Retail Data Engineering and Benchmarking Across SQL and NoSQL Systems

Group 6 (Shwetha Shenoy-Kamath, Swarnaditya Maitra, Hatim Urabi)

Northwestern University School of Professional Studies

MSDS 420

30 August 2025

ABSTRACT

This study evaluates design and performance trade-offs across three environments found in retail data platforms. Part 1 benchmarks surrogate keys versus composite keys for a large transactional fact table in MySQL, measuring query latency, maintenance cost, and storage footprint. Part 2 compares PostgreSQL and Microsoft SQL Server on an identical analytical workload to understand ingestion speed, concurrency behavior, resource usage, and on-disk size. Part 3 contrasts relational modelling in PostgreSQL with a document model in MongoDB for multi-table retail analyses, examining data-engineering complexity and query performance. Findings: surrogate keys yield more balanced performance across varied queries in MySQL; SQL Server loads faster while PostgreSQL sustains slightly higher throughput under concurrency; PostgreSQL's relational engine handles complex joins more directly than MongoDB, while MongoDB suits pre-aggregated or session-oriented views. These results align with recent comparative studies of DBMS performance and with reviews on SQL versus NoSQL suitability for analytics and operational workloads.

DATA AND CODE AVAILABILITY

All scripts, notebooks, outputs, and benchmarking artefacts are available in our GitHub repository: https://github.com/filterkaapicodes/sk-nwu-msds420-project-group6

TABLE OF CONTENTS

Retail Data Engineering and Benchmarking Across SQL and NoSQL Systems

1 INTRODUCTION

Retail organizations depend on timely, accurate views of transactions, promotions, and household behaviour. Engineering choices at the storage layer shape not only query performance but also the amount of modelling effort needed to deliver repeatable analyses. Modern choices span mature relational engines and a broad family of NoSQL systems. Contemporary literature indicates that relational DBMSs remain strong for complex joins and rigorous analytics, while NoSQL databases emphasise schema flexibility and horizontal scaling; the optimal choice is workload dependent (Khan et al. 2023; Taipalus 2024). This paper reports a three-part evaluation based on the dunnhumby Complete Journey dataset and a suite of retail queries. Part 1 isolates a MySQL schema decision that often arises in data warehousing: surrogate keys versus composite keys on a large fact table. Part 2 examines PostgreSQL and SQL Server side by side to gauge differences in ingestion and concurrent analytical processing. Part 3 implements equivalent questions in PostgreSQL and MongoDB to highlight modelling and performance implications across relational and document paradigms. Throughout, findings are cross-referenced to peer-reviewed studies that have compared SQL and NoSQL systems using controlled benchmarks.

2 DATA AND METHODS

2.1 DATASET

We use the Complete Journey dataset, which includes two years of transactions for roughly 2,500 households, plus product, campaign, coupon, and promotion tables. Recent documentation and user guides describe entities and relationships used for instruction and research. For a subset of analyses, we also reference dunnhumby's public user guides to confirm table semantics (Boehmke 2020; dunnhumby 2023). All code, results, outputs, and data analyses are in the Group 6 GitHub repository: https://github.com/filterkaapicodes/sk-nwu-msds420-project-group6

2.2 WORKLOADS

Analyses reflect four recurring retail questions: customer spending trends over time; demographic influences on purchasing; category growth and product trends among households with rising versus falling spend; and campaign and coupon impacts on sales. In relational systems these involve joins between transactions and dimensions, followed by groupings and filters. In MongoDB, we design document shapes and indexes to express equivalent pipelines using aggregations. Our measurements originate from this project's experimental runs.

2.3 TOOLING AND MEASUREMENT

For MySQL we profile with EXPLAIN and EXPLAIN ANALYZE to understand plan shapes and iterator costs; we also use OPTIMIZE TABLE when assessing maintenance overhead. For PostgreSQL and SQL Server we collect ingestion timings, throughput under concurrent load, resource usage, and database sizes. For MongoDB we measure pipeline latency under representative index configurations (Oracle 2024; Oracle 2025).

3 PART 1: SURROGATE KEYS VERSUS COMPOSITE KEYS IN MYSQL

3.1 DESIGN ALTERNATIVES

We engineered two tables with identical columns for transaction line items. The composite-key table used a primary key formed by the natural ordering attributes required by frequent filters, specifically store and day, with a uniqueifier as needed. The surrogate-key table used an auto-increment integer primary key and maintained secondary B-tree indexes on the same selective attributes. Both designs included equivalent foreign keys in staging to ensure referential integrity during load. Because InnoDB stores table data in the clustered index, primary key choice changes physical row order, storage layout, and the need for additional secondary indexes (Oracle 2025).

3.2 BENCHMARK QUERIES AND OPERATIONS

We ran four analytic queries across both designs: a spend trends query constrained by date and store; a demographic join and aggregate; a category growth analysis across two annual windows; and a campaign impact query involving multiple joins and roll-ups. We also measured six operational tasks, which are bulk insert throughput, targeted deletes over a date range, table rebuild time, and on-disk size after load. Plans were validated with EXPLAIN and EXPLAIN ANALYZE (Oracle 2024).

3.3 RESULTS

Across the analytic queries, the surrogate-key design was consistently faster except when the access pattern aligned exactly with the composite key's leftmost ordering. For a date-range-by-

store filter that matched the composite key, the composite table held a small edge. For demographic and category growth queries that required broader joins and groupings, surrogate keys were faster by roughly tens to a few hundred milliseconds per query on our dataset. In one representative category growth run, the surrogate design was about 20 percent faster. The heaviest campaign query also favoured the surrogate design, but with a small absolute gap because time was dominated by aggregation. On targeted maintenance, deleting a contiguous date slice was faster on the composite-key table because matching rows were adjacent in the clustered index. In one trial, deleting a day's range completed in approximately 91 ms on the composite design versus 119 ms on the surrogate design. The composite table also occupied less space on disk, reflecting the wider clustered index on the surrogate design and extra secondary indexes. Rebuilding with OPTIMIZE TABLE took longer on the surrogate table because more indexes were maintained. These observations are consistent with InnoDB's storage model (Oracle 2025; Oracle 2024).

3.4 INTERPRETATION

The results match engine internals and recent guidance: using a narrow surrogate primary key reduces clustering side effects on unrelated access patterns and allows composite secondary indexes tailored to several frequent predicates. A composite primary key can be excellent for one or two dominant range queries but less flexible for mixed workloads. Documentation highlights that EXPLAIN ANALYZE is essential for validating iterator-level costs and ensuring that additional indexes translate into real plan improvements (Oracle 2024). Takeaway for MySQL: favour a surrogate key plus targeted secondary indexes if your fact table supports diverse analytical queries; consider a composite primary key only when a single left-most ordering dominates and update patterns align with that clustering.

4 PART 2: POSTGRESQL VERSUS MICROSOFT SQL SERVER ON AN ANALYTICAL WORKLOAD

4.1 PURPOSE AND ENVIRONMENT

We compared PostgreSQL and Microsoft SQL Server using the same dataset, schema, and queries. The comparison focuses on ingestion time, throughput and latency under concurrent load, CPU and memory behaviour, and storage footprint. Licensing and ecosystem features were out of scope. The aim was to understand engine behaviour under OLAP-style queries common in retail.

4.2 RESULTS

SQL Server completed bulk CSV ingestion faster in our runs. PostgreSQL showed slightly higher sustained throughput and marginally lower latency under concurrent load once data were in place. Resource utilisation was moderate on both systems. PostgreSQL produced a smaller on-disk database after load, reflecting differences in storage and metadata structures. These findings are compatible with the broader literature. A recent synthesis notes that results vary by workload and configuration, but relational engines often trade small ingestion differences for throughput advantages once warmed under analytical concurrency (Taipalus 2024). Controlled studies using TPC-H-like workloads also show differing strengths across PostgreSQL, MySQL, and SQL Server depending on query mix and hardware, with notable gaps on individual queries but no single dominant system across all metrics (Vershinin and Mustafina 2021; Alves et al. 2023).

4.3 IMPLICATIONS

For retail analytics, the choice between PostgreSQL and SQL Server at the engine level is rarely decisive once operational constraints and skill sets are considered. SQL Server's faster CSV load may reduce batch window pressure; PostgreSQL's compact footprint and steady concurrency may lower storage cost and improve predictable latencies. Published comparisons reinforce that tuning, data distribution, and query shape dominate performance outcomes more than vendor identity for this class of workload (Taipalus 2024).

5 PART 3: POSTGRESQL VERSUS MONGODB FOR ANALYTICAL QUESTIONS

5.1 MODELLING APPROACH

We loaded the same retail dataset into PostgreSQL and into MongoDB. In PostgreSQL we used a conventional star-like schema with transaction facts and dimension tables. In MongoDB we designed collections to support the same questions with minimal lookups: a households collection with summary totals, a products collection with descriptive attributes, a baskets collection that embeds line items by basket, and large collections for coupons, redemptions, campaigns, and promotions. The goal was to keep denormalised documents aligned with query predicates while respecting document size limits.

5.2 ANALYSES AND OBSERVATIONS

Spending trends. Both systems produced comparable weekly trend series across two years. PostgreSQL expressed the query as joins and groupings. MongoDB's aggregation pipeline achieved the same result once nested arrays in basket documents were unwound with appropriate

indexes. Survey and experimental studies report that document stores perform well on CRUD and simple aggregations at scale, with scan-heavy patterns often more efficient in relational engines depending on index selectivity (Carvalho, Sa, and Bernardino 2023; Gyorodi et al. 2022).

Demographic influence. We joined transactions to household demographics to compare category preferences. In PostgreSQL this was a straightforward join and aggregation. In MongoDB we either pre-joined key demographic attributes into household documents or accepted more lookups. Denormalisation reduced pipeline complexity but introduced duplication to manage. This maps to review findings: NoSQL document stores trade referential integrity for flexibility, while relational systems remain efficient and simpler to express when analyses require multiple joins (Khan et al. 2023).

Category growth. We contrasted category trends for households with increasing versus decreasing spend year over year. PostgreSQL handled the multi-step join-group-filter pattern directly. MongoDB required careful pipeline staging with $group and $lookup and benefited from compound indexes. Comparative studies that include document stores and relational engines often find PostgreSQL outperforming MongoDB on join-heavy analytics, while MongoDB fares well when queries align with document locality and pre-aggregated fields (Makris et al. 2021; Carvalho, Sa, and Bernardino 2023).

Campaign and coupon effects. Measuring lift among redeemers versus non-redeemers required linking campaign assignments, coupon redemptions, and transactions. PostgreSQL expressed the joins cleanly. In MongoDB we either embedded campaign IDs on household documents or used $lookup across collections. Both systems produced convergent results, though MongoDB needed more modelling care to keep pipelines concise (Khan et al. 2023; Gyorodi et al. 2022).

5.3 QUANTITATIVE PERFORMANCE NOTES

In our runs, PostgreSQL's queries executed with low latencies for multi-join aggregates once indexes were in place. MongoDB performed well when pipelines operated within a single collection or when we embedded aggregates into documents for common retrieval paths. Where pipelines needed several $lookup stages, PostgreSQL held the advantage. Published results corroborate this qualitative split: PostgreSQL often leads on join-heavy workloads, while MongoDB performs well when data locality within documents is high (Makris et al. 2021; Carvalho, Sa, and Bernardino 2023).

5.4 ENGINEERING EFFORT

Relational modelling required fewer decisions for the questions we asked. Document modelling required explicit choices about which relationships to embed and which to reference. When the demand is for fast access to pre-aggregated or session-scoped views, MongoDB's document locality is attractive. When analysts routinely compose new joins, relational engines keep pipelines shorter. Recent reviews recommend selecting the paradigm that minimises cross-document joins or repeated reshaping for the dominant workload (Khan et al. 2023; Taipalus 2024).

6 OVERALL COMPARATIVE CONCLUSION

This section synthesizes results across the three comparisons and provides a concise decision aid. Table 1 summarizes key trade-offs drawn from this project's measurements and analyses.

Table 1 Summary comparison across scenarios

Aspect	**MySQL (SK vs CK)**	**PostgreSQL vs SQL Server**	**PostgreSQL vs MongoDB**
Performance	SK is faster on most join-heavy and non-aligned lookups. CK is slightly faster where filters align to (store, day). CK loads faster and uses less disk; SK adds index overhead.	SQL Server loads about 24% faster. PostgreSQL sustains about 7% higher throughput with slightly lower p95 latency. Total sizes are within about 12%.	PostgreSQL excels at multi-join analytics. MongoDB is fast for single-collection or embedded, pre-aggregated paths. $lookup-heavy pipelines can lag.
Complexity	SK increases index maintenance but makes it easier to add new access paths. CK is simpler to maintain but needs extra indexes if query patterns diversify.	SQL and administration are comparable; tuning differs by platform but development complexity is similar.	PostgreSQL queries are straightforward for cross-entity analysis. MongoDB needs denormalization choices and multi-stage pipelines to reach parity.
Data model fit	Both are relational. CK mirrors natural uniqueness. SK decouples clustering	Both fit the normalised retail schema equally well.	Relational model fits the full dataset. Document model fits baskets and other self-contained

	from access paths and is a common warehouse choice.		aggregates; very large, cross-entity relations are less natural.
Integrity	Both enforce foreign keys. CK's natural key prevents duplicate facts by construction. SK may need an extra unique constraint if that is required.	Both provide ACID guarantees and referential integrity.	PostgreSQL enforces keys and constraints. MongoDB relies on application-level checks for cross-collection consistency.
Flexibility	SK adapts better to new query patterns via secondary indexes. CK optimises one dominant access path.	Both are flexible; PostgreSQL has rich extensions and JSON; SQL Server has strong enterprise features.	MongoDB offers high schema flexibility and rapid evolution. PostgreSQL offers flexible ad-hoc querying across stable schemas.

7 CROSS-STUDY CONTEXT FROM RECENT LITERATURE

A 2023 systematic review observes that NoSQL databases are commonly selected for big-data analytics with high write scalability and flexible schemas, whereas SQL databases remain the default for OLTP and complex, set-based analytics that demand consistency (Khan et al. 2023). An MDPI 2023 evaluation of document stores finds MongoDB typically fastest across several

YCSB workloads except scans, which aligns with our observation that pipeline shape and locality drive outcomes (Carvalho, Sa, and Bernardino 2023). A GeoInformatica comparison reports PostgreSQL faster across many query classes and notes the strong effect of indexing choices (Makris et al. 2021). A 2024 synthesis highlights that configuration and workload dominate results, reinforcing the need to test with representative data and queries rather than relying on generic rankings (Taipalus 2024). For relational engines, conference work has compared PostgreSQL, SQL Server, and MySQL using TPC-H-style queries and reported mixed leadership across queries (Vershinin and Mustafina 2021; Alves et al. 2023).

8 RECOMMENDATIONS

MySQL fact tables. Use a surrogate key as the clustered index and add composite secondary indexes for frequent predicates. Reserve a composite primary key for cases where the workload is dominated by one left-most range that aligns with physical clustering. Validate with EXPLAIN ANALYZE (Oracle 2024).

Relational engine choice. Between PostgreSQL and SQL Server, prefer the engine that your team can operate and tune well. Expect SQL Server to load CSVs quickly and PostgreSQL to deliver compact storage and steady concurrency for analytics. Confirm on your hardware using a representative slice of the warehouse (Taipalus 2024; Vershinin and Mustafina 2021).

Relational versus document. Use PostgreSQL when analysts frequently compose new joins or when data quality demands strong integrity constraints. Use MongoDB for fast retrieval of pre-aggregated or session-oriented views and when reshaping documents is part of the application workflow. Recent studies support these patterns (Khan et al. 2023; Makris et al. 2021; Gyorodi et al. 2022).

9 REFERENCES

Alves, Luis, Carlos Wanzeller, Filipe Araujo, Pedro Moura, and Maryam Rezaei. 2023. "Comparison of Semi-Structured Data on MSSQL and PostgreSQL." In *Proceedings of ICMarkTech 2022*, 37–46.

Carvalho, Ines, Filipe Sa, and Jorge Bernardino. 2023. "Performance Evaluation of NoSQL Document Databases: Couchbase, CouchDB, and MongoDB." *Algorithms* 16 (2): 78. https://doi.org/10.3390/a16020078.

Gyorodi, Cornelia A., Diana V. Dumse-Burescu, Doina R. Zmaranda, and Robert S. Gyorodi. 2022. "A Comparative Study of MongoDB and Document-Based MySQL for Big Data Application Data Management." *Big Data and Cognitive Computing* 6 (2): 49. https://doi.org/10.3390/bdcc6020049.

Khan, Wisal, Teerath Kumar, Cheng Zhang, Kislay Raj, Arunabha M. Roy, and Bin Luo. 2023. "SQL and NoSQL Database Software Architecture Performance Analysis and Assessments — A Systematic Literature Review." *Big Data and Cognitive Computing* 7 (2): 97. https://doi.org/10.3390/bdcc7020097.

Makris, Antonios, Konstantinos Tserpes, Giannis Spiliopoulos, Dimitrios Zissis, and Dimosthenis Anagnostopoulos. 2021. "MongoDB vs PostgreSQL: A Comparative Study on Performance Aspects." *GeoInformatica* 25 (2): 243–268. https://doi.org/10.1007/s10707-020-00407-w.

Taipalus, Toni. 2024. "Database Management System Performance Comparisons." *Journal of Systems and Software* 207: 111872. https://doi.org/10.1016/j.jss.2023.111872.

Vershinin, I. S., and A. R. Mustafina. 2021. “Performance Analysis of PostgreSQL, MySQL, Microsoft SQL Server Systems Based on TPC-H Tests.” In *2021 International Russian Automation Conference (RusAutoCon)*, 683–687. https://doi.org/10.1109/RusAutoCon52004.2021.9537400.

Boehmke, Bradley. 2020. “The Complete Journey User Guide.” *CRAN Vignette*. https://cran.r-project.org/web/packages/completejourney/vignettes/completejourney.html.

dunnhumby. 2023. “Let’s Get Sort-of-Real: Dummy Data to Test Techniques and Algorithms. User Guide.” https://www.dunnhumby.com/wp-content/uploads/2023/08/Let_s-Get-Sort-of-Real-User-Guide-dunnhumby.pdf.

Oracle Corporation. 2024. “EXPLAIN Statement.” *MySQL 8 Reference Manual*. https://dev.mysql.com/doc/en/explain.html.

Oracle Corporation. 2024. “OPTIMIZE TABLE Statement.” *MySQL 8 Reference Manual*. https://dev.mysql.com/doc/refman/8.1/en/optimize-table.html.

Oracle Corporation. 2025. “InnoDB Indexes: Clustered and Secondary Indexes.” *MySQL 8 Reference Manual*. https://dev.mysql.com/doc/refman/8.4/en/innodb-index-types.html.