maintainability. The ODM we built provides an alternative that's properly aligned with how Cosmos DB actually works both structurally and operationally. That alignment produces a development experience that's more reliable and lets developers get more done.

8. Conclusion

This study looked at how different approaches to working with Azure Cosmos DB affect developer productivity, accuracy, and overall usability. We ran a controlled experiment comparing raw Cosmos SQL, Prisma-style fluent interfaces, Mongoose-style ODM patterns, and the Cosmos-native ODM we developed. The results showed that generic abstractions and manual query writing consistently introduce problems—syntax mistakes, field mismatches, schema inconsistencies. This mirrors what broader NoSQL research has been finding: when developers lack platform-aware tools, data operations become unreliable and error-prone[11].

The ODM we built tackles these problems by bringing together Zod-based schema validation, strong type safety, parameterized SQL generation, typed field references, and an Explorer interface designed specifically around how Cosmos DB actually operates. Across every metric we measured task completion time, error frequency, cognitive load the ODM consistently beat both raw SQL and the generalized patterns inspired by Prisma and Mongoose. Participants wrote more correct queries, needed fewer attempts to get things right, and felt more confident about their results.

Raw SQL certainly still has value when you need specialized or heavily optimized queries. But the findings make it clear that for most everyday development work, a Cosmos-specific abstraction provides a more reliable and productive workflow. The study underscores why platform-aligned tooling matters so much for cloud-native NoSQL systems. When abstractions don't match the platform, they hide how the database actually works and create avoidable errors. Future work could test this with larger participant groups, more complex workloads, or production-scale evaluations to further validate and refine what the ODM accomplishes.

What this research ultimately demonstrates is that thoughtfully designed, domain-specific abstractions can genuinely improve both developer experience and data reliability in modern distributed applications. Generic ORMs and query builders simply can't capture the operational details that matter for Azure Cosmos DB. The platform's specific characteristics demand tools built with those characteristics in mind.

9. References

1. Svarre, Tanja, and Tony Russell-Rose. 2022. “Think Outside the Search Box: A Comparative Study of Visual and Form-Based Query Builders.” *arXiv*. https://arxiv.org/abs/2205.04212.

2. Fernández-Candel, Carlos J., David Sevilla-Ruiz, and Juan J. García-Molina. 2021. “A Unified Metamodel for NoSQL and Relational Databases.” *arXiv*. https://arxiv.org/abs/2105.06494.

3. Decan, Alexandre, Maël Goeminne, and Tom Mens. 2017. “On the Interaction of Relational Database Access Technologies in Open-Source Java Projects.” *arXiv*. https://arxiv.org/abs/1701.00416.

4. Lu, Jingwei, Yifan Song, Zhenyu Qin, Hongyang Zhang, Changxing Zhang, and Raymond Chi-Wing Wong. 2025. “Enabling Natural Language Queries for NoSQL Databases Through Text-to-NoSQL Translation.” *arXiv*. https://arxiv.org/pdf/2502.11201.

5. Scherzinger, Stephan, Markus Klettke, and Ulf Störl. 2013. “Managing Schema Evolution in NoSQL Data Stores.” *arXiv*. https://arxiv.org/pdf/1308.0514.

6. Hernández-Chillón, Alejandro, Markus Klettke, David Sevilla-Ruiz, and Juan García-Molina. 2022. "A Taxonomy of Schema Changes for NoSQL Databases." *arXiv*. https://arxiv.org/abs/2205.11660.

7. Fernández-Candel, Carlos J., Anne Cleve, and Juan J. García-Molina. 2025. "Towards the Automated Extraction and Refactoring of NoSQL Schemas from Application Code." *arXiv*. https://arxiv.org/pdf/2505.20230.

8. Ba, Yang, and Manuel Rigger. 2024. "Towards a Unified Query Plan Representation." *arXiv*. https://arxiv.org/pdf/2408.07857.

9. Belefqih, Sahar, et al. 2020. "Semantic Schema Extraction in NoSQL Databases." *Data Science Journal*. https://datascience.codata.org/articles/1688/files/6752d0abc93d2.pdf.

10. Anonymous. 2024. "Design and Development of a Unified Query Platform as Middleware for NoSQL Datastores." *International Journal of Advanced Computer Science and Applications*. https://thesai.org/Downloads/Volume15No7/Paper_62-Design_and_Development_of_a_Unified_Query_Platform.pdf.

11. Nurhadi, N., Rabiah Abdul Kadir, and Ely Salwana Mat Surin. 2022. "Complex SQL–NoSQL Query Translation for Data Lake Management." *Journal of Computer Science*. https://thescipub.com/pdf/jcssp.2022.1179.1188.pdf.

12. Aguilar, R., et al. 2023. "NoSQL Database Modeling and Management: A Systematic Literature Review." *ResearchGate*. https://www.researchgate.net/publication/375665580_NoSQL_Database_Modeling_and_Management_A_Systematic_Literature_Review.

13. Khan, Waseem. 2023. "SQL and NoSQL Database Software Architecture: A Software Engineering Perspective." *Informatics*. https://www.mdpi.com/2504-2289/7/2/97.

14. Roy, Abhilash. 2017. "A Single Access Platform for Different Structural NoSQL and SQL Databases." *Norma Academic Repository*. https://norma.ncirl.ie/3297/1/abhilashroy.pdf.

15. Kaura, H., 2019. "Analysis of NoSQL Database State-of-the-Art Techniques." *Semantic Scholar*. https://pdfs.semanticscholar.org/2db8/1a71211f00bebd8182a703d44c7eed27d361.pdf.