A Comparative Study of PostgreSQL and MongoDB in a Banking Application

Aidan Ryther
Rochester Institute of Technology
arr9180@g.rit.edu

Tiffany Lee
Rochester Institute of Technology
tl5110@rit.edu

Vivian Hernandez
Rochester Institute of Technology
vah7365@rit.edu

ABSTRACT

We evaluate SQL and NoSQL for a small banking application by implementing the same features with PostgreSQL and MongoDB Community Server. The application supports a ton of different features like opening accounts, depositing, withdrawing, transferring, balancing checks, and a view of your transactions. Both systems run with ACID transactions and durable writes. We compare three things. For development, we track how long each feature takes and how much code touches the database, plus any changes to schemas or indexes. For query complexity, we count joins in SQL and note how involved the equivalent MongoDB queries are. We measure performance by looking at throughput and response time while running a fixed mix of reads and writes under different levels of concurrency on the same machine with the same setup. Using this banking application, our goal is to see whether SQL or NoSQL is the better choice when strong consistency is prioritized over availability, and to point out tradeoffs that go beyond this single case.

KEYWORDS

Banking, PostgreSQL, MongoDB, SQL, NoSQL, ACID Transactions, OLTP, Performance

1 INTRODUCTION

Retail banking puts correctness ahead of availability. Balances must be exact, transfers committed as one unit, and two concurrent withdrawals from the same account should not both succeed. This paper looks at how a relational model and a document model handle that basic workload. We build the same small banking service twice: once on PostgreSQL and once on MongoDB Community, both configured for ACID transactions and durable writes. Now to keep the runs fair and repeatable, we follow the benchmarking best practices that have been thoroughly tested in previous experiments [3, 8]. To keep these experiments grounded, we'll mainly be comparing the real world results.

In comparing SQL and NoSQL there are too many tradeoffs for one paper, so we keep the scope tight around a concrete system design problem. A lot of online systems are built to keep running smoothly by prioritizing availability, even if that means relying on eventual consistency. Banking is the opposite. Money movements must be exact, so we use a small banking app to see how each model handles strict correctness. PostgreSQL is our relational baseline and MongoDB Community stands in for a document store. We hold hardware, dataset size, and index plans fixed so that the only differences we see come from the data model and transaction behavior, not doing trickery with the settings [1, 3, 8].

We picked PostgreSQL because it is widely used and supports ACID transactions. It has real constraints, good indexing, and it's simple to run on one machine. We chose MongoDB Community Server because it's very easy to set up, one of the most popular choices with programmers, allows us to do multi-document transactions, and gives a flexible schema when you need to have flexible schemas. Both are free to run self hosted, so we can keep everything on the same box and control the settings. That lets us line up indexes, write durability, and transaction settings so the runs are fair. The data maps cleanly in both cases too. We have accounts and transactions, with keys, balances, and history. With this setup, any differences we see should come from the data model and query style, not from service [1].

From this experiment, we want to figure out which database fits better for any system that values consistency over availability, such as our bank app. You will see the same features built two ways, one in PostgreSQL and one in MongoDB, and what that teaches us. With PostgreSQL we keep accounts and transactions in related tables with keys and checks, and queries join them to move money and show recent activity. In MongoDB, we store data in documents, sometimes embedding recent activity with the account and sometimes keeping it in a separate collection and referencing it. These choices will shape the indexes we build, the way reads and writes are handled, and how much of the code has to interact with storage. We follow a simple test plan. First, increase concurrent users, then report latency and throughput so the results can be checked and repeated [8]. By the end, you should have a clear view of the tradeoffs you would feel in practice, including developer effort, query shape, and speed, so you can judge when SQL or NoSQL makes more sense for a correctness first app.

At a high level, we compare three areas. First, development, which is recording the time it takes each feature to build and how much code touches the database [2]. Second, query shape. In PostgreSQL that means joins, constraints, and a fixed schema. In MongoDB that means reading and updating documents, which sometimes have references in them, and a flexible schema [1]. We also see how these choices line up with common SQL versus NoSQL tradeoffs, particularly taking note of which indexes each one needs [1]. Third, their performance, or runtime behaviors as we add concurrent users. Using the same machine we report their throughput and response times, following basic benchmarking guidance to avoid common traps [3, 8]. At the same time we enforce the banking rules (atomic transfers, no negative balances), but we do not rate security features or admin tools. The goal is to show the costs you feel while building and what happens when the app is busy so you can map the results to your own case.

Conference'17, July 2017, Washington, DC, USA
2025. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 PLAN TO EVALUATE

For our evaluation, we maintain the same banking rules no matter which database we use. Each account has an id and a balance that cannot go below zero unless an overdraft is on. When first opening an account, the user starts at zero. Deposits and withdrawals are performed in a single transaction. A transfer debits one account and credits the other as one unit, and we tag each transfer with an external id so a retry does not double charge. The transaction log is appended only with time, amount, and account ids, and reads only show committed data. On the database side, we turn on ACID transactions and durable writes in both systems. In PostgreSQL we rely on keys and checks, and wrap money moves in a transaction. In MongoDB, we use multi-document transactions with majority writes and snapshot-style reads. We keep the index plan aligned in intent on both builds, so queries follow the same access paths even if the definitions differ. The rest of the setup stays fixed, too. This means that when testing, we plan to use the same machine, same client mix, same pool sizes, a short warm-up, and timed runs. These choices follow basic guidelines on fair testing and avoiding configuration tricks [3, 8].

2.1 Data Models and Indexes

In PostgreSQL we use two tables:

- accounts(id, balance, overdraft, ...)
- transactions(id, account_id, amount, type, transfer_id, ...)

PostgreSQL primary and foreign keys allow us to tie rows together, and we use CHECK constraints to handle simple rules like non-negative amounts. When we move money between two accounts, we save two records with the same transfer_id, one for the debit and one for the credit. They're saved together in a single transaction, so either both are written or neither is, which prevents half-finished transfers. To improve speed, we use a composite B-tree index on recent activity per account (account_id, ts DESC) and a unique index on transfer_id to prevent duplicates [6, 7].

In MongoDB we mirror that shape with two collections:

- accounts
- transactions

Account documents hold the current balance and flags. Transaction documents store { accountId, ts, amount, type, transferId, ... }. Reads and writes work on whole documents, and when we need a link we keep accountId. We create a compound index { accountId: 1, ts: -1 } for "recent activity per account" and a unique index on { transferId: 1 } to stop a double post [4, 5]. The access paths line up in intent across both systems even though the definitions are different.

2.2 Workload mix, dataset size, and concurrency levels

For this project, we will design a workload that mirrors operations of a real banking system. The operations include opening new accounts, deposits, withdrawals, transfers, balance checks, and viewing recent transactions. Deposits and withdrawals operations will be the most frequent, followed by balance checks and transfers. Account creation will be less common. Transfers will test whether each system can handle multi-record transactions correctly.

The dataset will consist of approximately 100,000 accounts and approximately 1 million transactions. This size is large enough to mirror real-world banking systems while remaining feasible on a single machine. To evaluate concurrency handling, the workload will be executed with different numbers of clients (1, 5, 10, 20, and 50) running simultaneously. This variation will allow us to observe how each system performs under both light and heavy concurrent loads.

2.3 Metrics to Report and How We Record Them

We will measure three categories of metrics to provide a balanced evaluation.

- **Development effort:** measured by the time it takes to build each feature, how many lines of code directly work with the database, and how many schema or index changes are required.
- **Query and model complexity:** measured by the number of joins in PostgreSQL queries, the complexity of MongoDB aggregation pipelines or transactions, and a qualitative look at how readable and maintainable the queries are.
- **Performance:** measured by transactions per second and response latency at different concurrency levels. Each test will be run several times and the results will be averaged to smooth out the random variation.

Altogether these three areas let us see the tradeoffs between ease of development, complexity of queries, and runtime performance.

2.4 Fairness Rules, Hardware, and Run Procedure

As mentioned earlier, PostgreSQL and MongoDB both run on the same machine to keep things fair. This machine will specifically have a 4-core CPU with 16GB RAM and SSD storage. Each will be set up with durable writes and ACID transactions turned on. We'll use equivalent indexing strategies so neither database has an edge, and we'll avoid system-specific optimizations that could tilt the results toward one side.

The procedure will be consistent across systems. First, the dataset will be loaded into the database. Then, a warm-up phase with preliminary queries will be executed to bring caches and buffers into a steady state. Finally, the workload will be run at differing concurrency levels (1, 5, 10, 20, 50 clients). Each experiment will be repeated several times, and the average results will be reported to improve reliability.

2.5 Limits and Threats to Validity

There are a few limits we should acknowledge. The workload and dataset are simpler than a real banking system and don't cover advanced features. Both databases are tested on a single node, so results may not represent distributed setups like PostgreSQL replication or MongoDB sharding, which would matter for scalability and availability. Measuring development effort is also partly subjective and may depend on the developer's background with SQL or MongoDB since familiarity could affect how quickly features are implemented and how complex the code ends up being.

REFERENCES

[1] Astera Analytics Team. 2025. SQL vs. NoSQL: 5 Main Differences. Astera Knowledge Center blog. Retrieved 2025-10-01 from https://www.astera.com/knowledge-center/sql-vs-nosql/

[2] Carlos Eduardo Carbonera, Kleinner Farias, and Vinicius Bischoff. 2020. Software development effort estimation: a systematic mapping study. *IET Software* 14, 4 (2020), 328–344. doi:10.1049/iet-sen.2018.5334

[3] Mat Keep and Henrik Ingo. 2020. Performance Best Practices: Benchmarking. MongoDB Blog. Retrieved 2025-10-01 from https://www.mongodb.com/company/blog/performance-best-practices-benchmarking Updated April 20, 2020.

[4] MongoDB, Inc. 2025. *MongoDB Manual: Data Model Design.* Retrieved 2025-10-02 from https://www.mongodb.com/docs/manual/data-model-design/

[5] MongoDB, Inc. 2025. *MongoDB Manual: Indexes.* Retrieved 2025-10-02 from https://www.mongodb.com/docs/manual/indexes/

[6] PostgreSQL Global Development Group. 2025. *PostgreSQL Documentation: Multicolumn Indexes.* Retrieved 2025-10-02 from https://www.postgresql.org/docs/current/indexes-multicolumn.html

[7] PostgreSQL Global Development Group. 2025. *PostgreSQL Documentation: Table Constraints.* Retrieved 2025-10-02 from https://www.postgresql.org/docs/current/ddl-constraints.html

[8] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. 2018. Fair Benchmarking Considered Difficult: Common Pitfalls in Database Performance Testing. In *DBTest'18: Workshop on Testing Database Systems* (Houston, TX, USA). Association for Computing Machinery, New York, NY, USA, 6 pages. doi:10.1145/3209950.3209955