Journal Publication of International Research for Engineering and Management (JOIREM)
Volume: 04 Issue: 02 | Feb-2026
ISSN (O) 3107-6696

A Comparative Study of SQL and NoSQL Databases for Cloud-Based Web Applications: Performance, Scalability, and Use Case Analysis

K Gnana Harish Babu[1], Dr K Venkata Naganjaneyulu[2], S Rama Krishna Sarma A[3], S. Narasimha Murthy[4], M Chathurya[5], M Vyshnavi[6]

[1] 3rd Year B.Tech CSE, Malla Reddy University, Hyderabad.

[2] Professor, School of Computer Science & Engineering, Malla Reddy Engineering College for Women (Autonomous), JNTU-Hyderabad, Telangana State, India. Email: kvnaganjaneyulu75@gmail.com

[3] Assistant Professor, Department of Computer Science and Engineering, Malla Reddy Engineering College for Women, Misammaguda, Hyderabad, Telangana. Email: srksanupindi@gmail.com

[4] Assistant Professor, Department of Computer Science and Engineering, Malla Reddy Engineering College for Women's, Hyderabad. Contact: 8143443020

[5] 3rd Year B.Tech CSE, Malla Reddy University, Hyderabad.

[6] 4th Year CSE, Malla Reddy Engineering College For Women, Hyderabad, Telangana

---

Abstract — Picking the right database for a cloud application sounds simple until you actually have to do it. SQL systems like MySQL have been the go-to for decades, and for good reason — but as applications started dealing with more varied data at larger scale, NoSQL systems like MongoDB began showing up in more and more production stacks. The question was never really which one is better. It's which one actually fits your situation. This paper grew out of that frustration. We wanted something more concrete than the usual "it depends" answer, so we put both systems through real tests on the same dataset — 100,000 e-commerce records — and measured what actually happened: query speeds, read and write throughput, storage consumption, and how each system held up as data volume grew. MySQL was faster when queries got complicated, particularly joins and aggregations. MongoDB pulled ahead on write speed and handled document-style data more naturally, though it used noticeably more storage for the same records. We also cover the theory — ACID, CAP, BASE — not to pad the paper, but because those concepts genuinely explain why the benchmarks turned out the way they did. The goal is simple: give developers and architects a clear enough picture to make this call confidently for their own application.

Keywords: SQL, NoSQL, MySQL, MongoDB, Cloud Databases, Query Optimization, ACID Properties, CAP Theorem, Benchmarking, Distributed Systems, Scalability, Web Applications

1. INTRODUCTION

If you were building a web application ten years ago, the database question was mostly settled before you even asked it. You picked MySQL, or maybe PostgreSQL, and you moved on. Those systems had earned their reputation — they were reliable, well-understood, and backed by decades of tooling. The relational model fit most applications well enough that questioning it felt unnecessary.

That's changed. Today's applications — e-commerce platforms with millions of concurrent users, social networks generating billions of events per day, healthcare systems that need real-time access across distributed infrastructure — put pressures on a database that the traditional relational model wasn't really designed to handle. Vertical scaling, the main lever SQL systems offer, works until it doesn't, and at that point the costs get serious.

The NoSQL movement emerged largely because some of the biggest technology companies in the world hit exactly that wall. Google, Amazon, and Facebook couldn't just keep buying bigger servers. So they built different kinds of databases — systems that prioritized horizontal scalability and fault tolerance over strict consistency — and eventually published the research that let the rest of the industry follow. Document stores, key-value stores, column-family databases, and graph databases each took a different approach to the same underlying problem.

But the proliferation of options hasn't made the decision easier. If anything, it's harder now. There's a lot of noise in this space — NoSQL advocates who treat relational databases as legacy baggage, and SQL traditionalists who dismiss document stores as hype. Neither camp is particularly helpful if you're trying to make a real architectural decision for a real application.

This paper tries to cut through that. We compare MySQL and MongoDB directly — both through the theory and through hands-on benchmarks — and we translate those findings into

practical guidance. The contributions are three: a grounded theoretical comparison covering ACID versus BASE and what the CAP theorem actually means for system design; experimental results on query latency, throughput, and storage using a standardized dataset; and a decision framework that maps what your application actually does to the system that handles it best.

2. RELATED WORK

The academic literature on this topic is substantial, and a few studies in particular shaped how we approached this work. Cattell's 2011 survey was one of the first serious attempts to map the NoSQL landscape systematically. He laid out the fundamental trade-offs between consistency and availability in distributed systems in a way that still holds up, and the taxonomy he introduced continues to structure how the field thinks about these systems.

Abramova and Bernardino (2013) took a more empirical approach, benchmarking Cassandra against MySQL and showing that NoSQL systems can achieve significantly higher write throughput under heavy load. Useful work, but their focus on column-family stores left an obvious gap — MongoDB operates on a fundamentally different model, and it wasn't clear how their findings would translate.

Győrödi et al. (2015) addressed that gap more directly by putting MySQL and MongoDB head to head in a web application context. They found MongoDB ahead on insert-heavy workloads and MySQL ahead on complex multi-table queries — a pattern our results largely confirm. What their study didn't dig into was storage efficiency or cloud deployment characteristics, both of which matter a lot in practice. That's where our work picks up.

Parker et al. (2013) examined consistency models across distributed databases and made a point worth emphasizing: the choice between strong and eventual consistency isn't just a performance knob, it's an architectural decision that shapes how you write application logic. Nayak et al. (2013) provided a useful classification of NoSQL types by domain suitability, which informed how we structured our experimental scenarios.

What's still largely missing from the literature is a study that combines solid theoretical grounding with practical benchmarks specifically targeting cloud-hosted web applications. Most existing comparisons treat the database as an isolated system rather than one component in a larger deployment. This paper is an attempt to fill that gap.

3. BACKGROUND

3.1 SQL Databases and ACID Properties

Relational databases organize data into tables with fixed, predefined schemas, and enforce relationships between records through foreign keys and constraints. What defines them, though, is the transaction model. SQL systems implement what's known as ACID guarantees — four properties that together ensure your data stays correct even when things go wrong.

Atomicity means a transaction is all-or-nothing. If you're transferring money between two accounts and the debit succeeds but the credit fails, atomicity ensures the whole thing gets rolled back — no money disappears into the void. Consistency means every successful transaction leaves the database in a valid state, respecting every constraint and rule you've defined. Isolation means concurrent transactions don't step on each other — from each transaction's perspective, it's as if no other transaction is running at the same time. Durability means once a transaction commits, it stays committed. A crash immediately afterward won't undo it.

MySQL implements all four of these, and it does so while supporting the full range of relational query operations — joins, subqueries, aggregations, stored procedures. Its indexing infrastructure, including B-tree and hash indexes, lets the query optimizer find efficient paths through complex queries. For applications where data integrity is non-negotiable, this is exactly what you want.

3.2 NoSQL Databases and the CAP Theorem

NoSQL databases start from a different premise entirely. Instead of defining your data structure upfront and enforcing it on every write, they let the structure evolve. MongoDB stores records as BSON documents — essentially binary JSON — which means you can represent a customer with all their orders as a single nested document rather than splitting the data across multiple tables and reassembling it with joins at query time. For certain access patterns, that's a meaningful advantage.

The theoretical framework that explains why distributed databases have to make hard choices is the CAP theorem, introduced by Eric Brewer in 2000. It says that a distributed system can guarantee at most two of three properties: Consistency (every read gets the most recent write), Availability (every request gets a response), and Partition Tolerance (the system keeps working even if parts of the network stop talking to each other).

In practice, you can't opt out of partition tolerance — network failures happen, and a system that just stops when they do isn't useful. So the real trade-off is between consistency and availability. Most NoSQL systems choose availability, adopting what's called the BASE model: Basically Available, Soft-state, Eventually consistent. Data might be slightly stale for a moment after a write, but the system stays up and

responsive. That trade-off is what enables horizontal scaling across many nodes without coordination overhead strangling performance.

4. METHODOLOGY

4.1 Experimental Setup

All tests were run on a single machine — an Intel Core i5 system with 8 GB of RAM and a 256 GB SSD running Ubuntu 22.04 LTS. Both MySQL 8.0 and MongoDB 6.0 were installed with default configurations. We didn't tune either system, because the goal was a baseline comparison that reflects what a team would see when getting started, not a peak-performance shootout between heavily optimized deployments. Test scripts were written in Python 3.10.

Parameter	Specification
Processor	Intel Core i5 (3.2 GHz)
RAM	8 GB DDR4
Storage	256 GB SSD
OS	Ubuntu 22.04 LTS
MySQL Version	8.0
MongoDB Version	6.0
Programming Language	Python 3.10
Dataset	E-Commerce Orders Dataset (Kaggle)

Table 1: Experimental Setup Specifications

4.2 Dataset

We used a publicly available e-commerce orders dataset from Kaggle with 100,000 records. Each record covers order ID, customer ID, product information, order date, quantity, and total price — the kind of data a real application would actually store and query. We loaded it into MySQL as normalized relational tables and into MongoDB as a document collection, keeping the underlying data identical in both cases. Changing the data model would have introduced variables we didn't want; we specifically wanted performance differences to reflect the database engine, not modeling choices.

4.3 Metrics Measured

We focused on five things: Query Response Time (how long standard reads took, in milliseconds), Write Throughput (how many records per second each system could insert under sustained load), Read Throughput (records retrieved per second), Storage Consumption (how much disk space the same 100,000 records occupied in each system), and Scalability (how the numbers changed as we scaled from 10,000 to 100,000 records). Together these cover the dimensions that actually matter when you're deciding which system to use.

5. RESULTS AND ANALYSIS

5.1 Query Response Time

The query results were more nuanced than a simple "one system wins" story. For simple lookups by ID and date range queries, MongoDB was slightly faster — 1.8 ms versus 2.3 ms for ID lookups, 12.1 ms versus 14.7 ms for range queries. When a query maps directly to a single document, MongoDB's model has a natural edge because there's nothing to join.

Aggregation queries flipped the picture. MySQL handled SUM and COUNT operations in 18.4 ms; MongoDB took 22.6 ms — about 23% slower. The most dramatic gap showed up in multi-table joins: MySQL completed them in around 45 ms, while MongoDB needed nearly 90 ms. That gap reflects something fundamental — MongoDB wasn't designed for relational queries, and asking it to simulate joins across collections costs real performance. Full-text search went the other direction, with MongoDB finishing in 19.3 ms versus MySQL's 31 ms.

Query Type	MySQL (ms)	MongoDB (ms)
Simple SELECT (by ID)	2.3	1.8
Range Query (date filter)	14.7	12.1
Aggregation (SUM, COUNT)	18.4	22.6
Multi-table JOIN	45.2	89.4
Full-text Search	31.0	19.3

Table 2: Average Query Response Time Comparison

5.2 Read/Write Throughput

MongoDB inserted records about 23% faster than MySQL — 5,184 records per second against MySQL's 4,210. That gap makes sense once you understand what's happening under the hood. Every MySQL write goes through schema validation, constraint checking, and index maintenance. MongoDB skips most of that. When you're inserting at high volume and every millisecond of overhead adds up, that difference is real.

Read throughput told a closer story — 6,830 records per second for MySQL, 6,470 for MongoDB on straightforward queries. The gap widened when queries got more complex, which tracks with the response time findings. For simple retrieval, the systems are basically comparable. For analytical reads involving joins or cross-collection aggregations, MySQL pulls ahead.

Operation	MySQL	MongoDB
Write Throughput (records/sec)	4,210	5,184
Read Throughput (records/sec)	6,830	6,470
Storage Consumption (100K records)	42 MB	61 MB

Table 3: Throughput and Storage Comparison

5.3 Storage Efficiency

MySQL stored the full 100,000-record dataset in 42 MB. MongoDB needed 61 MB for exactly the same data — about 45% more. The reason is structural. In BSON, every document carries its own field names. So across 100,000 documents, the string "customer_id" is stored 100,000 times. In a relational

table, it appears once in the schema definition. That redundancy is the price of schema flexibility, and in storage-constrained or cost-sensitive cloud environments, it's worth knowing about upfront.

6. DISCUSSION

Looking at these results together, the takeaway isn't that one system is better. It's that each system is better at different things, and the question is which things matter most for what you're building.

6.1 When to Choose MySQL

If your application lives and dies by relational queries — complex joins, multi-table aggregations, anything that involves assembling a coherent picture from normalized data — MySQL is the right call. The same goes for anything where data integrity is genuinely non-negotiable. Financial transactions, medical records, inventory systems where an inconsistent state could mean real-world consequences — these need ACID guarantees, and MySQL delivers them reliably. If your data has a stable, well-defined structure and you're operating in an environment where storage costs are a real concern, the relational model's efficiency is a practical advantage too.

6.2 When to Choose MongoDB

MongoDB is the better fit when write throughput is a first-class concern. Logging pipelines, event streams, IoT sensor data, user activity tracking — any scenario where you're inserting at high volume and strict consistency matters less than keeping up with the firehose. It also shines when the data model is likely to change. Early-stage products where the schema is still being figured out, content management systems where document shape varies by content type, rapid prototyping where you don't want a schema migration every time the requirements shift — these are MongoDB's natural home. And if horizontal scaling across cloud infrastructure is in the roadmap, MongoDB's built-in replication and sharding make that path considerably smoother than MySQL's.

6.3 Hybrid Approaches

It's worth saying clearly that these aren't mutually exclusive options. A lot of production systems use both, which the industry calls polyglot persistence. An e-commerce platform might run MySQL for orders and payments — where every transaction needs to be exactly right — and MongoDB for the product catalog, user sessions, and recommendation data, where flexibility and speed matter more than strict consistency. Yes, that means two systems to maintain and monitor. But it also means each part of the application uses the storage model that actually fits it, rather than forcing one model to do everything it wasn't designed for.

7. CONCLUSION AND FUTURE WORK

We started this paper with a practical question — how do you actually choose between MySQL and MongoDB for a cloud-based application — and we tried to answer it seriously, through both theory and measurement. What the benchmarks showed is that both systems are genuinely good at what they were designed for. MySQL is the stronger choice when queries are relational and complex, when data integrity is critical, and when storage efficiency matters. MongoDB earns its place when write throughput is high, when the data model needs to stay flexible, and when the system needs to scale horizontally.

The honest conclusion is that the "right" database is whichever one fits your specific workload. That sounds obvious, but it's surprisingly easy to make this decision based on familiarity or trend rather than actual requirements. We hope the framework here makes it easier to think through that clearly.

There's plenty left to explore. This study used a single-node setup for both systems, which doesn't reflect how most cloud applications actually run. A natural next step is running the same benchmarks on distributed multi-node configurations — that's where the architectural differences between MySQL and MongoDB become most pronounced. We also want to bring PostgreSQL and Cassandra into the comparison, and to test managed cloud services like AWS RDS and MongoDB Atlas rather than local installs. Emerging hybrid databases like CockroachDB and FaunaDB, which try to offer relational semantics with horizontal scalability, are an interesting development worth investigating too.

REFERENCES

1. Cattell, R. (2011). Scalable SQL and NoSQL Data Stores. ACM SIGMOD Record, 39(4), 12–27.

2. Abramova, V., & Bernardino, J. (2013). NoSQL Databases: MongoDB vs Cassandra. Proceedings of the International C* Conference on Computer Science and Software Engineering, 14–22.

3. Győrödi, C., Győrödi, R., Pecherle, G., & Olah, A. (2015). A Comparative Study: MongoDB vs. MySQL. Proceedings of the 13th International Conference on Engineering of Modern Electric Systems, 1–6.

4. Parker, Z., Poe, S., & Vrbsky, S. V. (2013). Comparing NoSQL MongoDB to an SQL DB. Proceedings of the 51st ACM Southeast Conference, Article 5.

5. Nayak, A., Poriya, A., & Poojary, D. (2013). Type of NOSQL Databases and its Comparison with Relational Databases. International Journal of Applied Information Systems, 5(4), 16–19.

6. Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, 13(6), 377–387.

7. Brewer, E. A. (2000). Towards Robust Distributed Systems. Proceedings of PODC, 19th ACM Symposium on Principles of Distributed Computing, 7.

8. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., et al. (2007). Dynamo: Amazon's Highly Available Key-Value Store. ACM SIGOPS Operating Systems Review, 41(6), 205–220.

9. Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., et al. (2008). Bigtable: A Distributed Storage System for Structured Data. ACM Transactions on Computer Systems, 26(2), Article 4.

10. MongoDB, Inc. (2023). MongoDB Documentation: Data Modeling. Retrieved from https://www.mongodb.com/docs/

11. Oracle Corporation. (2023). MySQL 8.0 Reference Manual. Retrieved from https://dev.mysql.com/doc/

12. Han, J., Haihong, E., Le, G., & Du, J. (2011). Survey on NoSQL Database. Proceedings of the 6th International Conference on Pervasive Computing and Applications, 363–366.

13. Stonebraker, M. (2010). SQL Databases v. NoSQL Databases. Communications of the ACM, 53(4), 10–11.

14. Velte, A., Velte, T., & Elsenpeter, R. (2010). Cloud Computing: A Practical Approach. McGraw-Hill Education.

15. Fowler, M., & Sadalage, P. J. (2012). NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional.