Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Μελέτη Απόδοσης Συστημάτων Lakehouse

Data Lakehouse Performance Study

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΚΩΝΣΤΑΝΤΙΝΟΣ ΣΙΔΕΡΗΣ

Επιβλέπων : Δημήτριος Τσουμάκος
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2025

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Μελέτη Απόδοσης Συστημάτων Lakehouse

Data Lakehouse Performance Study

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΚΩΝΣΤΑΝΤΙΝΟΣ ΣΙΔΕΡΗΣ

Επιβλέπων : Δημήτριος Τσουμάκος
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Ιουνίου 2025.

....	....	....
Δημήτριος Τσουμάκος Αναπληρωτής Καθηγητής Ε.Μ.Π.	Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2025

....

ΚΩΝΣΤΑΝΤΙΝΟΣ ΣΙΔΕΡΗΣ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Καθώς οργανισμοί υιοθετούν ολοένα και περισσότερο αρχιτεκτονικές lakehouse για την ανάλυση μεγάλων δεδομένων, είναι απαραίτητο να κατανοήσουμε την συμπεριφορά τους, καθώς και τους ενδεχόμενους συμβιβασμούς που συνεπάγεται η χρήση τους. Σκοπός της παρούσας εργασίας είναι η παρουσίαση μιας ολοκληρωμένης μελέτης της απόδοσης δύο διαδεδομένων συστημάτων lakehouse, του Delta Lake και του Apache Hudi, με έμφαση τόσο στην επεξεργασία κατά παρτίδες (batch processing) όσο και στην επεξεργασία ροής (stream processing). Μέσω της διαδικασίας αξιολόγησης, θα συγκρίνουμε τα Delta Lake και Hudi έναντι τυπικών υλοποιήσεων λιμνών δεδομένων, οι οποίες αποτελούνται από ένα απλό επίπεδο αποθήκευσης το οποίο επερωτάται από μια μηχανή ανάλυσης, στην περίπτωσή μας, το HDFS και το Apache Spark. Τα lakehouses αποτελούν επεκτάσεις των λιμνών δεδομένων, συνεπώς αξιοποιούν τα θετικά χαρακτηριστικά τους, ενώ ταυτόχρονα εισάγουν νέες δυνατότητες, όπως ACID συναλλαγές, επιβολή και εξέλιξη σχήματος (schema enforcement and evolution), καθώς και μηχανισμούς διακυβέρνησης δεδομένων, με σκοπό την αντιμετώπιση των υφιστάμενων προβλημάτων των λιμνών δεδομένων. Παράλληλα, ενσωματώνουν βελτιστοποιήσεις όπως ταξινόμηση, data skipping και partition pruning, με σκοπό την περαιτέρω βελτίωση τους. Στην παρούσα μελέτη, παρουσιάζονται τα παραπάνω χαρακτηριστικά και, μέσω μετρήσεων απόδοσης, αξιολογείται το κατά πόσο βελτιώνουν την απόδοση ή, στην περίπτωση χειρότερης απόδοσης, εάν οι πρόσθετες λειτουργικότητες δικαιολογούν τη χρήση των lakehouses.

Λέξεις κλειδιά

Μεγάλα Δεδομένα, Λίμνες Δεδομένων, Επεξεργασία κατά Παρτίδες, Επεξεργασία Ροής, Delta Lake, Apache Hudi.

Abstract

As organisations increasingly adopt lakehouse architectures to support big data analytics, understanding the performance trade-offs of utilising enhanced storage layers instead of standard data lake architectures is essential. This masters dissertation aims to present a comprehensive performance evaluation of two leading data lakehouse solutions, Delta Lake and Apache Hudi, focusing on both batch and stream processing workloads. Through the benchmarking process, we compare Delta Lake and Hudi against standard data lake implementations, which consist of a simple storage layer queried by an analytics engine, in this case, HDFS and Apache Spark. Being built on top of data lakes, lakehouses leverage their strengths, while simultaneously, introducing new features, such as ACID transactions, schema enforcement, schema evolution and data governance mechanisms, to address the issues data lakes face. Additionally, they introduce optimisations, such as indexing, data skipping, and partition pruning, to further improve them. Throughout this thesis, we present these features and through benchmarks, evaluate how they improve performance and whether the added functionalities justify the use of lakehouses, even in cases where they may underperform.

Key words

Big Data, Data Lakes, Batch Processing, Stream Processing, Delta Lake, Apache Hudi.

Περιεχόμενα (Contents)

Κατάλογος Σχημάτων (List of Figures)

Κατάλογος Αποσπασμάτων Κώδικα (List of Code Listings)

Ευρετήριο Συντομεύσεων (Abbreviation Index)

Abbreviation	Meaning
AI	Artificial Intelligence
BI	Business Intelligence
CSV	Comma Separated Values
DFS	Distributed File System
ELT	Extract Load Transform
ETL	Extract Transform Load
HDFS	Hadoop Distributed File System
IoT	Internet of Things
ML	Machine Learning
MVCC	Multi Version Concurrency Control
OLAP	Online Analytical Processing
POSIX	Portable Operating System Interface
RDBMS	Relational Database Management System
SUT	System Under Test

Κεφάλαιο 0

Εκτενής Περίληψη

0.1 Λίμνες Δεδομένων

Η ανάγκη για την αποδοτική διαχείριση μεγάλων δεδομένων οδήγησε στην σύσταση των λιμνών δεδομένων, καθώς δεν ήταν πλέον δυνατό να καλυφθεί από τα προγενέστερα συστήματα. Οι λίμνες δεδομένων είναι αποθετήρια δεδομένων, βασισμένα σε συστήματα αποθήκευσης χαμηλού κόστους, τα οποία υλοποιούνται με αρχιτεκτονική επιβολής σχήματος κατά την ανάγνωση (schema-on-read). Αποθηκεύουν ανεπεξέργαστα δεδομένα σε ανοιχτές μορφές αρχείων, όπως CSV, Apache Parquet μεταξύ άλλων [Armb21]. Οι δύο κυρίαρχες υλοποιήσεις λιμνών δεδομένων είναι με βάση την συμβατική αρχιτεκτονική και την αρχιτεκτονική δύο επιπέδων.

Η συμβατική υλοποίηση αποτελείται είτε από κάποιο κατανεμημένο σύστημα αποθήκευσης, όπως το HDFS [Data25], είτε από κάποια υποδομή αποθήκευσης στο νέφος, σε συνδυασμό με κάποια μηχανή ανάλυσης δεδομένων, όπως το Apache Spark [Amaz25]. Αυτή η αρχιτεκτονική προσφέρει ευελιξία στις λίμνες δεδομένων όσων αφορά την κλιμακωσιμότητα, καθώς μπορεί να εφαρμοστεί οριζόντια επέκταση (scale out) με την προσθήκη περισσότερων κόμβων, ενώ ταυτόχρονα είναι πιο συμφέρουσα ως προς το κόστος σε σχέση με την χρήση ενός κεντρικού διακομιστή. Η χρήση ανοιχτών μορφών αρχείων συνεπάγεται γρήγορη αποθήκευση νέων δεδομένων, με ευελιξία οργάνωσης κατά περίπτωση, καθώς και ευελιξία στην χρήση σε συνδυασμό με διαφορετικές μηχανές ανάλυσης και συστήματα. Η ευκολία σύζευξης καθιστά την μεταφορά δεδομένων σε άλλη πλατφόρμα πολύ πιο εύκολη σε σύγκριση με προγενέστερες λύσεις, ενώ η διαχείριση των μετασχηματισμών δεδομένων από μηχανές ανάλυσης μειώνει τις ασυνέπειες στις μορφές δεδομένων εντός της λίμνης.

Στην αρχιτεκτονική δύο επιπέδων, τα δεδομένα αρχικά αποθηκεύονται στην λίμνη δεδομένων και στην συνέχεια ένα μικρό υποσύνολο των δεδομένων μετασχηματίζεται και αποθηκεύεται σε μία αποθήκη δεδομένων για χρήση στην υποστήριξη λήψης αποφάσεων [Armb21]. Αυτή η αρχιτεκτονική είναι πλέον κυρίαρχη [Armb21], καθώς παρέχει την ευελιξία των λιμνών δεδομένων σε συνδυασμό με την ποιότητα και διακυβέρνηση δεδομένων που εγγυούνται οι αποθήκες δεδομένων. Το υποσύνολο των δεδομένων που περιέχονται στην αποθήκη, αποτελείται από υψηλής ποιότητας επεξεργασμένα δεδομένα τα οποία είναι έτοιμα για χρήση από εφαρμογές, ενώ μέσω των ώριμων μηχανισμών προστασίας που συναντώνται στις αποθήκες, εγγυάται η ασφάλεια των δεδομένων.

Οι παραπάνω αρχιτεκτονικές έδωσαν λύσεις σε πολλά από τα προβλήματα που αντιμετωπίζονταν στον τομέα της επεξεργασίας δεδομένων, ωστόσο οι παραπάνω αρχιτεκτονικές συνοδεύονται από τις εξής παθογένειες:

- Ανάγκη για υλοποίηση περίπλοκων και ακριβών αλυσίδων επεξεργασίας για την προετοιμασία δεδομένων, οι οποίες εισάγουν νέα σφάλματα.
- Έλλειψη χαρακτηριστικών σχεσιακών βάσεων δεδομένων, όπως μεταδεδομένα, ACID συναλλαγές, ταξινόμηση, caching, βελτιστοποίηση ερωτημάτων.
- Ακαταλληλότητα για περίπλοκες εφαρμογές, όπως η μηχανική μάθηση.
- Δυσκολία διατήρησης της συνέπειας μεταξύ της λίμνης και της αποθήκης.
- Δυσκολία μεταφοράς των δεδομένων της αποθήκης σε άλλη πλατφόρμα.

0.2 Data Lakehouses

Ως απάντηση στα παραπάνω προβλήματα συστάθηκε μία νέα αρχιτεκτονική, η αρχιτεκτονική data lakehouse. Τα lakehouses είναι μια υβριδική αρχιτεκτονική διαχείρισης δεδομένων που συνδυάζει τα πλεονεκτήματα των λιμνών δεδομένων και των αποθηκών δεδομένων. Αποτελούν πρόσθετα επίπεδα τα οποία συνδέονται άμεσα με το επίπεδο αποθήκευσης και υλοποιούν δικά τους μοντέλα αποθήκευσης δεδομένων, βασισμένα σε ανοικτές μορφές αρχείων. Το Delta Lake [IBM25] και το Apache Hudi [Drem25] αποτελούν δύο διαδεδομένα συστήματα lakehouse, τα οποία θα συγκρίνουμε σε αυτή την διπλωματική εργασία. Η αρχιτεκτονική lakehouse αντιμετωπίζει τα προβλήματα των σύγχρονων αρχιτεκτονικών λιμνών δεδομένων εισάγοντας τα παρακάτω χαρακτηριστικά:

- Επίπεδο μεταδεδομένων.
  - ACID συναλλαγές.
  - Ασφαλείς ταυτόχρονες αναγνώσεις και εγγραφές.
  - Επιβολή σχήματος.
  - Εξέλιξη σχήματος.
  - Έλεγχος πρόσβασης (audit logging).
  - Συλλογή βοηθητικών δεδομένων.
- Ανοιχτές μορφές αρχείων.
  - Εύκολη μεταφορά σε διαφορετικές πλατφόρμες.
  - Κλιμακωσιμότητα.
- Βελτιστοποιήσεις SQL ερωτημάτων.
  - Caching.
  - Ταξινόμηση (indexing).
  - Data skipping.
  - Partition pruning.

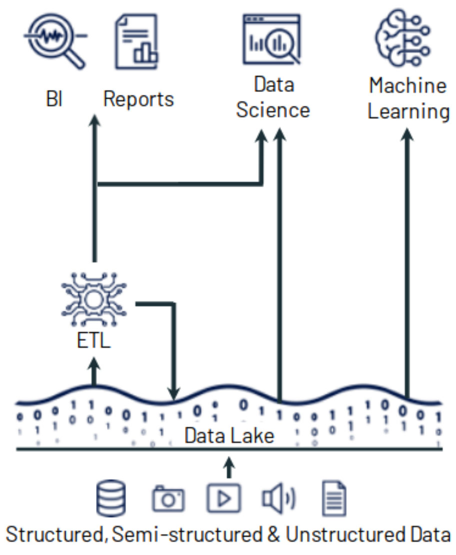
0.3 Πειραματική Διάταξη

Για τα πειράματα επεξεργασίας κατά παρτίδες χρησιμοποιούμε 6 εργάτες στην πλατφόρμα okeanos-knossos, του GRNET, με 4 vCPUs, 8 GB μνήμης RAM και δίσκο μεγέθους 30 GB ανά εργάτη. Για την επεξεργασία ροής χρησιμοποιούμε 6 εργάτες στην πλατφόρμα AWS, υλοποιημένους με εικόνες c5.xlarge, 4 vCPUs, 8 GB μνήμης RAM και δίσκο μεγέθους 200 GB.

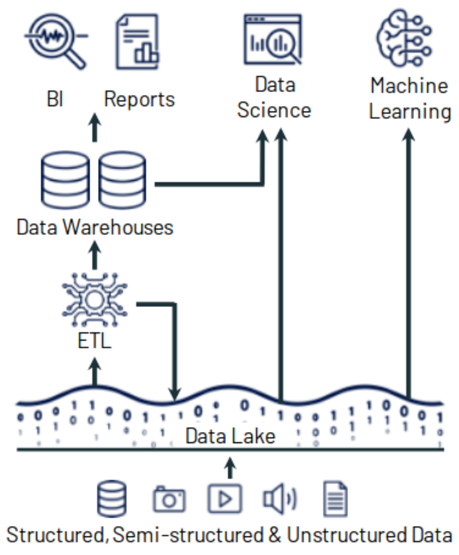
Ως επίπεδο αποθήκευσης, χρησιμοποιούμε το HDFS 3.4.0[1] και ως μηχανή ανάλυσης το Spark 3.5.1[2], τα οποία επιτρέπουν να συγκρίνουμε τα δύο συστήματα με δίκαιο τρόπο, καθώς υποστηρίζουν και τα δύο χωρίς όμως να είναι βελτιστοποιημένα για κανένα. Χρησιμοποιούμε τις προεπιλεγμένες ρυθμίσεις του Spark, με εξαίρεση την αύξηση της μνήμης του driver σε 1 GB και της μνήμης των executors σε 6 GB για να αποφύγουμε out-of-memory σφάλματα.

Όσων αφορά τα lakehouses, χρησιμοποιούμε το Delta Lake 3.1.0[3], με τις προεπιλεγμένες ρυθμίσεις, και δημιουργούμε μία προσαρμοσμένη έκδοση του Apache Hudi 0.14.1[4], η οποία περιλαμβάνει ενημερωμένη εικόνα του HBase συμβατή με το Hadoop 3. Όπως και στην περίπτωση του Delta, χρησιμοποιούμε τις προεπιλεγμένες ρυθμίσεις.

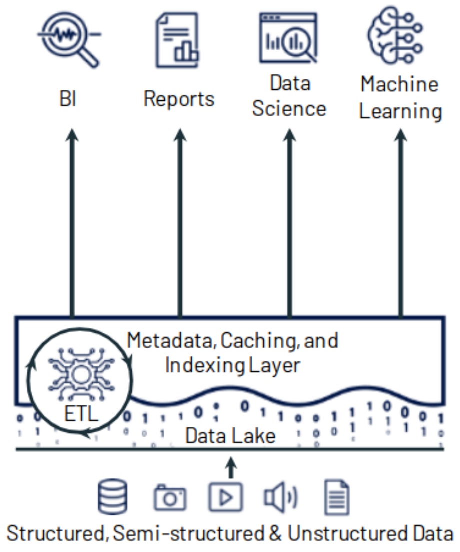
[1] HDFS 3.4.0: https://hadoop.apache.org/docs/r3.4.0/
[2] Apache Spark 3.5.1: https://spark.apache.org/docs/3.5.1/
[3] Delta Lake 3.1.0: https://docs.delta.io/3.1.0/index.html
[4] Apache Hudi 0.14.1: https://hudi.apache.org/docs/0.14.1/overview



(a) Συμβατική αρχιτεκτονική λίμνης δεδομένων.



(b) Αρχιτεκτονική δύο επιπέδων λίμνης δεδομένων.


BI
Reports
Data Science
Machine Learning
Metadata, Caching, and Indexing Layer
ETL
Data Lake
Structured, Semi-structured & Unstructured Data

(c) Αρχιτεκτονική data lakehouse.

Σχήμα 0.1: Αρχιτεκτονικές συστημάτων διαχείρισης μεγάλων δεδομένων. Αναπαραγωγή και επεξεργασία από [Armb21].

0.4 Επεξεργασία κατά Παρτίδες (Batch Processing)

Για τα πειράματα επεξεργασίας κατά παρτίδες χρησιμοποιούμε το σύνολο δεδομένων TPC-DS, 6 απλά ερωτήματα και 10 περίπλοκα ερωτήματα από το TPC-DS. Τα απλά ερωτήματα έχουν σκοπό την αξιολόγηση των δύο συστημάτων για συγκεκριμένες λειτουργίες, ενώ τα περίπλοκα αποσκοπούν στην αξιολόγηση της ικανότητας κάθε συστήματος να δημιουργήσει αποδοτικό πλάνο εκτέλεσης του ερωτήματος. Ως κύρια μονάδα μέτρησης για τα πειράματα χρησιμοποιούμε τον χρόνο εκτέλεσης.

Η πειραματική διαδικασία αποτελείται από 4 φάσεις. Στην πρώτη φάση, εκτελούμε τα ερωτήματα χρησιμοποιώντας την διαμόρφωση αναφοράς η οποία αποτελείται από 2 εργάτες, ο κάθε ένας με 4 vCPUs, 8 GB μνήμης RAM και σύνολο δεδομένων μεγέθους 30 GB. Στην δεύτερη φάση, επιλέγουμε ένα αντιπροσωπευτικό υποσύνολο ερωτημάτων και αυξάνουμε τους εργάτες από 2 σε 4 και 6, ώστε να αξιολογήσουμε την κλιμακωσιμότητα του κάθε συστήματος. Στην τρίτη φάση, χρησιμοποιώντας τον ιδανικό αριθμό εργατών, εκτελούμε τα πειράματα αυξάνοντας το μέγεθος του σύνολου δεδομένων από 30 GB σε 60 GB και 90 GB, ώστε να αξιολογήσουμε την απόδοση για αυξανόμενα σύνολα. Στην τέταρτη και τελευταία φάση, διαμερίζουμε τους πίνακες και εφαρμόζουμε ταξινόμηση Z-Order, ώστε να αξιολογήσουμε τις βελτιστοποιήσεις που υποστηρίζουν τα δύο συστήματα.

Από τα αποτελέσματα της πρώτης φάσης, συμπεραίνουμε ότι για απλά ερωτήματα, το Delta έχει καλύτερη επίδοση από το Hudi, με το Spark να έχει ελαφρώς καλύτερη επίδοση και από τα δύο. Στα περίπλοκα ερωτήματα το Spark πάλι έχει ελαφρώς καλύτερη επίδοση, με το Hudi να αποδίδει καλύτερα σε ερωτήματα με πολλές ενώσεις πινάκων και το Delta σε ερωτήματα με εκτεταμένο φιλτράρισμα και aggregations. Από την δεύτερη φάση, συμπεραίνουμε ότι, ενώ και τα 3 συστήματα επωφελούνται από την οριζόντια κλιμάκωση, το Delta και το Hudi επωφελούνται σε μεγαλύτερο βαθμό, με καλύτερες επιδόσεις από το Spark σε κάποιες περιπτώσεις. Στην τρίτη φάση, όπως είναι αναμενόμενο, παρατηρούμε υψηλότερους χρόνους εκτέλεσης για όλα τα ερωτήματα καθώς αυξάνεται το μέγεθος των δεδομένων, ωστόσο, το Delta και το Hudi, παρουσιάζουν μικρότερες αποκλίσεις από το Spark καθώς αυξάνεται το μέγεθος. Τέλος, στην τέταρτη φάση, ο διαμερισμός και η ταξινόμηση των δεδομένων επιφέρουν αισθητές βελτιώσεις, για τα απλά αλλά και τα περίπλοκα ερωτήματα, με τα Delta και Hudi να έχουν καλύτερη επίδοση σε σύγκριση με το Spark σε πολλές περιπτώσεις.

0.5 Επεξεργασία Ροής (Stream Processing)

Για τα πειράματα επεξεργασίας ροής χρησιμοποιούμε δεδομένα μετοχών από το εθνικό χρηματιστήριο της Ινδίας. Οργανώνουμε τα δεδομένα χρησιμοποιώντας την αρχιτεκτονική medallion, όπου το bronze επίπεδο απαρτίζεται από ανεπεξέργαστα δεδομένα, το silver επίπεδο από δεδομένα στα οποία έχει προστεθεί μία παράγωγη στήλη και τέλος, το gold επίπεδο, το οποίο απαρτίζεται από δεδομένα που προέρχονται από την εκτέλεση κάποιου ερωτήματος. Συνολικά εκτελούμε πειράματα με 7 ερωτήματα, 2 ερωτήματα φιλτραρίσματος και 5 windowed aggregation ερωτήματα. Ως κύρια μονάδα μέτρησης για τα πειράματα χρησιμοποιούμε τον ρυθμό διεκπεραίωσης (throughput).

Η πειραματική διαδικασία ξεκινά με μερικά πρωταρχικά πειράματα για τον καθορισμό του βιώσιμου ρυθμού διεκπεραίωσης (sustainable throughput). Ο βιώσιμος ρυθμός διεκπεραίωσης είναι ο μέγιστος ρυθμός με τον οποίο μπορεί να επεξεργάζεται δεδομένα το σύστημα χωρίς να υπάρχει ουρά ανεπεξέργαστων δεδομένων. Βρίσκουμε ότι είναι 50,000 σειρές ανά δευτερόλεπτο και 100,000 για τα ερωτήματα aggregation και φιλτραρίσματος αντίστοιχα. Χρησιμοποιώντας 6 εργάτες, ο καθένας με 4 vCPUs και 8 GB μνήμης RAM, εκτελούμε τα ερωτήματα με ρυθμό εισόδου δεδομένων των βιώσιμο ρυθμό διεκπεραίωσης και επίσης αυξάνοντας και μειώνοντας τον ρυθμό κατά 10,000 σειρές ανά δευτερόλεπτο, ώστε να αξιολογήσουμε τα συστήματα όταν λειτουργούν στον μέγιστο ρυθμό, καθώς και κάτω ή πάνω από αυτόν.

Από τα πειράματα συμπεραίνουμε ότι στο bronze επίπεδο, το Spark έχει καλύτερη επίδοση καθώς το Hudi και το Delta συλλέγουν μεταδεδομένα κατά την είσοδο των δεδομένων. Στο επίπεδο silver παρατηρούμε και πάλι καλύτερες επιδόσεις από το Spark, με μικρότερες αποκλίσεις όμως, καθώς τα δύο lakehouses συλλέγουν μεταδεδομένα μόνο για την νέα στήλη. Για τα ερωτήματα φιλτραρίσματος και τα τρία συστήματα έχουν παρόμοια επίδοση με το Spark να είναι ελαφρώς καλύτερο. Για μεγαλύτερους ρυθμούς εισόδου δεδομένων, το Delta παρουσιάζει καλύτερη επίδοση, ενώ το Hudi λειτουργεί καλύτερα για τον βιώσιμο ρυθμό διεκπεραίωσης. Για τα aggregation ερωτήματα, παρατηρούμε ότι το Spark παρουσιάζει ελαφρώς καλύτερη επίδοση, με εξαίρεση το ερώτημα για την εύρεση ελαχίστου και το ερώτημα με πολλά aggregations, όπου για μεγαλύτερους ρυθμούς εισόδου, το Delta και το Hudi έχουν καλύτερες επιδόσεις.

0.6 Συμπεράσματα

Η αρχιτεκτονική lakehouse εισάγει πολλά νέα χαρακτηριστικά και βελτιστοποιήσεις με σκοπό την αντιμετώπιση των κοινών προβλημάτων των λιμνών δεδομένων. Κατά τα πειράματα επεξεργασίας κατά παρτίδες, παρατηρούμε ότι το Delta και το Hudi επιδεικνύουν συγκρίσιμες και κατά περιπτώσεις καλύτερες επιδόσεις σε σχέση με την παραδοσιακή λίμνη δεδομένων, ενώ στα πειράματα επεξεργασίας ροής, παρά το γεγονός ότι το Spark παρουσιάζει καλύτερη επίδοση κατά την είσοδο των δεδομένων, και τα τρία συστήματα παρουσιάζουν συγκρίσιμη επίδοση στα ερωτήματα. Συμπεραίνοντας, το Delta και το Hudi επιφέρουν εντυπωσιακά αποτελέσματα καθώς επιδεικνύουν συγκρίσιμη ή ακόμη και καλύτερη επίδοση σε σχέση με τις λίμνες δεδομένων. Σε κάθε περίπτωση, το μεγάλο εύρος χαρακτηριστικών που εισάγουν δικαιολογεί την χρήση τους, ακόμη και σε περιπτώσεις όπου παρουσιάζουν ελαφρώς χειρότερη επίδοση.

Κείμενο στα αγγλικά

Chapter 1

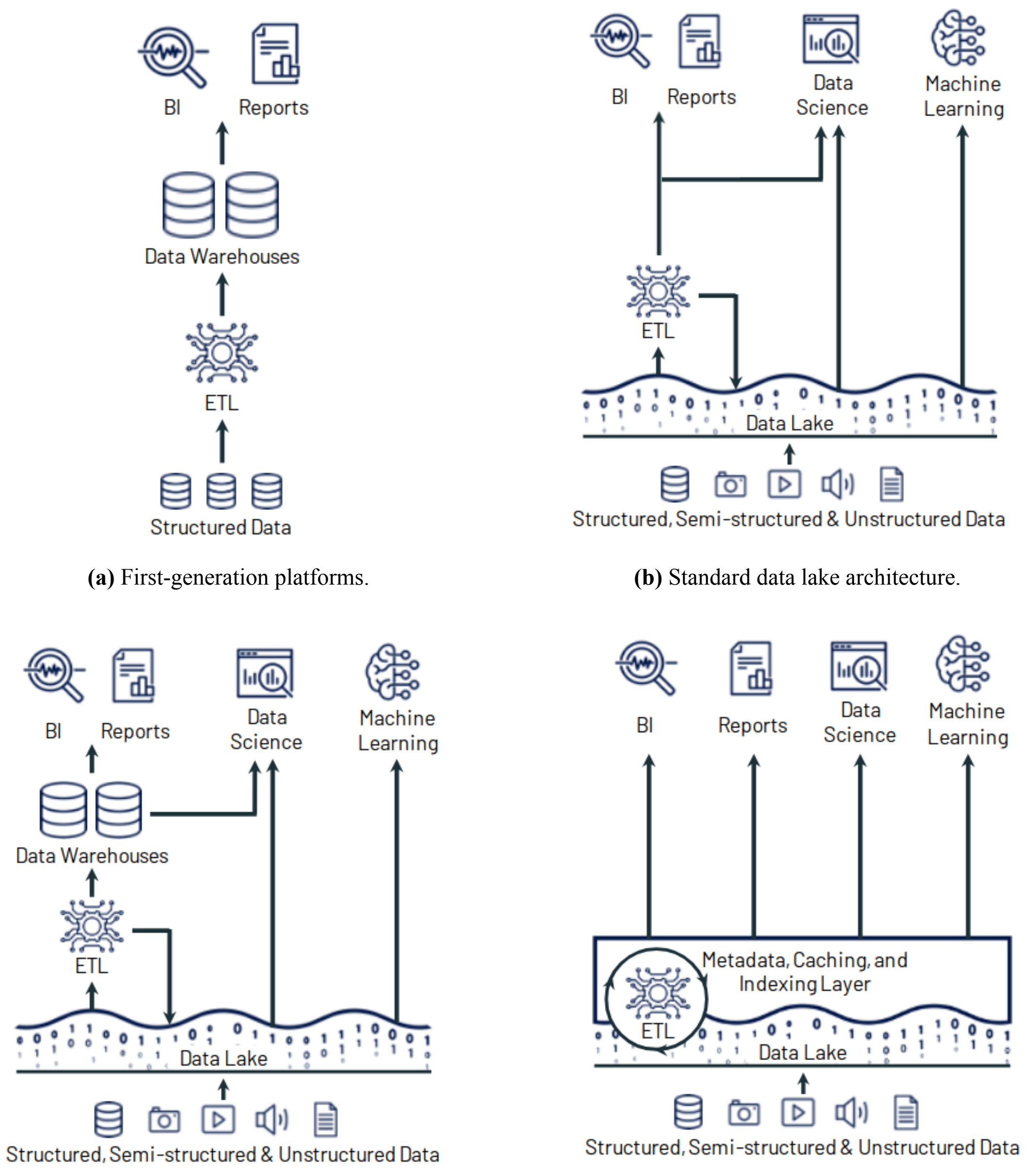
Introduction

The demand for large scale data management solutions has long been present and has only been exacerbated by the exponential growth of the rate at which data is generated, prompted by digital technology advancements, such as connectivity, mobility, the IoT, and AI. The accumulation of massive volumes of structured and unstructured data led many enterprises to outgrow traditional data management systems, such as RDBMSs, which cannot efficiently store or facilitate the processing of data at this scale. Thus, new solutions were proposed and are still being proposed every day as we continue to navigate the big data age.

Data warehousing was born out of the need to provide analytical insights by collecting data from operational databases into centralized warehouses, which then could be used for decision support and BI. Data in these warehouses would be extracted from multiple sources, transformed and loaded into the warehouse under a unified schema [Silb11]. These first generation systems faced problems, such as rising costs, as datasets grew and became increasingly varied and unstructured. They were no longer able to efficiently store and query the data. To remedy these problems, the second generation of platforms was introduced, data lakes [Armb21].

Data lakes are big data repositories, based on low-cost storage, that utilise schema-on-read architecture and hold raw data in generic and open file formats, such as CSV, Apache Parquet etc. While offloading raw data into lakes solved a few of the challenges that arose with data warehouses, the problem of data quality and governance was deferred downstream. The need for a system that can handle large amounts of structured and unstructured data from various sources, while also preserving data quality and providing a robust data governance structure led to the creation of the current generation of data lake platforms, the two-tier architecture [Armb21].

In this architecture, data is first loaded into the lake and then a subset of it is ETLed into a downstream data warehouse, to be used for important decision support and BI applications. The use of open formats also made the data directly accessible to a wide range of other analytics engines, namely query engines and ML systems [Armb21]. Data in this architecture is often stored according to the medallion architecture, a widely used ETL framework which we will utilise for the stream processing benchmarks and discuss in detail in chapter 4. While this design solved the before mentioned problems, it also introduced new ones. As a result of data first being ETLed into lakes, and then ETLed again into warehouses, added complexity, delays, and new failure modes emerge. Moreover, enterprise use cases have grown to be more and more complex, including advanced analytics such as machine learning, for which neither data lakes nor warehouses are ideal [Armb21].


BI
Reports
Data Warehouses
ETL
Structured Data
Data Science
Machine Learning
Data Lake
Structured, Semi-structured & Unstructured Data
Metadata, Caching, and Indexing Layer

(a) First-generation platforms.

(b) Standard data lake architecture.

(c) Two-tier data lake architecture.

(d) Data lakehouse architecture.

Figure 1.1: Evolution of data management platform architectures to the standard data lake, the two-tier model (a-c) and the data lakehouse model (d). Reproduced and edited from [Armb21].

1.1 Use of Data Lakes in Organisations

In this section we dive deeper into how data lakes are implemented in organisations to manage big data. We discuss the characteristics of different implementations of data lakes, including the two-tier architecture, and how they resolve many of the challenges faced by existing systems as well as the new problems they create that led to the proposal of a new solution: Data Lakehouses.

1.1.1 The Standard Data Lake

The most rudimentary implementation of a data lake involves using either a physical distributed storage solution, such as an HDFS [Data25] cluster, or a cloud store, such as Amazon S3[1], Google Cloud Storage[2], Microsoft Azure Data Lake[3] etc, paired with analytics engines.

The underlying storage layers of data lakes are built on distributed storage solutions, unlike traditional systems such as data warehouses, which rely on centralised servers. This architecture makes data lakes significantly more flexible in terms of scalability, as they can scale out, an approach that is more cost effective and less limited than scaling up, which is unavoidable when using centralised servers.

Data lake storage platforms act as data agnostic repositories where data can be quickly stored without any specific organizational structure or organised as seen fit for every use case. Data is accessed through a schema-on-read approach, therefore, only the analytics engines requires new code. This minimizes data format inconsistencies, as all data transformations occur within the analytics layer. This storage scheme also makes data lakes platform independent, allowing any analytics engine to process the data with minimal setup. Integration is only necessary to grant the engine access to the data lake and no additional work is required to read the data since it is stored using open file formats.

The platform agnostic nature of data lakes also renders migration to different technologies easy. As mentioned before, using new analytics engines, depending on the organisations needs, requires minimal integration and data is stored in open file formats, meaning it can be very easily moved to a different platform without the need to reformat the data.

1.1.2 The Two-Tier Architecture

In this architecture, the data lake replaces the integration layer of the data warehouse architecture and serves as the single point of truth [Herd20]. More specifically, instead of gathering and integrating data from multiple sources into a data warehouse under a unified schema, both structured and unstructured data, is first stored in the data lake and a small subset of it is then ETLed downstream to the data warehouse, where it can be used for faster and more straightforward BI and reports.

The two-tier data lake architecture is now dominant in the industry, being used by virtually all Fortune 500 enterprises [Armb21], as it manages to combine most of the benefits and flexibility of a data lake and the data quality and governance ensured by a data warehouse, for a subset of data used for BI and reports. The rigid structure of a warehouse ensures that, the small subset of data within the lake that is conducive to BI and business decisions, contains consistent, high quality, cleansed data. Besides more efficient BI, data warehouse security mechanisms are more mature compared to those of the relatively new data lakes, and thus a data warehouse provides an extra layer of protection between the underlying data lake and external analytics engines.

1.1.3 The Problems of Current Data Lake Solutions

Current data lake solutions are based on either the standard data lake or some combination of a data lake and a data warehouse downstream, with the two-tier architecture being one of the most widely used examples of this approach. Although data lakes represent a significant advancement toward robust big data management systems, they still face several challenges and lack essential functionalities necessary for the efficient storage and utilization of data. Specifically, modern data lake solutions suffer from the following limitations:

[1] Amazon S3: https://aws.amazon.com/s3/, Accessed: 2025-05-13

[2] Google Cloud Storage: https://cloud.google.com/storage, Accessed: 2025-05-13

[3] Microsoft Azure Data Lake: https://azure.microsoft.com/en-us/products/storage/data-lake-storage, Accessed: 2025-05-13

The Problems of the Standard Data Lake

The standard data lake faces two main problems. The first issue is that in modern data management systems, data is predominantly consumed by advanced analytics applications, such as machine learning platforms, however current data lake solutions lack the features necessary to facilitate these use cases. Schema-less data stored in a lake needs to be cleaned, labelled and organised before being used by downstream analytics and ML engines, a problem typically addressed by introducing additional ETL pipelines. Therefore, to build efficient data lakes, the lake architecture requires a redesign of both storage and functionality [Hai23].

Metadata is essential to improve the performance of analytical workloads and ML models, in terms of model accuracy and efficiency. In RDBMSs, it is crucial for storing constraints, table relationships, data types, etc. Utilising metadata is also crucial for model training as primary key-foreign key relationships, join dependencies [Chen17] and functional dependencies [Kham17] can significantly reduce training time. ML datasets often include complex data types such as images, audio, and video. When handling such datasets, extracting metadata from embeddings is essential, as it enables more efficient organisation and processing. Besides the need to provision embeddings for the data itself, it is also essential to collect metadata throughout the lifecycle of ML models. This lifecycle includes multiple stages, such as model training, parameter tuning, and evaluation, during which metadata must be extracted and modelled. However, data lakes inherently lack the ability to collect and store this metadata, making ML workflows inefficient at best and non-functional at worst [Hai23].

To address all of these issues, various ETL pipelines involving multiple systems are often built based on the use case. However, this workaround introduces significant complexity as it requires new code, intricate data pipelines to link the various platforms and familiarity with a wide range of tools and systems, making it difficult to maintain [Makr21].

The second problem is the lack of traditional key RDBMS features such as transaction management, indexing, and caching, which makes data lakes less suitable and efficient for complex analytical workloads. Besides lacking metadata, relational data stored in lakes do not adhere to ACID properties, lack transaction support and offer limited query optimisation. The current workaround is to add a data warehouse downstream. However, maintaining both a data lake and a data warehouse, and ensuring consistency between them, introduces significant overhead and architectural complexity [Hai23].

The Problems of the Two-Tier Architecture

As mentioned above, a system consisting of a data lake and a downstream warehouse is the current workaround for the lack of metadata and traditional RDBMS features in data lakes. This solution is architecturally complex and increases the cost of ownership significantly as it involves maintaining continuous ETL pipelines and incurring additional storage costs for duplicate data stored in both the lake and the warehouse. Additionally, commercial warehouses often lock data into proprietary formats, making migration to other platforms more difficult and costly [Armb21].

Besides the increased maintenance cost, this solution also introduces reliability and data staleness issues. The flow of information within such a system dictates that the data lake serves as the source for the warehouse. Updates, deletes or inserts are not directly available to the warehouse and thus, continuous effort is required to ETL data between the two systems, making keeping them consistent complex and costly. When dealing with large volumes of data, loading into the warehouse may take days, causing the data in the warehouse to be stale in comparison to the data in the lake. The reliability of the platform is also diminished, not only due to potential failures during each ETL stage but also due to the data quality being compromised by possible bugs, stemming from subtle inconsistencies between the data storing formats in each system [Armb21].

Lastly, despite the presence of warehouse features such as metadata, ACID transactions, data versioning and indexing, support for advanced analytics remains limited. In this architecture, warehouses typically store only a small subset of the data, which is not sufficient for effective machine learning, as ML models require access to large datasets. ML platforms utilise complex, non-SQL code, making it difficult to read data directly from the warehouse. The lack of direct access to the warehouse's proprietary internal formats, results in the need for additional ETL steps, which we have already concluded increase complexity and costs [Armb21].

1.2 Data Lakehouses: A Solution to the Challenges

Data lakes continue to evolve by adding new features and functionalities. However, significant gaps remain, as mentioned in the previous section, that prevent them from fully meeting the demands of today's complex big data landscape. To address these problems, a new data management solution has emerged: the data lakehouse. Data lakehouses are an added layer that is integrated with the storage layer of a data lake. It introduces a new storage format based on open file formats and supported by logs and metadata tables. In this section we will discuss some key features of lakehouses and in the next chapter we will dive deeper into how Delta Lake [IBM25] and Apache Hudi [Drem25], our SUTs, implement these features.

1.2.1 Metadata Layer

Data lakehouses introduce metadata layers over data lake storage, thus raising the abstraction level and enabling the implementation of capabilities such as ACID transactions, schema enforcement and other data quality and governance features [Armb21].

Transaction Support

Data lake storage systems, such as HDFS or cloud stores, provide a low-level object store or filesystem interface where even simple operations, such as updating a table distributed across multiple files, are not atomic and isolated [Armb21]. Data lakehouses introduce ACID properties through metadata, ensuring safe table-level concurrent reads and writes [Mazu23].

Schema Enforcement and Evolution

Similar to the implementation of constraints and table relationships in RDBMSs, lakehouses leverage the metadata layer to implement and enforce a schema for lake data. Combined with the use of open data formats, this also enables schema evolution over time, without having to bear the cost of rewriting an entire table [Mazu23].

Access Control

Finally, metadata layers are a natural place to implement governance features such as access control and audit logging. With data lakes lacking mature safety features, a metadata layer can enforce access control by verifying whether a client is authorized to access a specific table before granting credentials to read raw data from the object store. It also ensures that all access events are reliably logged [Mazu23].

1.2.2 Open File Formats

Lakehouses introduce new table formats through transaction logs and metadata tables, that work on top of open file formats, such as Parquet. This enables lakehouses to introduce additional features without locking data into a proprietary format. Therefore, analytical workloads can be run by different engines including both lakehouse specific and non-specific [Mazu23].

Easy Migration

The use of open data formats enables organisations to move data across platforms without the need for reformatting, while also enabling organisations that already utilise data lakes to easily convert an existing directory of Parquet files into a lakehouse format and reverse. For example, Delta Lake can convert a table with zero copies just by adding a transaction log that starts with an entry that references all the existing files [Armb21].

Scalability

The metadata layer of lakehouses, being built on top of distributed open file formats, enables lakehouses to introduce necessary warehouse features, needed for complex analytical workloads, without having to sacrifice the data lake's ability to efficiently scale out. The transaction logs and metadata tables are also stored using open file formats, thus making every aspect of lakehouses highly scalable.

1.2.3 SQL Performance Optimisations

Warehouses were incorporated in data lake architectures to introduces RDBMS features and improve SQL query performance. Besides transactions, schema enforcement etc, lakehouses also introduce SQL specific optimisations to enhance BI and analytic query performance. These optimisations are introduced through the following features.

Caching

Transactional metadata layers ensure data quality and consistency, thus enabling lakehouse systems to cache files from the object store to faster storage devices, such as SSDs and RAM, on the processing nodes. Utilising transaction logs, it is easy to determine which cached files are still valid for reading. Moreover, data can be cached in a transcoded format that is more efficient for the query engine to run on, mirroring similar optimisations that would be used in traditional warehouse engines [Armb21].

Auxiliary Data

Lakehouses maintain data, which help query optimisation, in auxiliary files they have full control over. Such data include column min-max statistics for each sub file within the same Parquet file used, which enables data skipping and partition pruning optimisations when the base data is clustered by particular columns. Data skipping is the utilisation of column data to avoid reading unnecessary data during query execution, and thus improve performance [Armb21].

Data Layout

Data layout is instrumental to access performance, especially for distributed formats such as Parquet where tables are split between storage nodes. Lakehouses allow data partitioning using individual dimensions, a feature which is also supported by data lakes, as well as introducing partitioning based on space filling curves, such as the Z-order and Hilbert curves, to provide locality across multiple dimensions [Armb21].

1.3 The Aim and Scope of this Thesis

The aim of this thesis is to evaluate the performance of two data lakehouses compared to a traditional data lake. Having already made a case for data lakehouses as a compelling solution in big data management, we are interested in exploring when lakehouses outperform lakes, when they perform similarly, when they underperform and whether the performance trade-off is justified by the added benefits of using a lakehouse. More specifically, we will evaluate two of the leading data lakehouse software solutions, Delta Lake and Apache Hudi. We will compare the performance of Hudi over HDFS queried by Spark [Amaz25], Delta Lake over HDFS queried by Spark and plain HDFS queried by Spark for batch and streaming workloads. Each software product's characteristics and why it was chosen will be discussed in detail in the next chapter.

Based on the claims made by Delta Lake [Aysh24] and Hudi [Hudi24] we expect to see similar or even better performance to a traditional data lake. Both software products have been created to pair well with Spark and HDFS and boast features that, besides adding new functionalities to traditional data lakes, are supposed to improve performance. Therefore, we expect to verify these claims.

The scope of the project is restricted to evaluating the performance of Hudi and Delta Lake, thus we will try to separate the performance fluctuations attributed to Hudi and Delta Lake from the performance fluctuations attributed to the supporting software products, HDFS and Spark, and the performance fluctuations attributed to the hardware. Throughout the paper it will be mentioned whenever these distinctions are made and how they are made. We chose an open source storage solution and query engine, namely HDFS and Spark, to avoid vendor-specific platforms, that may have software specific enhancements, which boost performance.

Chapter 2

Experimental Setup

In this chapter we will discuss the experimental setup used for the benchmarks, both the hardware and the software utilised. We will introduce our SUTs and explain how the lakehouse features we analysed in the previous chapter are implemented internally.

2.1 Hardware

For the batch processing part of this thesis, we perform our experiments with 6 workers on okeanos-knossos, GRNET's cloud service, with 4 vCPUs, 8 GB of RAM and 30 GB of disk space each. For the stream processing part of this work, we utilise 6 workers on AWS c5.xlarge instances, with 4 vCPUs, 8 GB of RAM and 200 GB of disk space each.

2.2 Hadoop Distributed File System

HDFS is an open source, scalable and fault-tolerant DFS specifically designed to store and manage large volumes of data. It partitions data into blocks and distributes them across nodes in a cluster composed of commodity hardware, while also relaxing a few POSIX requirements for storage systems to enable streaming access to data. HDFS is commonly used as a foundational storage layer for data lakes, as it provides reliable and highly available storage, even in the event of hardware failures. It is designed for horizontal scalability, meaning it can scale out efficiently through the addition of more nodes to the cluster. For our experiments, we utilise HDFS 3.4.0[1], chosen not only for its robust features mentioned above but also due to its independence from any specific lakehouse architecture, which makes it a flexible and neutral storage layer suitable for our purposes.

2.3 Apache Spark

Spark is an open source, distributed computing framework designed for fast and large scale data processing. It provides a unified engine capable of handling machine learning, graph processing, batch and streaming workloads. By allowing in-memory computation and optimised execution plans, it significantly enhances performance. Spark is widely used for big data analytics as it supports multiple programming languages, such as Scala, Python, Java and R, and integrates with various storage systems including HDFS and cloud storage systems. We utilise Spark 3.5.1[2], for our experiments because it allows as to compare our lakehouse formats fairly, as it supports both, while not being specially optimised for either. We use the default configuration for Spark, without any manual tuning except increasing the driver memory size to 1 GB and executor memory to 6 GB to avoid out-of-memory errors.

[1] HDFS 3.4.0: https://hadoop.apache.org/docs/r3.4.0/
[2] Apache Spark 3.5.1: https://spark.apache.org/docs/3.5.1/

2.4 Delta Lake

Our first SUT is Delta Lake, an open source lakehouse platform designed to run on top of an existing data lake to improve its reliability, security, and performance. Delta introduces ACID transactions, scalable metadata, unified streaming and batch processing. It addresses common challenges in standard data lake design with features like schema evolution, time travel, and versioned data. It leverages a transaction log to track changes to data over time, in order to maintain data integrity and enable efficient query execution. For our experiments we utilise Delta Lake 3.1.0[3], with the default configuration as it is intended to support a diverse range of workloads without workload specific tuning.

2.5 Apache Hudi

Our second SUT is Apache Hudi, another open source data lakehouse platform, built on a high performance open table format intended to bring database functionality to data lakes. Similar to Delta Lake, it is a unified batch and stream processing engine that introduces features such as schema evolution, ACID transactions and rollbacks. Hudi employs a combination of storage formats and indexing techniques, to enable efficient query execution. For our experiments, we package our own version of Hudi 0.14.1[4] to include a new image of HBase that is compatible with Hadoop 3, since we use HDFS 3.4.0. HBsae is necessary as Hudi uses HFiles as the base format for its metadata table. Similarily to Delta, we use the default configuration.

2.6 Delta Lake and Apache Hudi Design

In this section, we examine the design decisions made for both SUTs: Delta Lake and Apache Hudi. Each lakehouses adopts its own approach to implementing metadata management, data update strategies, transactions atomicity and isolation. An overview of these design choices and their respective implementations in each system is presented in Table 2.1.

	Table Metadata	**Transaction Atomicity**	**Isolation Levels**
Delta Lake	Transaction Log and Metadata Checkpoints	Atomic Log Appends	Serializability, Strict Serializability
Apache Hudi	Transaction Log and Metadata Table	Table-Level Lock	Snapshot Isolation

Table 2.1: Lakehouse system design features.

In the following subsections, we provide a detailed analysis of each design decision, highlighting aspects where Delta and Hudi adopt similar approaches as well as where their implementations diverge.

2.6.1 Transaction Coordination

Both systems implement transactions that span all records within a single table but do not support transactions across multiple tables. They use multi version concurrency control (MVCC) to manage transactions and a metadata structure determines which file versions are associated with a given table. When a transaction begins, it reads this metadata structure to obtain a snapshot of the table and then

[3] Delta Lake 3.1.0: `https://docs.delta.io/3.1.0/index.html`
[4] Apache Hudi 0.14.1: `https://hudi.apache.org/docs/0.14.1/overview`

performs all read operations based on that snapshot. To commit, the transaction atomically updates the metadata structure. Delta Lake depends on the atomic guarantees of the underlying storage system, while Hudi enforces atomicity through table-level locks implemented in ZooKeeper, Hive MetaStore, or DynamoDB [Jain23].

To ensure isolation between transactions, both Delta Lake and Hudi employ optimistic concurrency control. Before committing, each transaction undergoes a validation process to detect conflicts with concurrently committed transactions. Hudi enforces snapshot isolation by confirming that a transaction does not attempt to write to any files that have already been modified by committed transactions that were not in the transaction snapshot. Transactions always read from a snapshot of committed transactions as of the time the transaction began and are permitted to commit only if no conflicting writes have occurred during their execution. Delta also verifies no data read by a transaction is altered by concurrently committed transactions that were not in the transaction snapshot, thus providing serializability, meaning the outcome of concurrent transactions is equivalent to some serial execution of those transactions, though not necessarily in the order of which they appear in the transaction log [Jain23].

2.6.2 Metadata Management

Lakehouse systems leverage the higher data read rates of object storage by storing metadata in files kept alongside the actual data files. Since metadata files are much fewer than data files, listing and reading them, instead of listing data files directly from storage, results in faster query planning times. In the tabular format, utilised by Delta Lake and Hudi, metadata for a lakehouse table is stored in a separate, dedicated table. Hudi uses a specialised metadata table, while Delta uses a transaction log and checkpoints, composed of Parquet and JSON files. Transactions do not write to the metadata table directly, but instead they generate log records, which are periodically compacted into the table using the merge-on-read approach [Jain23].

There are two different metadata access schemes used to gather the necessary data needed to plan queries, single node planing and distributed planning. Delta and Hudi queries are typically planned in a distributed fashion, as a batch job must scan the metadata table to find all files required for a query [Jain23].

2.6.3 Data Update Strategies

Lakehouses employ two strategies for data updates, Copy-On-Write (CoW) and Merge-On-Read (MoR), with differing trade-offs between read and write performance. The CoW strategy locates files containing records that need to be updated and rewrites them to new files with the updated data. This method results in slower writes but offers improved read performance, as data is constantly up to date. The MoR strategy does not rewrite files during updates. It instead logs record-level changes in auxiliary files and defers the reconciliation until query execution. This improves write performance but can negatively impact read performance due to the additional overhead of merging data at query time [Jain23].

The Delta MoR implementation uses auxiliary ”tombstone” files that mark records in data files. At query time, these tombstoned records are filtered out. Record updates are implemented by tombstoning the existing record and writing the updated record into Parquet/ORC files. By contrast, the Hudi implementation of MoR stores all the record-level inserts, deletes and updates in row-based Avro files. On query time, Hudi reconciles these changes while reading data from the Parquet files. It is worth noting that Hudi, by default, deduplicates and sorts ingested data by keys, thus incurring additional write latency even when using MoR [Jain23].

Chapter 3

Batch Processing

Batch processing is one of the two main methods of processing data in modern big data management systems. It refers to the execution of a series of jobs where data is collected, entered, and processed in groups or batches rather than individually. The processing is typically scheduled and may be performed at set intervals, leading to higher latency but suitability for large volumes of data [Benj20]. By dividing large tasks into smaller units, this approach leverages distributed computing to achieve higher performance and scalability. It is usually utilised for advanced analytics and complex queries when accuracy and data volume are more important than latency. In this chapter we perform experiments to evaluate how the addition of Delta or Hudi to a data lake impacts performance for batch processing workloads.

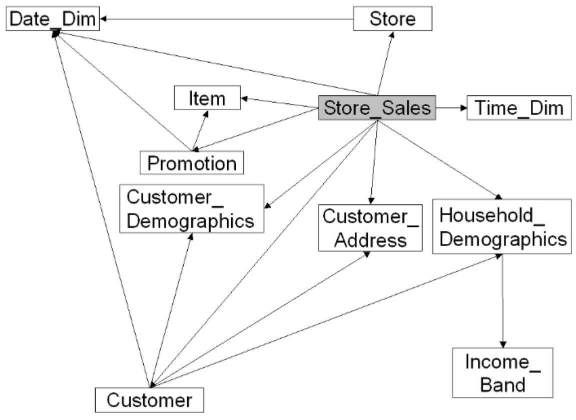
3.1 TPC-DS Benchmark Suite

The TPC-DS benchmark suite is a widely adopted standard for evaluating decision support systems. It is designed to model the operations of a retail decision support system. Specifically, it emulates complex decision support and business intelligence batch workloads for analytical systems that process large amounts of data, typically deployed on big data management platforms, such as data warehouses and data lakes. TPC-DS includes a relational dataset based on a retail business model, comprising multiple fact and dimension tables, along with a diverse set of queries categorized into four broad classes: reporting queries, ad hoc queries, iterative OLAP queries, and data mining queries. The above characteristics make TPC-DS an ideal benchmarking tool for assessing the query performance, scalability and system efficiency of data lake systems under large, analytical batch workloads [Tran21].

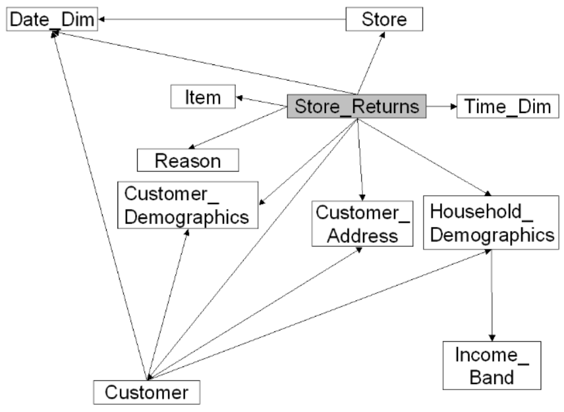
In the following subsection, we analyse the TPC-DS dataset in greater detail. In subsection 3.2.2, we examine the TPC-DS queries more closely and present which were chosen for our experimentation process and the reasons behind the selection.

3.1.1 Dataset

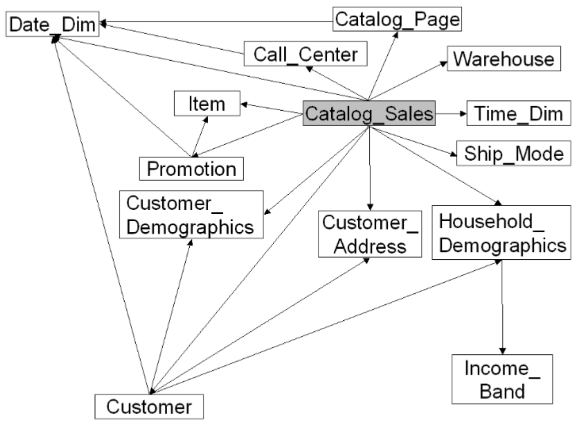
TPC-DS emulates the business model of a large retail company with multiple brick and mortar stores that also sells goods through catalogs and the internet. Along with tables to model the associated sales and returns, it includes simple inventory and promotion systems. It organises data with a snowflake schema that consists of 17 dimension tables and 7 fact tables. Fact tables contain the surrogate keys of the referenced dimension tables along with some quantitative metrics, while dimension tables hold the descriptive information for the related fields in the fact table. Specifically, the schema includes two fact tables modelling product sales and returns for the store, the catalog and the internet and one fact table modelling the inventory for the catalog and internet sales. In Figure 3.1 we can see the ER diagrams that comprise the TPC-DS dataset schema. The fact tables are shaded in gray.



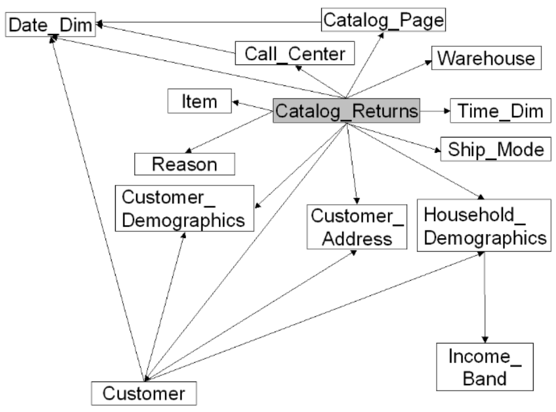
(a) Store Sales ER Diagram



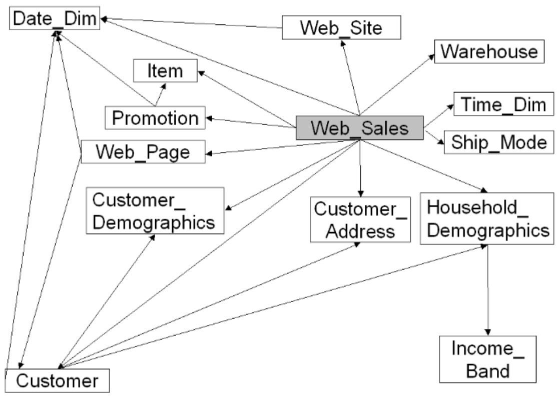
(b) Store Returns ER Diagram



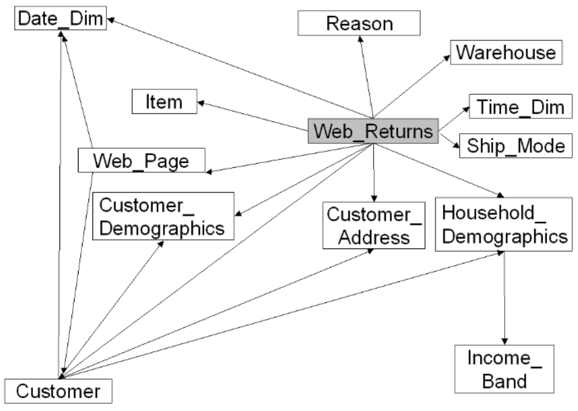
(c) Catalog Sales ER Diagram



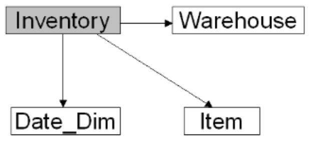
(d) Catalog Returns ER Diagram



(e) Web Sales ER Diagram



(f) Web Returns ER Diagram



(g) Inventory ER Diagram

Figure 3.1: The TPC-DS dataset schema. Reproduced from [Tran21].

3.2 Queries

In this section we present the queries used to evaluate our SUTs. We differentiate between simple and complex queries. We classify basic SQL queries which perform one single operation, such as filtering, aggregation, sorting etc, as simple. They are used to evaluate how effectively each system performs these basic operations. The complex queries are a set of TPC-DS queries we selected that reflect a diverse group of analytics workloads. These queries incorporate a combination of SQL operations and are used to assess how effectively our SUTs formulate query plans and perform in real-world use cases often encountered in decision support and business intelligence systems.

3.2.1 Simple

The first query is a simple select all operation that retrieves all columns from the catalog_sales table, which is the biggest table of the dataset. The purpose of this query is to evaluate the overhead introduced by Delta and Hudi during metadata table access.

```
SELECT * FROM catalog_sales
```

Listing 3.1: Simple Query 1 (Squery 1).

The second query is a select all operation after performing an inner join on the catalog_sales and customer tables. With this query we evaluate how effectively each system performs joins.

```
SELECT * FROM customer INNER JOIN catalog_sales
ON cs_bill_customer_sk = c_customer_sk
```

Listing 3.2: Simple Query 2 (Squery 2).

The third query is an equality filtering query on the catalog_sales table. The purpose of this query is to evaluate how effectively Delta and Hudi leverage the metadata table to data skip.

```
SELECT * FROM catalog_sales WHERE cs_sold_date_sk = 2451806
```

Listing 3.3: Simple Query 3 (Squery 3).

The fourth query is a range filtering query on the inner join of the catalog_sales and customer tables, applying range filters on one column of each table. This query is an extension of the second and third queries and its aim is to evaluate how data skipping when filtering can impact join performance.

```
SELECT * FROM customer INNER JOIN catalog_sales
ON cs_bill_customer_sk = c_customer_sk
WHERE cs_ext_wholesale_cost < 1920 AND c_last_name > 'Lao'
```

Listing 3.4: Simple Query 4 (Squery 4).

The fifth query is an average aggregation query on the catalog_sales table. With this query we examine whether Delta's and Hudi's metadata layers make an impact on aggregation performance.

```
SELECT AVG(cs_ext_wholesale_cost), cs_sold_date_sk
FROM catalog_sales GROUP BY cs_sold_date_sk
```

Listing 3.5: Simple Query 5 (Squery 5).

The sixth and final simple query is a simple sorting query with the purpose of evaluating the effect of column level metadata on performance.

```
SELECT * FROM catalog_sales ORDER BY cs_ext_list_price
```

Listing 3.6: Simple Query 6 (Squery 6).

3.2.2 Complex (TPC-DS Queries)

The TPC-DS queries address a wide array of complex business problems, utilising a variety of access patterns, query phrasings and operators. They are categorised across four broad classes that characterise most decision support queries [Tran21]. We select 10 queries in total, from all four classes, which include diverse query patterns and operators. Following are the four classes in order of size and the queries selected from each class.

Reporting Queries

These queries are executed periodically to produce reports that answer common, pre-defined questions about the financial and operational health of a business. Although they generally tend to be static, minor changes can be made to shift focus to a different date range, geographic location, brand name etc [Tran21]. From this class we select the following queries:

Query 33: What is the monthly sales figure, based on extended price, for a specific month in a specific year, for manufacturers in a specific category in a given time zone. Group sales by manufacturer identifier and sort output by sales amount, by channel, and give total sales.

Query 35: For the groups of customers living in the same state, having the same gender and marital status who have purchased from stores and from either the catalog or the web during a given year, display the state, gender, marital status, count of customers, min, max, average, count distinct of the customer's dependent count, min, max, average, count distinct of the customer's employed dependent count, min, max, average, count distinct of the customer's dependents in college count. Display the ”count of customers” multiple times to emulate a potential reporting tool scenario.

Query 61: Find the ratio of items sold with and without promotions in a given month and year. Only items in certain categories sold to customers living in a specific time zone are considered.

Query 75: For two consecutive years track the sales of items by brand, class and category.

Ad-Hoc Queries

These queries express dynamic and impromptu questions, which answer immediate and specific requests for data. Ad-hoc queries differ from reporting queries in that they are generated in real-time to address specific use cases, without relying on predefined reports [Tran21]. From this class we select the following queries:

Query 28: Calculate the average list price, number of non empty (null) list prices and number of distinct list prices of six different sales buckets of the store sales channel. Each bucket is defined by a range of distinct items and information about list price, coupon amount and wholesale cost.

Query 72: For each item, warehouse and week combination count the number of sales with and without promotion.

Iterative OLAP Queries

OLAP queries retrieve and analyse multidimensional data, in order to provide insights and facilitate BI, through aggregations, dimensions, and hierarchies. They allow for the exploration and analysis of data to discover new and meaningful relationships and trends. While similar to ad-hoc queries they differ in that, contrary to ad-hoc queries which involve a specific request, OLAP queries comprise a sequence of simple and complex queries that express dimension and hierarchy [Tran21]. From this class we select the following queries:

Query 9: What is the ratio between the number of items sold over the internet in the morning (8 to 9am) to the number of items sold in the evening (7 to 8pm) of customers with a specified number of dependents. Consider only websites with a high amount of content.

Query 11: Find customers whose increase in spending was large over the web than in stores this year compared to last year.

Query 64: Find those stores that sold more cross-sales items from one year to another. Cross-sale items are items that are sold over the Internet, by catalog and in store.

Data Mining Queries

These queries involve sifting through large volumes of data to produce data content relationships, in order to predict future trends and behaviours. Data mining queries typically consist of joins and large aggregations that return large data result sets, which allow businesses to make proactive, knowledge driven decisions [Tran21]. From this class we select the following queries:

Query 38: Display count of customers with purchases from all 3 channels in a given year.

3.3 Experimentation Process

Our experimentation process consists of four phases. In the first phase, we execute multiple queries on a baseline system to evaluate the performance of our SUTs based on query type. The baseline system in this phase comprises 2 workers, each with 4 vCPUs, 8 GB of RAM, and a 30 GB dataset. These resources are divided between 2 Spark executors. This is the default Spark configuration as, according to Spark's documentation, the optimal number of CPU cores per executor is approximately five. Based on our experiments, we determined that for our systems the optimal distribution of CPU cores is 4 cores per executor.

In the second phase, we select a representative subset of queries from those used in the first phase and we conduct experiments to evaluate each system's scalability by increasing the number of workers. Using the same baseline configuration, we execute the queries, using 2, 4 and 6 workers to examine the scalability of our SUTs.

In the third phase, utilising the optimal number of workers determined in the previous phase, we execute the same subset of queries on datasets of 30 GB, 60 GB and 90 GB to evaluate the performance of each system with increasing dataset sizes.

In the fourth and final phase, using the optimal number of workers again, we execute a subset of queries after partitioning the tables and applying Z-Ordering in Delta and Hudi to index the data. Through this phase, we evaluate how partitioning and multidimensional indexing, which is only supported in lakehouses, impact query performance.

3.3.1 Benchmark Metrics

The primary metrics used in benchmarking batch processing systems are latency, resource utilisation and the error rate. We mainly focus on latency, as it directly reflects the responsiveness and efficiency of each SUT. Throughout our experiments, we observed that the resource utilisation and error rates across all three systems were largely consistent. Therefore, we will only highlight these metrics in cases where there are significant deviations.

3.3.2 Configuration Changes

The following table summarises the configurations utilised throughout each phase of the benchmarks, as they are mentioned in the experimentation process.

Workers	CPU Cores per Worker	RAM per Worker	Dataset Size
Phase 1: Benchmarks with the baseline configuration.			
2	4 vCPUS	8 GB	30 GB
Phase 2: Benchmarks with increasing number of workers.			
2	4 vCPUS	8 GB	30 GB
4	4 vCPUS	8 GB	30 GB
6	4 vCPUS	8 GB	30 GB
Phase 3: Benchmarks with increasing dataset size.			
6	4 vCPUS	8 GB	30 GB
6	4 vCPUS	8 GB	60 GB
6	4 vCPUS	8 GB	90 GB
Phase 4: Benchmarks with partitioning and Z-Order indexing.			
6	4 vCPUS	8 GB	90 GB

Table 3.1: Configuration changes for each phase of the batch processing benchmarks.

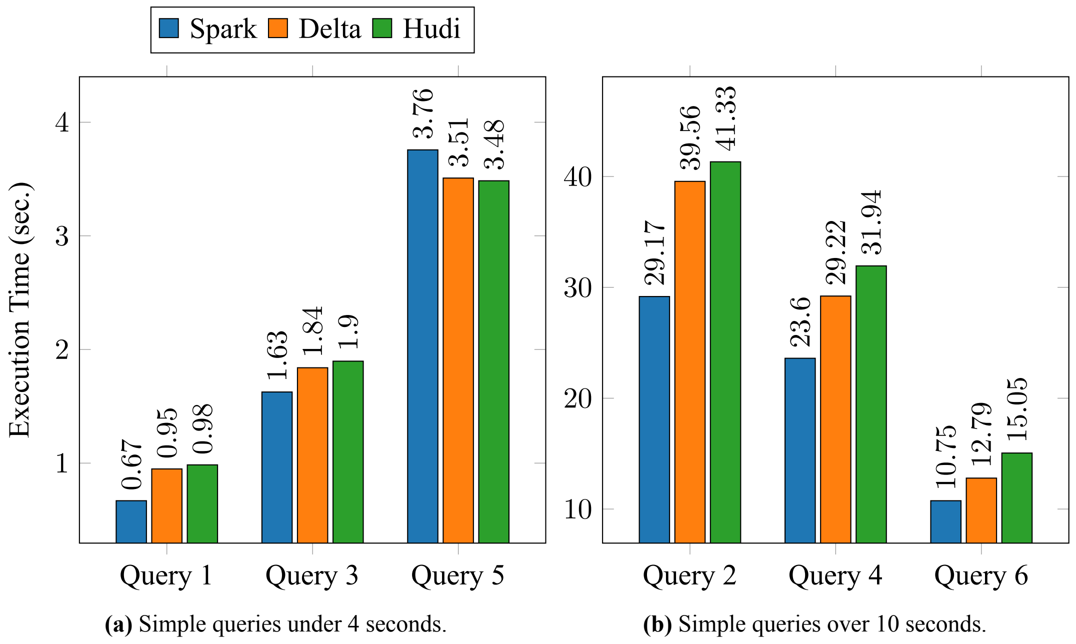
3.4 Results

In this section we will present the results of our benchmarks and discuss them. We present our results in the three phases we already mentioned in the experimentation process.

3.4.1 Phase 1

In phase 1 we execute multiple queries on the baseline system in order to evaluate the performance of our SUTs based on query type. We examine the simple and complex queries separately.

Simple Queries



(a) Simple queries under 4 seconds. **(b)** Simple queries over 10 seconds.

Figure 3.2: Phase 1: Simple query execution times using 2 workers (2 executors) and a 30 GB dataset.

In query 1, we observe that Delta and Hudi exhibit performance overhead when retrieving the entire table. This is expected, as accessing and reading the metadata table introduces additional overhead during the creation of the query plan. The overhead introduced by Delta and Hudi is comparable, with Delta exhibiting an overhead of approximately 42% and Hudi around 46% relative to Spark's execution time.

In query 2, Delta and Hudi's performance is comparable, with Delta edging Hudi out. However, both underperform compared to Spark by approximately 36% and 42%, respectively. This is consistent with the overhead we observed in Query 1, while also reflecting the fact that this query does not significantly benefit from data skipping or partition pruning.

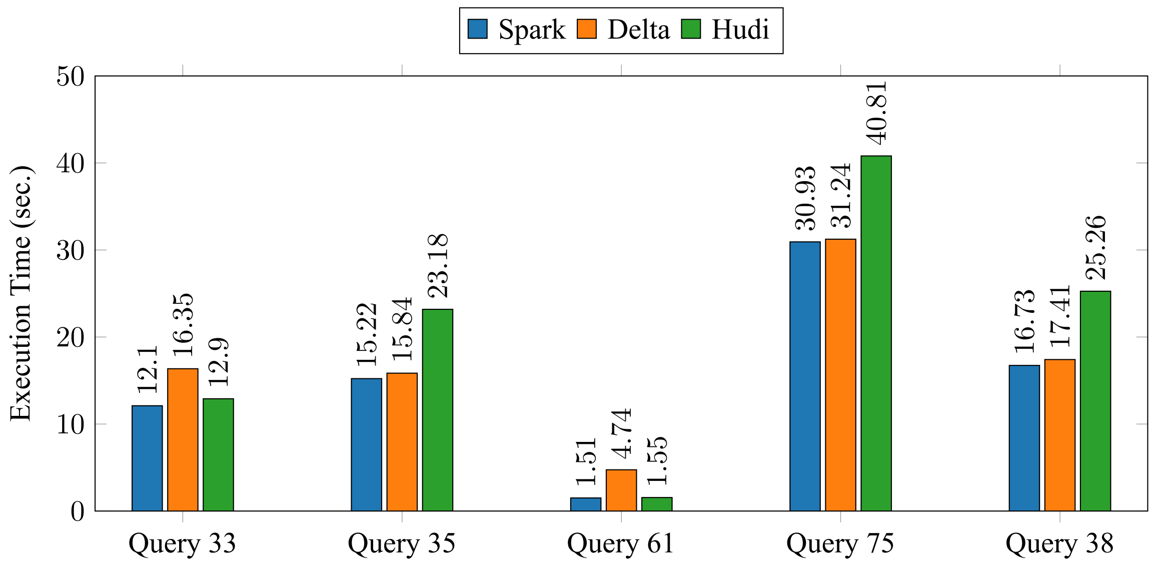
In query 3, although Delta and Hudi underperform relative to Spark, by 13% and 17% respectively, they perform noticeably better, against Spark, compared to the previous queries. Delta marginally outperforms Hudi, however it is important to note that despite the performance improvement, which is expected for filtering queries, the effectiveness of the metadata, particularly the min max statistics which would facilitate data skipping and partition pruning, is limited without partitioning.

In query 4, we again observe noticeably better performance, against Spark, compared to queries 1 and 2. However, performance falters compared to query 3, as the join operation in this query limits the effectiveness of the metadata.

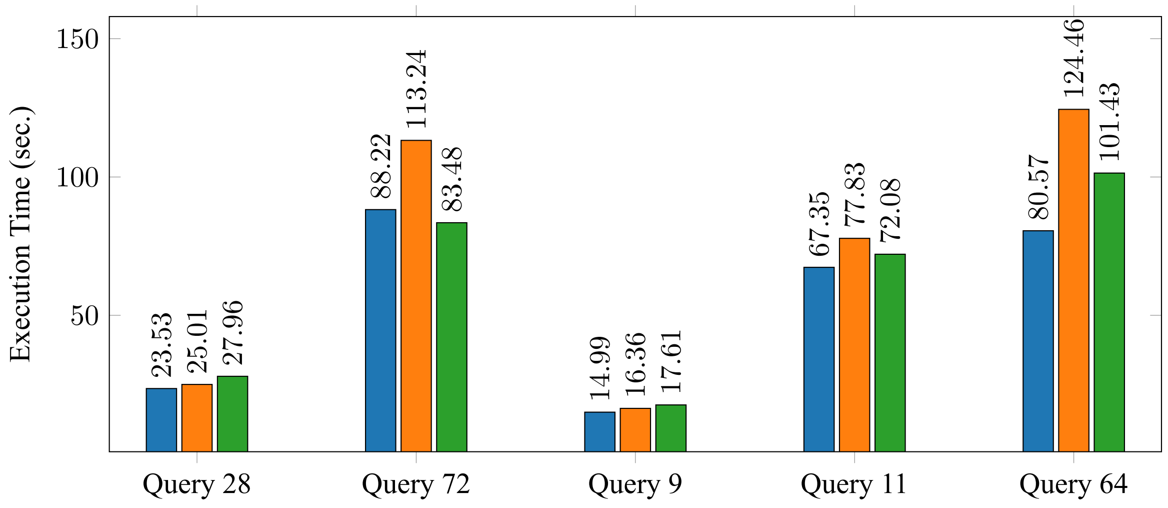
In query 5, Hudi outperforms Delta by a trivial amount, with both outperforming Spark. Since this query does not benefit from data skipping or partition pruning, the improved performance is likely attributed to more efficient query planning and small file compaction, which enables faster data reads.

In query 6, Delta and Hudi underperform relative to Spark, by 19% and 40% respectively. We observe that Delta leverages metadata more efficiently to optimise data sorting, whereas Hudi does so less effectively in comparison.

Complex Queries



(a) Reporting and data mining queries.



(b) Ad-hoc and OLAP queries.

Figure 3.3: Phase 1: Complex query execution times using 2 workers (2 executors) and a 30 GB dataset.

The complex queries allow us to evaluate how effectively each system generates query plans for workloads involving multiple operations and intricate query structures. We observe that Spark marginally outperforms both Delta and Hudi across all complex queries. When comparing Delta and Hudi, we observe that although all complex queries involve some combination of filtering, aggregation, partitioning, and unions, there are specific cases in which one outperforms the other.

Hudi appears to handle queries involving union operations more effectively than Delta. Particularly, queries that perform aggregations and filtering before the union, such as queries 33, 11, and 64. Hudi also performs better in cases where either aggregation or filtering is applied after a union, as demonstrated in queries 72 and 61 respectively. Delta, on the other hand, performs better when handling filtering and multiple aggregation operations. Specifically, queries with heavy filtering and case structures, such as query 9 and queries combining filtering with aggregation, such as queries 28 and 35, the latter also involving extensive partitioning and grouping of data. Delta also performs well when filtering is applied after unions of previously filtered and aggregated data, which is exhibited in query 75, as well as when filtering is applied before unions, such as in query 38.

3.4.2 Phase 2

The subset of queries selected for the subsequent phases includes the simple queries 2 through 6. From the complex queries, it comprises reporting queries 33 and 61, ad-hoc query 28, OLAP queries 11 and 64, and data mining query 38. In total, it contains 5 simple and 6 complex queries.

Through the phase 1 benchmarks and subsequent selective experimentation, we observe that queries 9, 35, 38, and 75 exhibit similar performance characteristics and scaling behaviour. Although query 75 has longer execution times, its scaling pattern is comparable, thus providing insights similar to those of queries 9, 35, and 38. Subsequently, we opted for query 38, as it is the only representation of the data mining queries. Queries 64 and 72 also exhibit similar performance characteristics and scaling behaviour, therefore, we exclude query 72 and retain query 64 due to its more complex and diverse structure compared to the other queries in the subset.

In the following subsections we evaluate the scalability of each system after executing the above subset of queries with 2, 4 and 6 workers. We examine the simple and complex queries separately.

Simple Queries

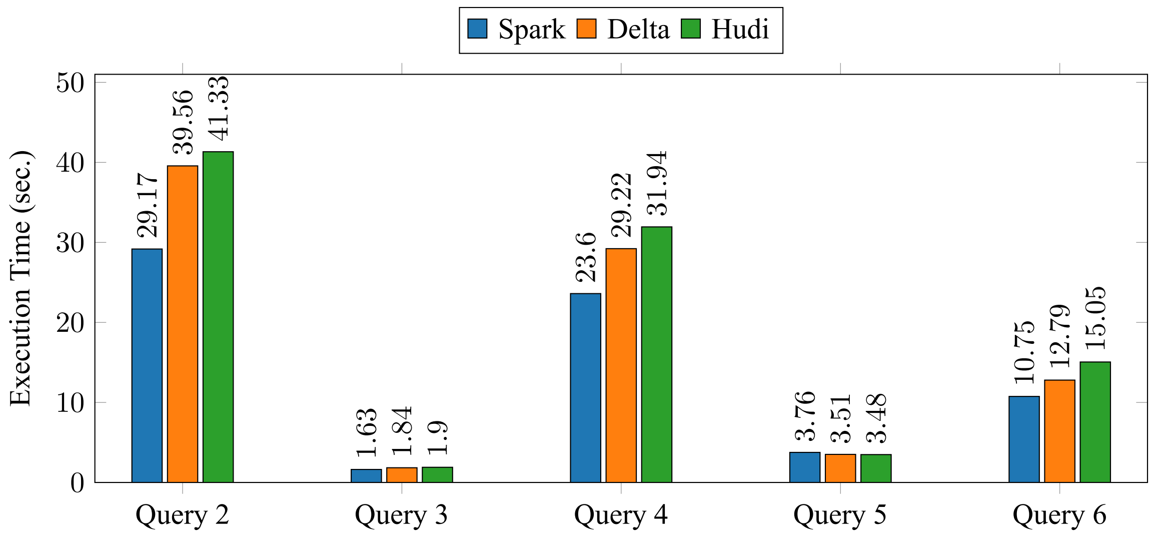


Figure 3.4: Phase 2: Simple query execution times using 2 workers (2 executors) and a 30 GB dataset.

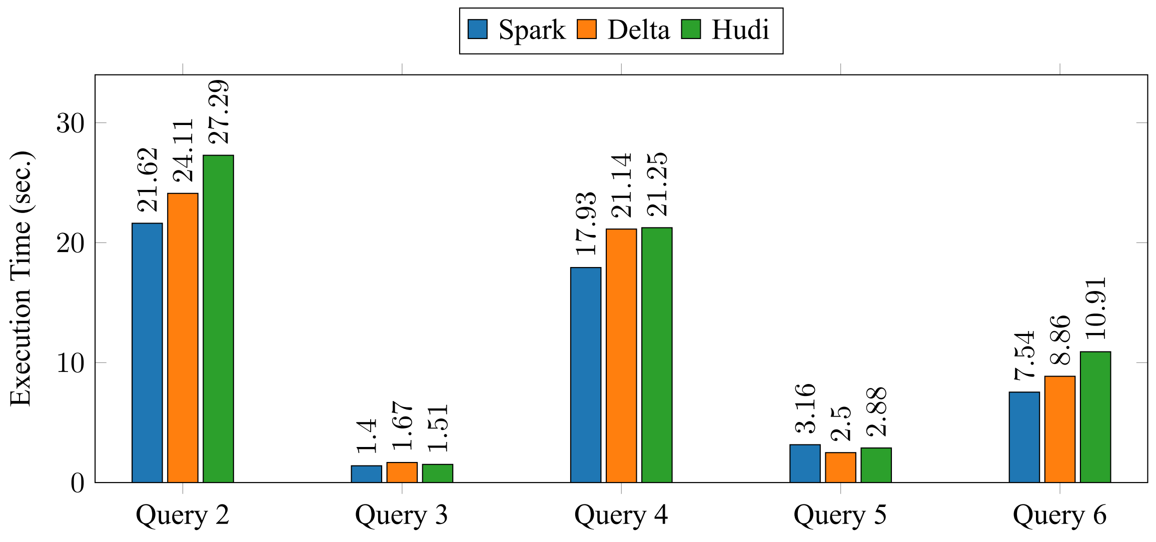


Figure 3.5: Phase 2: Simple query execution times using 4 workers (4 executors) and a 30 GB dataset.

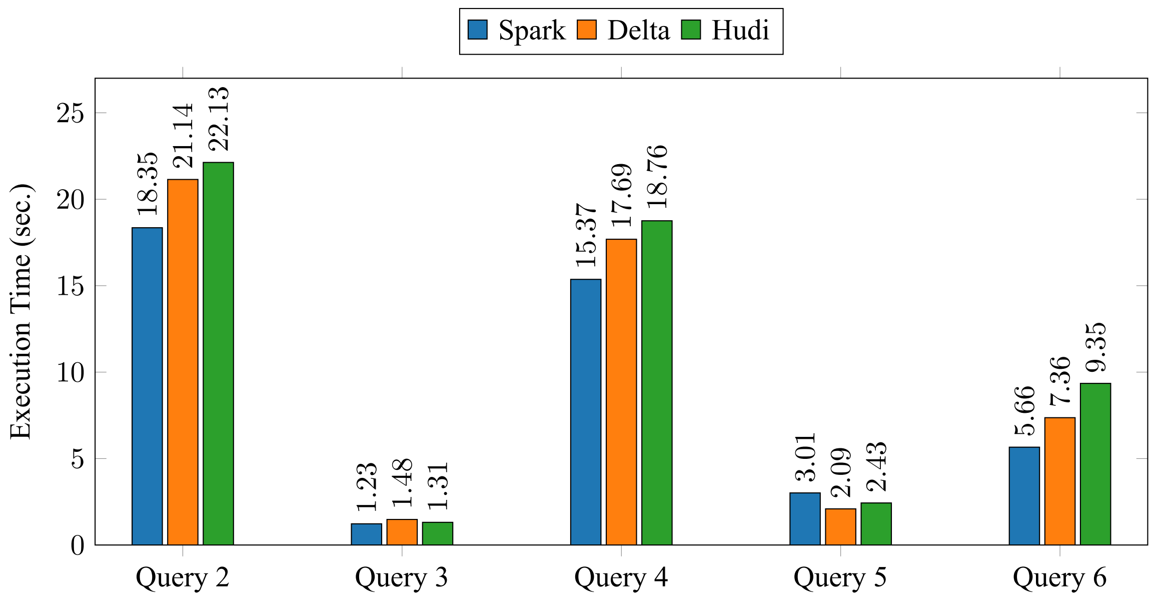


Figure 3.6: Phase 2: Simple query execution times using 6 workers (6 executors) and a 30 GB dataset.

As expected, increasing the number of workers improves performance across all three systems. Spark continues to outperform both Delta and Hudi, except in query 5, while Delta consistently outperforms Hudi, again with the exception of query 5. Hudi benefits the most from scaling out, narrowing the performance gap with Delta and even marginally outperforming it in query 3 when using 4 and 6 workers.

Complex Queries

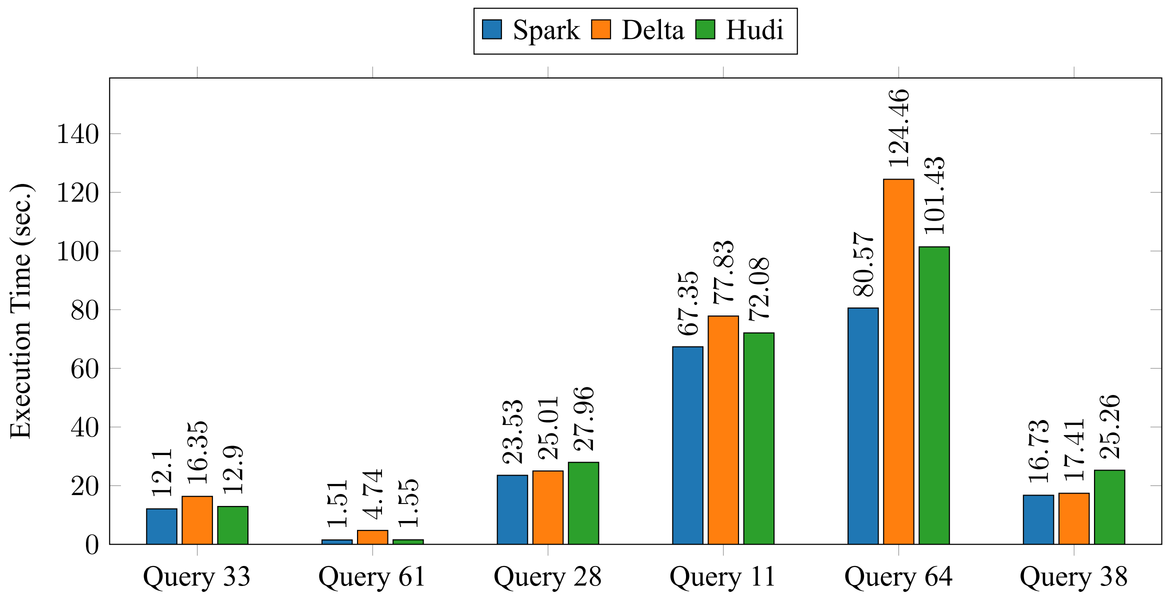


Figure 3.7: Phase 2: Complex query execution times using 2 workers (2 executors) and a 30 GB dataset.

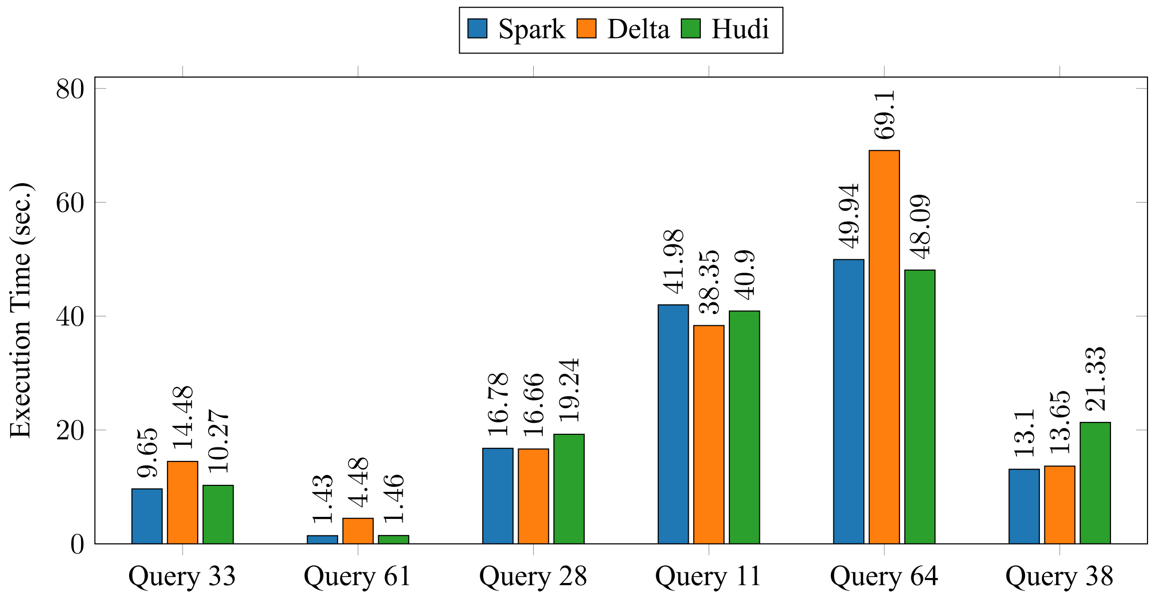


Figure 3.8: Phase 2: Complex query execution times using 4 workers (4 executors) and a 30 GB dataset.

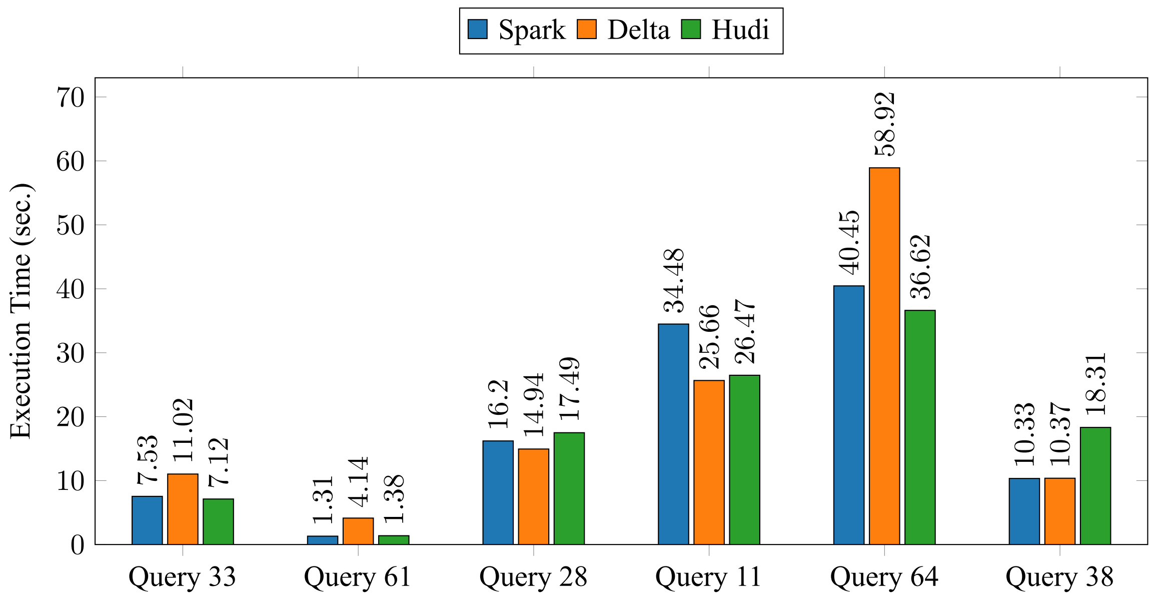


Figure 3.9: Phase 2: Complex query execution times using 6 workers (6 executors) and a 30 GB dataset.

Again, as expected, increasing the number of workers improves performance across all three systems. Both Delta and Hudi benefit from scaling out, which reduces the performance gaps between the two and also Spark. Delta even outperforms Spark in queries 28 and 11 when the number of workers is increased to four and six respectively. Similarly, Hudi outperforms Spark in query 64 after scaling out.

3.4.3 Phase 3

From the benchmarks of the previous phase, we conclude that using 6 workers results in better performance, which is to be expected as Delta, Hudi and Spark are optimised for distributed processing and favour scaling out. Therefore, in this phase we utilise 6 workers and 6 executors, while testing with different dataset sizes. Again, we examine the simple and complex queries separately.

Simple Queries

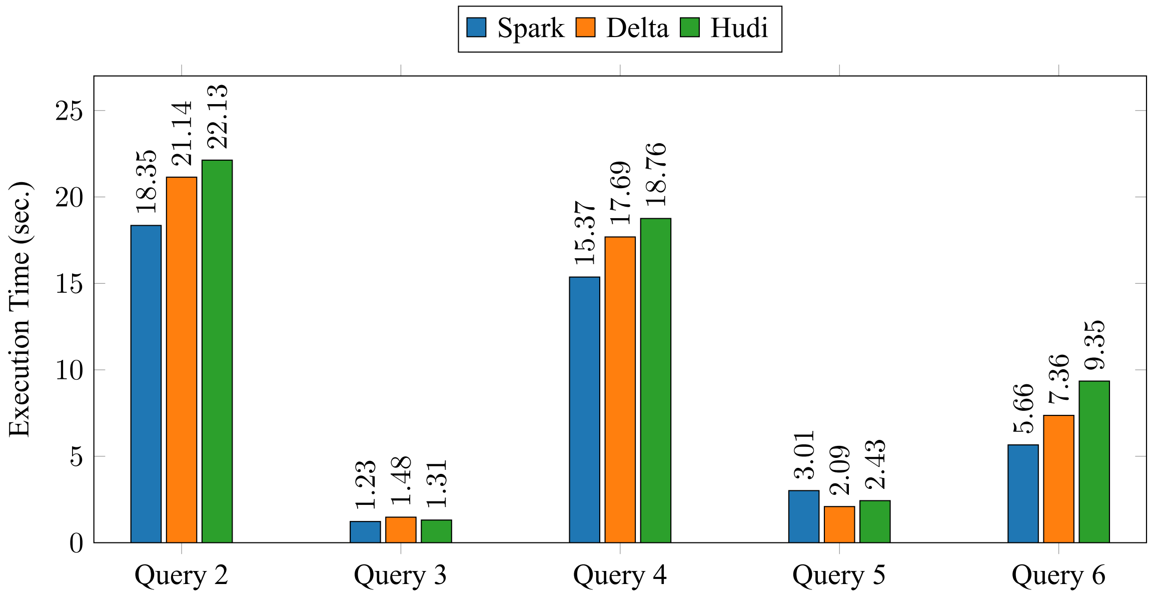


Figure 3.10: Phase 3: Simple query execution times using 6 workers (6 executors) and a 30 GB dataset.

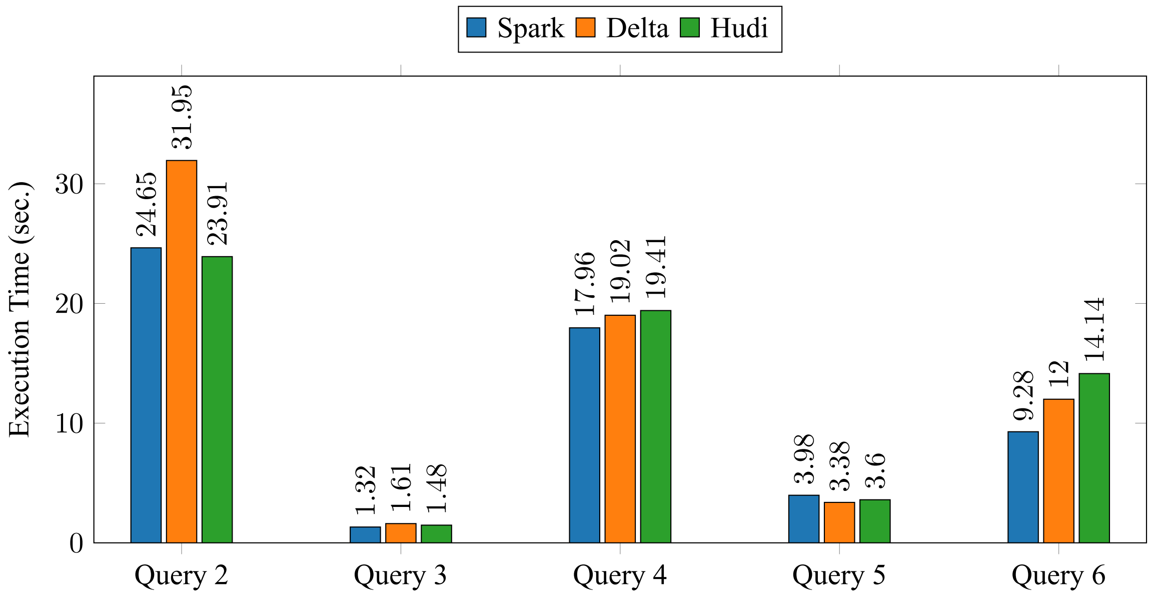


Figure 3.11: Phase 3: Simple query execution times using 6 workers (6 executors) and a 60 GB dataset.

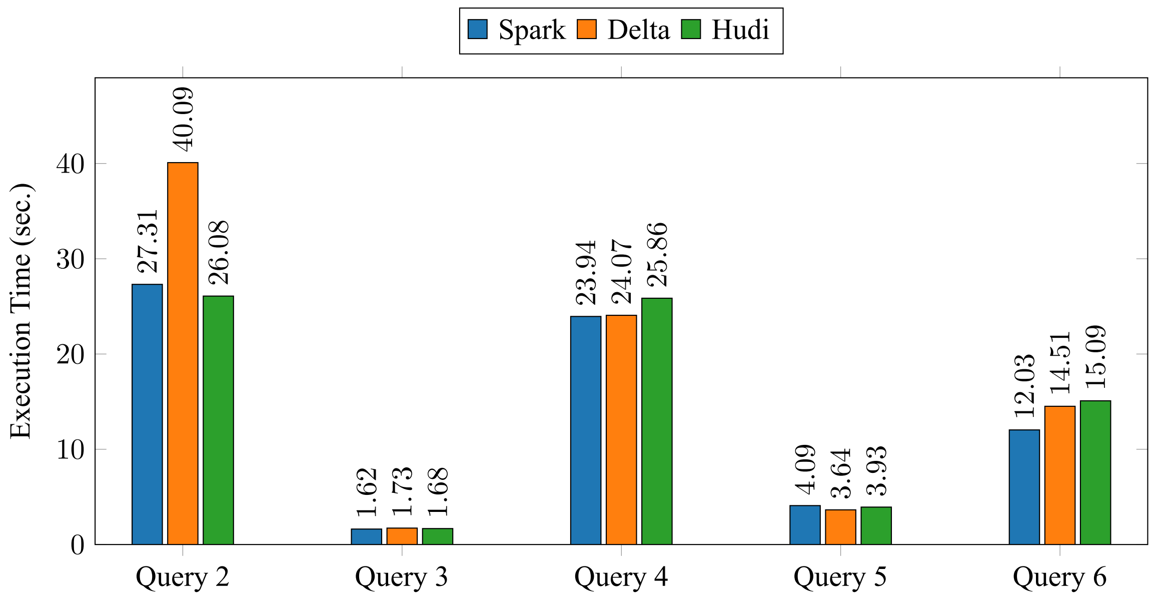


Figure 3.12: Phase 3: Simple query execution times using 6 workers (6 executors) and a 90 GB dataset.

As anticipated, as the dataset size grows so does the execution time of the queries. However, it is worth highlighting that Delta and Hudi's performance gaps compared to Spark decrease as the dataset grows, with Hudi even outperforming Spark in query 2.

Complex Queries

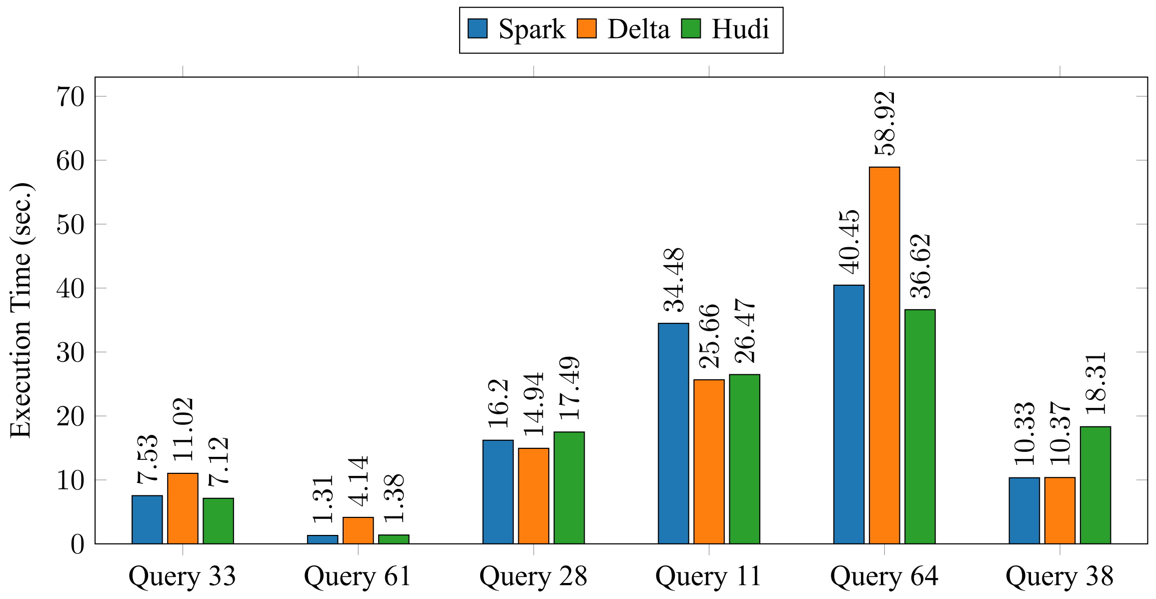


Figure 3.13: Phase 3: Complex query execution times using 6 workers (6 executors) and a 30 GB dataset.

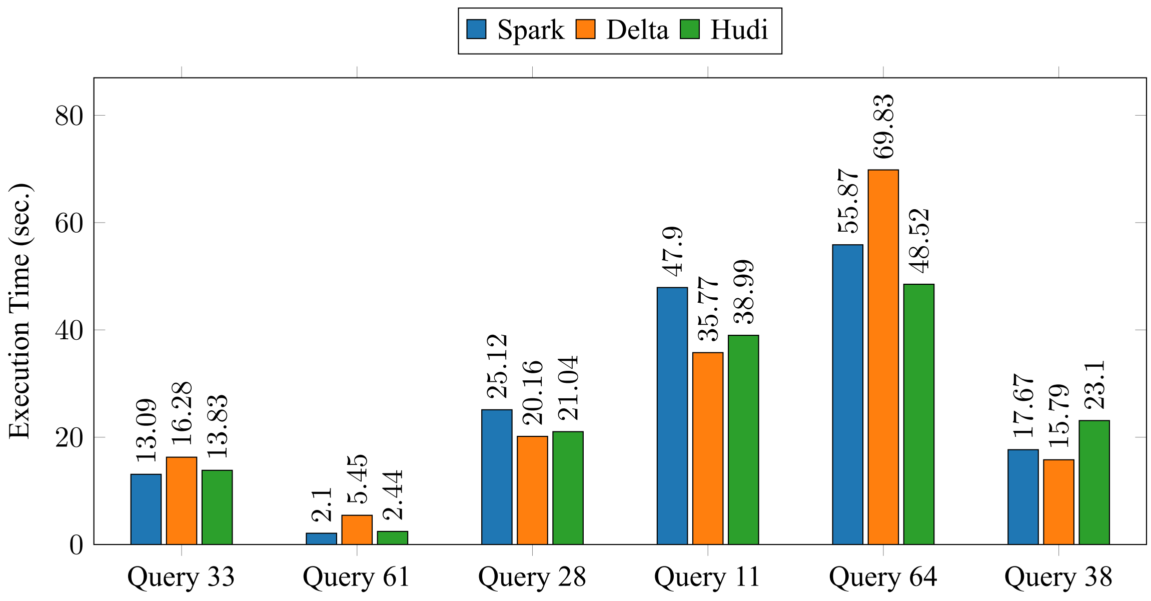


Figure 3.14: Phase 3: Complex query execution times using 6 workers (6 executors) and a 60 GB dataset.

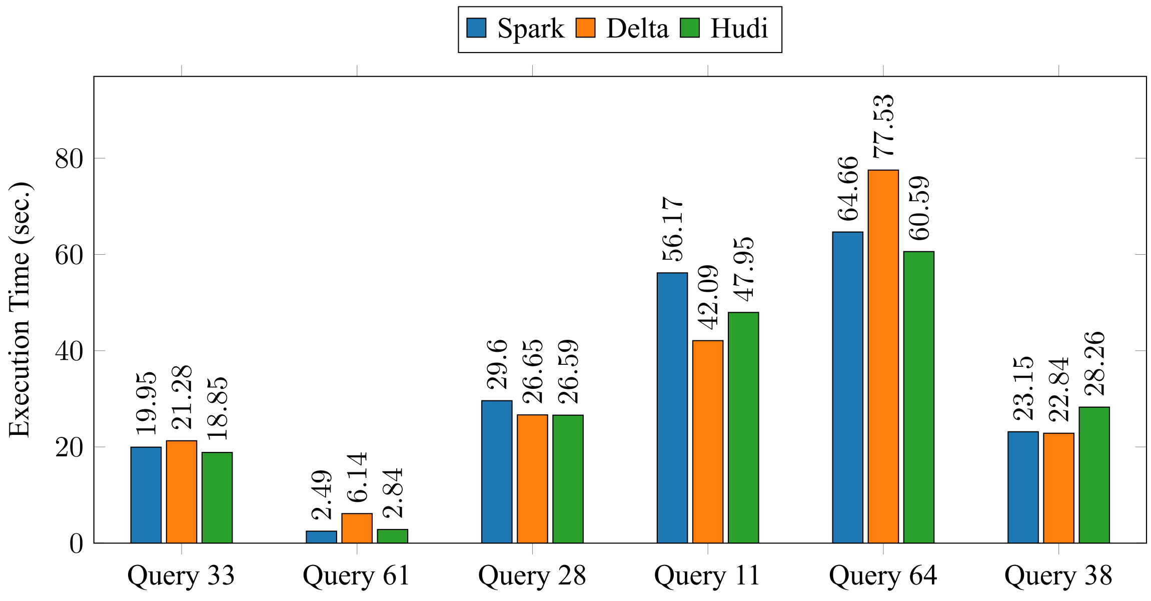


Figure 3.15: Phase 3: Complex query execution times using 6 workers (6 executors) and a 90 GB dataset.

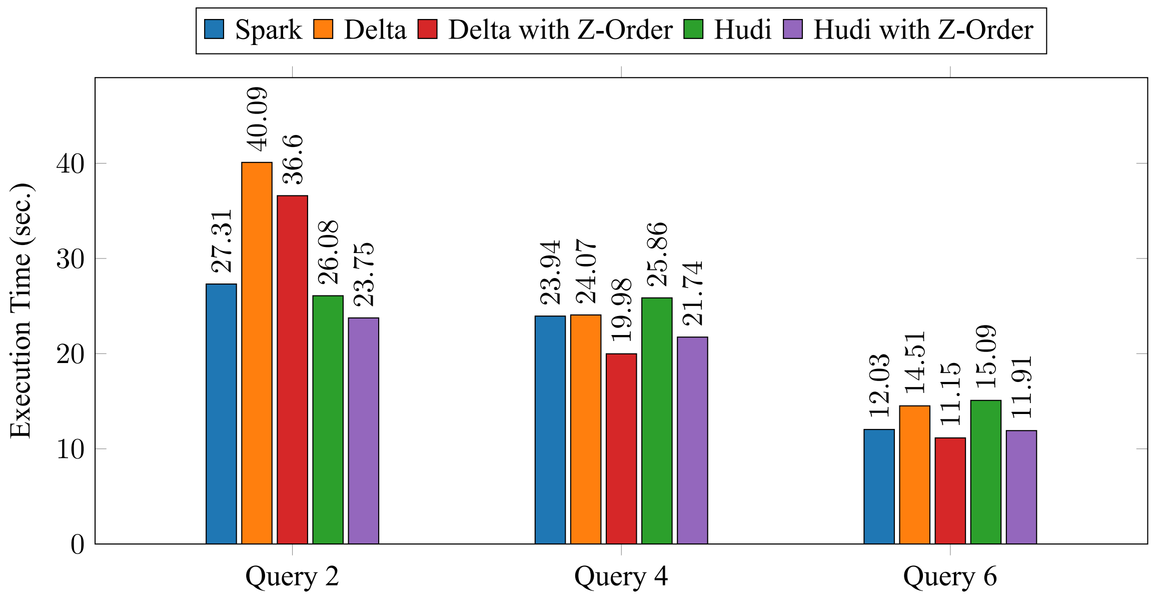
Again, as the execution times increase along with the dataset size, the performance gaps between Delta, Hudi and Spark decrease. Delta outperforms Spark in query 38 when using the 60 GB and 90 GB datasets, and while only Delta outperforms Spark in query 28 when using a 30 GB dataset, for larger datasets, so does Hudi.

3.4.4 Phase 4

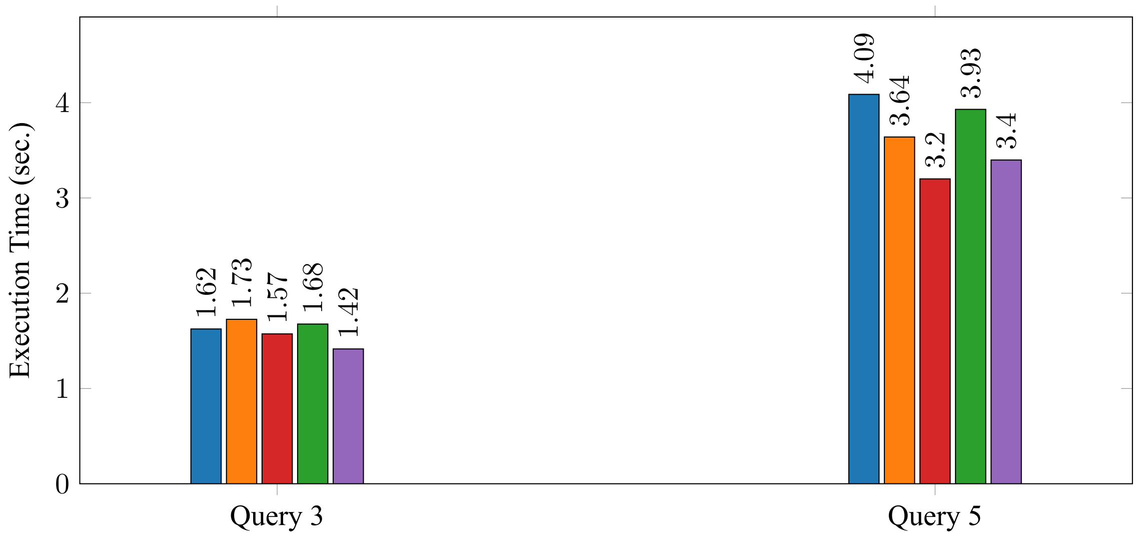
In this phase, using 6 workers and a 90 GB dataset, we compare performance after partitioning the data and indexing them using the Z-Order curve. For each query, we ensure that the data is indexed based on columns utilised for filtering, partitioning etc. in the query. Otherwise, indexing would not impact performance and thus, the benchmarks would not provide any insights. Since Delta and

Hudi are designed for big data, we select the 90 GB dataset as it is the largest dataset our setup can accommodate. We examine the simple and complex queries separately.

Simple Queries



(a) Simple queries under 29 seconds.



(b) Simple queries over 29 seconds.

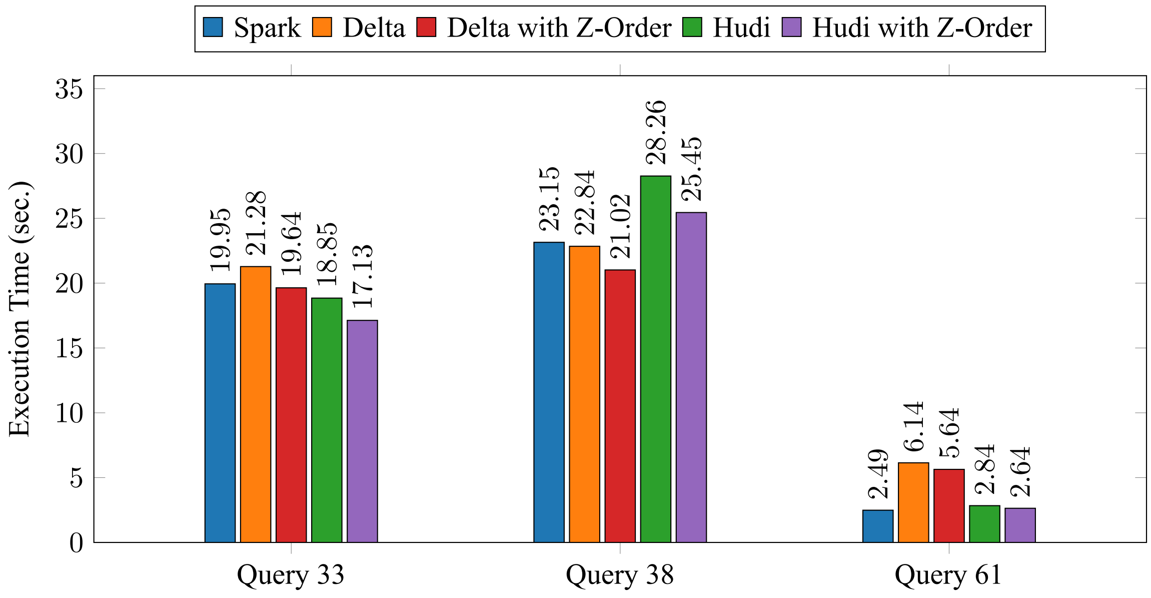
Figure 3.16: Phase 4: Simple query execution times using 6 workers (6 executors) and a 90 GB dataset.

For all queries, the application of partitioning and indexing results in performance improvements, in both Delta and Hudi. In particular, in queries 3, 4 and 6, Delta and Hudi even outperform Spark, which was not the case prior to the optimisations. Query 6, which performs data sorting, exhibits the most significant performance boost, with execution time being reduced by 26.2% and 23.6% for Delta and Hudi respectively. This outcome is justifiable, as the data is already partitioned and indexed, which significantly improves sorting efficiency.

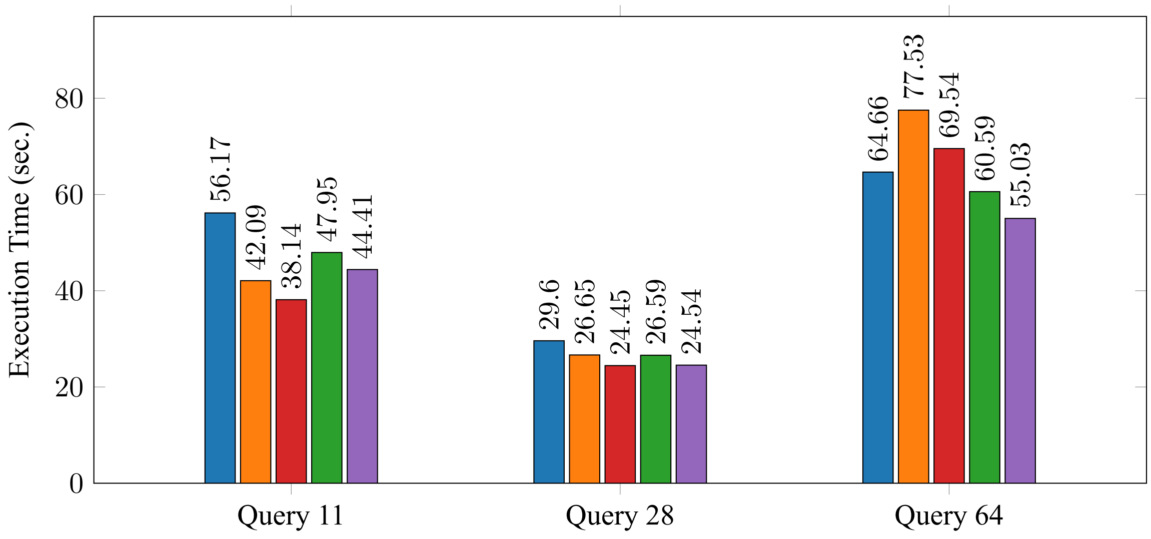
Filtering queries also benefit considerably, with query 3 demonstrating a 9.7% and 16.8% decrease for Delta and Hudi respectively. In query 4, where two tables are filtered before being joined, the performance improvements for filtering queries appear to be compounding, with a 18.6% and 17.3% decrease respectively.

Query 5 also benefits, due to improved data grouping for aggregation after partitioning the data, resulting in a 12.9% and 14.5% decrease respectively. Finally, query 2 demonstrates performance improvements of 9.1% and 9.4% for Delta and Hudi, respectively, as a result of partitioning on the join equality column.

Complex Queries



(a) Complex queries over 10 seconds.



(b) Complex queries under 5 seconds.

Figure 3.17: Phase 4: Complex query execution times using 6 workers (6 executors) and a 90 GB dataset.

Again, all queries exhibit performance improvements after applying partitioning and indexing for Delta and Hudi. Notably, in query 33, Delta outperforms Spark, contrary to before applying the optimisations. Although the performance improvements observed are significant, they are smaller in scale compared to those observed in the simple queries. This is to be expected, as complex queries involve a variety of operations and utilise multiple table columns, thus benefiting from the optimisations to a lesser extent.

3.5 Conclusions

Based on the phase 1 benchmarks, we infer that for queries involving simple operations and query structure, Delta Lake generally outperforms Hudi by a small margin, while both perform slightly below Spark. For more complex queries, Spark outperforms both Delta and Hudi, while the relative performance between Delta and Hudi depends on the query type. Hudi tends to generate more efficient query plans for queries involving many unions, whereas Delta performs better on queries with extensive filtering and aggregations.

Big data solutions such as Spark, Delta, and Hudi are designed and optimised for distributed processing and scaling out. As expected, increasing the number of workers improves performance across all three systems. However, from the phase 2 benchmarks, we deduce that while all systems benefit from scaling out, Delta and Hudi exhibit greater performance improvements, reducing the performance gaps with Spark and in some cases, even outperforming it.

From the phase 3 benchmarks, we conclude that both Delta and Hudi benefit from operating on larger datasets. While increasing dataset size naturally leads to longer execution times, it also reduces the performance gaps between the two systems and Spark. The performance improvements achieved by the optimisations introduced by Delta and Hudi, such as metadata, file compaction, and data skipping, outweigh their associated overheads only when the dataset is sufficiently large.

Delta and Hudi introduce many optimisations, one of them being indexing. From the phase 4 benchmarks, we deduce that indexing and partitioning the data is especially effective for the simple queries, when the columns used for indexing are also used in the query for filtering or partitioning the data. While slightly less effective for complex queries, the optimisations still provide substantial performance improvements, enough to fully justify their associated overheads.

Chapter 4

Stream Processing

Stream processing is the second of the two main methods of data processing in modern big data management systems. Stream processing is the continuous ingestion and processing of data as it arrives. Data items are processed individually and immediately upon arrival, as opposed to being grouped into batches. This method enables real time responses and low latency analytics required for applications such as monitoring, fraud detection, live analytics etc [Zhao17]. Stream processing systems are designed to handle high throughput data streams while maintaining consistency and fault tolerance. By leveraging distributed computing frameworks, they can scale out to accommodate large volumes of rapidly incoming data. In this chapter, we conduct experiments to evaluate how the addition of Delta or Hudi to a data lake impacts the performance of stream processing workloads.

4.1 Dataset

For our experiments, we utilise historical stock market data obtained from the National Stock Exchange (NSE) of India. The dataset comprises information on 100 stocks and 2 major indices, the Nifty 50 and the Nifty Bank. The data spans the period from 02-02-2015 to 08-11-2024 covering trading days from Monday to Friday. For each trading day, data is recorded at one minute intervals from 9.15 AM to 15.29 PM, resulting in 375 records per stock per day. Each data record includes a timestamp, the stock name and OHLCV values. These values are opening price, high price, low price, closing price, and trading volume.

4.1.1 Medallion Architecture

In this subsection, we present the data design pattern we adhere to for our stream processing experiments, in order to organise the data within the lake. The medallion architecture, also referred to as the multi-hop architecture, is suitable for developing incremental ETL pipelines for both streaming and small batch datasets. This architecture incorporates a series of data layers that denote the quality of data stored in them, typically comprising three layers: the bronze, the silver and the gold layer [Data24].

The principal purpose of this data design pattern is to incrementally and progressively enhance the structure, quality and reliability of the data as it flows through each layer. By progressing data through layers, organizations can systematically refine and validate datasets, thus making it more suitable for BI and machine learning applications. The medallion architecture guarantees atomicity, consistency, isolation, and durability (ACID) as data passes through multiple layers of validations and transformations before being stored in a layout optimised for efficient analytics. In the he following sections we examine each layer within the medallion architecture in detail [Data24].

Bronze Layer

In the bronze layer raw data is ingested and maintained in its original format from any combination of streaming or batch sources including cloud object stores, message buses, federated systems etc. The data grows over time as new data is appended incrementally. The data within this layer is intended for consumption by ETL pipelines that enrich silver layer tables, not for direct access. It serves as the single source of truth for the data lake, while also enabling reprocessing and auditing by retaining all historical data [Data24].

Silver Layer

The silver layer is responsible for cleansing and normalising the data. Its primary purpose is to enhance data quality by eliminating inconsistencies and structuring data into suitable formats for downstream processing. In this layer, data is read from one or more bronze or silver tables, validated and then written to silver tables. The introduction of this layer mitigates failures introduced due to schema changes or corrupt records in data sources, that would otherwise occur if the data was written directly from ingestion. When correctly applying the medallion architecture, this layer should always contain at least one validated, non-aggregated representation of each record. Common operations performed within the silver layer to enforce data quality include schema enforcement and evolution, handling of null and missing values, data deduplication, filtering based on business constraints etc [Data24].

Gold Layer

The gold layer contains highly refined views of the data that enable downstream analytics, ML and data driven applications. Data within this layer is often aggregated and enhanced with joins to align with business logic. Typically, it is stored in materialised views to optimise query performance for analytics and reporting purposes. The gold layer consists of semantically meaningful datasets that corresponds directly to business functions and requirements [Data24].

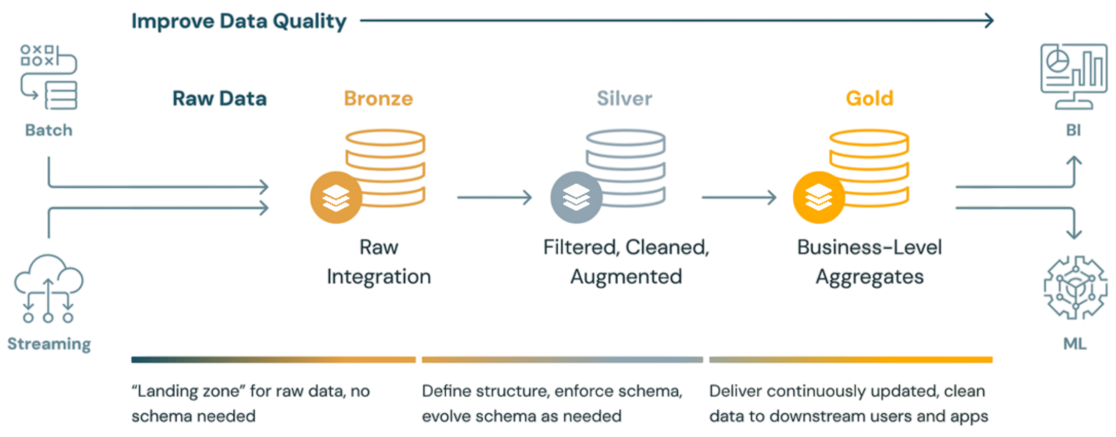

Improve Data Quality
Batch
Raw Data
Bronze
Silver
Gold
BI
Raw Integration
Filtered, Cleaned, Augmented
Business-Level Aggregates
Streaming
ML
"Landing zone" for raw data, no schema needed
Define structure, enforce schema, evolve schema as needed
Deliver continuously updated, clean data to downstream users and apps

Figure 4.1: The medallion architecture. Reproduced and edited from Databricks.

4.1.2 Organisation of the Dataset Using the Medallion Architecture

As mentioned above, the dataset used for our experiments is organised according to the medallion architecture. The architecture is implemented as follows: in the bronze layer, data is ingested in CSV format and stored in its raw form in HDFS. In the silver layer, the data is transformed by adding a derived date column, which is generated by applying the to_date SQL function, and then saved in Parquet format. Finally, in the gold layer, queries are executed on the data from the silver layer and the results are stored as gold tables. The following section presents the seven queries executed to generate the gold tables.

4.2 Queries

Streaming queries can be classified into two types, filtering or transformation operations applied independently to each record and aggregations or joins that must be executed within defined time windows, due to the continuous and unbounded nature of data streams. For our experiments, we have chosen to focus on filtering and windowed aggregation queries. Transformation queries are excluded, as these are already performed within the silver layer. Additionally, windowed join queries are omitted due to their increased complexity, which would require separate, dedicated experimentation. In the next subsections we introduce the two filtering queries and the five windowed aggregation queries we select to execute.

4.2.1 Filtering

The purpose of the filtering queries is to assess the impact of the Delta and Hudi metadata implementations on filtering performance, compared to that of a standard data lake. The first filtering query is an equality filtering query based on the stock name.

```
silverDF.filter(silverDF.name == "BPCL")
```

Listing 4.1: Filtering Query 1.

The second query is an range filtering query based on the low and high values of the data.

```
silverDF.filter((silverDF.low < 990) & (silverDF.high > 1000))
```

Listing 4.2: Filtering Query 2.

4.2.2 Windowed Aggregations

The purpose of the windowed aggregation queries is to evaluate the effectiveness of Delta and Hudi in executing various types of aggregation operations. The reasoning behind the selection of the window duration is discussed in detail in section 4.3. The first windowed aggregation query is a sliding window average on the high value of the data.

```
silverDF.withWatermark("timestamp", "1day")
        .groupBy(window("timestamp", "60days", "30days"), "name")
        .avg("high")
```

Listing 4.3: Windowed Aggregation Query 1.

The second query is a tumbling window average on the high value of the data.

```
silverDF.withWatermark("timestamp", "1day")
        .groupBy(window("timestamp", "60days"), "name")
        .avg("high")
```

Listing 4.4: Windowed Aggregation Query 2.

The third query is a tumbling window minimum on the low value of the data.

```
silverDF.withWatermark("timestamp", "1day")
        .groupBy(window("timestamp", "60days"), "name")
        .min("low")
```

Listing 4.5: Windowed Aggregation Query 3.

The fourth query is a tumbling window sum on the volume value of the data.

```
silverDF.withWatermark("timestamp", "1day")
         .groupBy(window("timestamp", "60days"), "name")
         .sum("volume")
```

Listing 4.6: Windowed Aggregation Query 4.

The fifth and final query is a tumbling window with multiple aggregations.

```
silverDF.withWatermark("timestamp", "1day")
         .groupBy(window("timestamp", "60days"), "name")
         .agg(avg("high"), min("low"), sum("volume"))
```

Listing 4.7: Windowed Aggregation Query 5.

4.3 Experimentation Process

We initially conduct a preliminary round of experiments to determine the sustainable throughput for each query. When a system ingests data at a rate exceeding its processing ability, it begins to accumulate backpressure, during which incoming data is queued for processing after the system has completed handling previously ingested data. As a result, once backpressure is triggered, the processing latency of subsequently queued data increases. Sustainable throughput refers to the maximum data ingestion rate a system can handle without experiencing sustained backpressure, which results in continuously increasing latency [Kari18].

Through our experiments, we determine that the sustainable throughput is 50,000 records per second for the filtering queries and 100,000 records per second for the windowed aggregation queries. The sustainable throughput also informed the size of the window in aggregation queries. We select a 60 day window to avoid increased latency and errors arising from excessive parallel query execution, as well as segmentation faults caused by processing large volumes of data. This duration is long enough to prevent over-segmenting each batch of data across multiple windows and short enough to avoid storing excessive amounts of data in memory before the window can be closed and processing can begin.

After determining these key metrics, the experiments are conducted in the following manner. Using 6 workers and 6 Spark executors, each with 4 vCPUs and 8 GB of RAM, the queries are executed at their sustainable throughput and also after increasing and decreasing the ingestion rate by 10,000 records per second. This approach not only enables us to evaluate how each SUT operates at sustainable throughput but also allows the evaluation of the performance when the ingestion rate falls below or exceeds the processing capacity.

4.3.1 Benchmark Metrics

The primary metrics used in benchmarking batch processing systems are latency and throughput. However, since our experiments are conducted around the sustainable throughput of each query, latency remains relatively stable and does not offer additional insights into system efficiency. Therefore, system performance is evaluated primarily based on throughput.

4.4 Results

The results of our experiments are presented in three parts, corresponding to the three layers through which the data flows in our application of the medallion architecture.

4.4.1 Bronze Layer

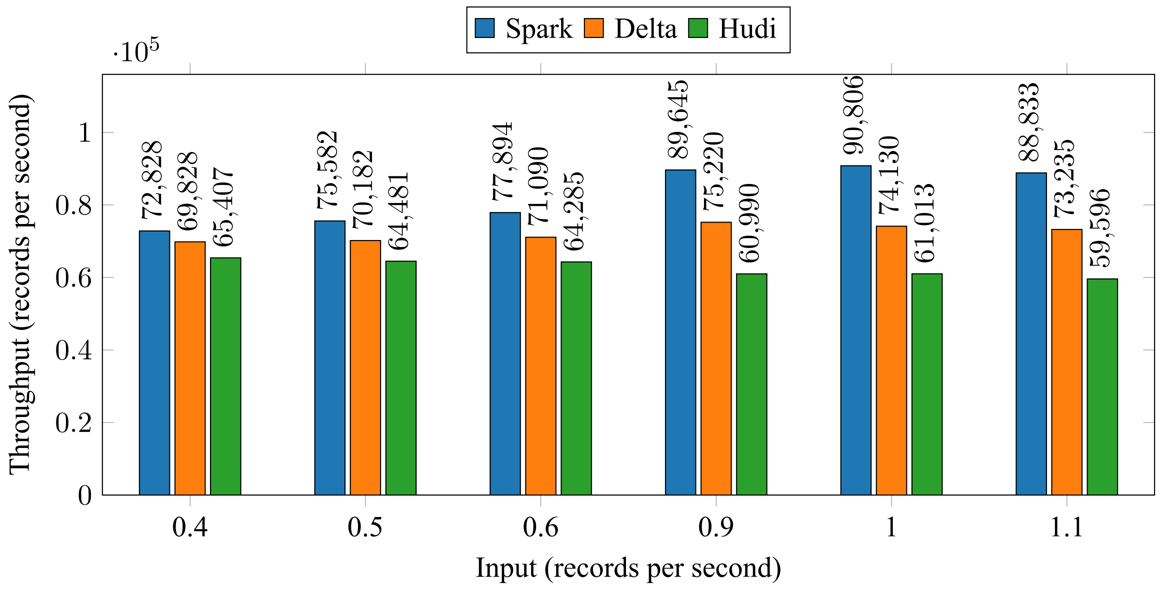


Figure 4.2: Bronze Layer: Throughput for increasing input rates.

In the bronze layer, Spark consistently outperforms both Delta and Hudi. Data is ingested directly from CSV files, which are not optimised for distributed processing. Combined with the overhead introduced by metadata collection during ingestion, it leads to the reduced performance of Delta and Hudi. Among the two, Delta performs better than Hudi, likely due to Hudi's more complex and robust metadata structures, which incur higher ingestion overhead. Additionally, Spark demonstrates better scalability as input size increases, with the throughput peaking at approximately 90,000 records per second, compared to around 75,000 for Delta and 65,000 for Hudi. We observe that, after reaching their peak, further increases to the input rate incurs a decrease in throughput, due to the introduction of backpressure, which applies additional stress on the systems.

4.4.2 Silver Layer

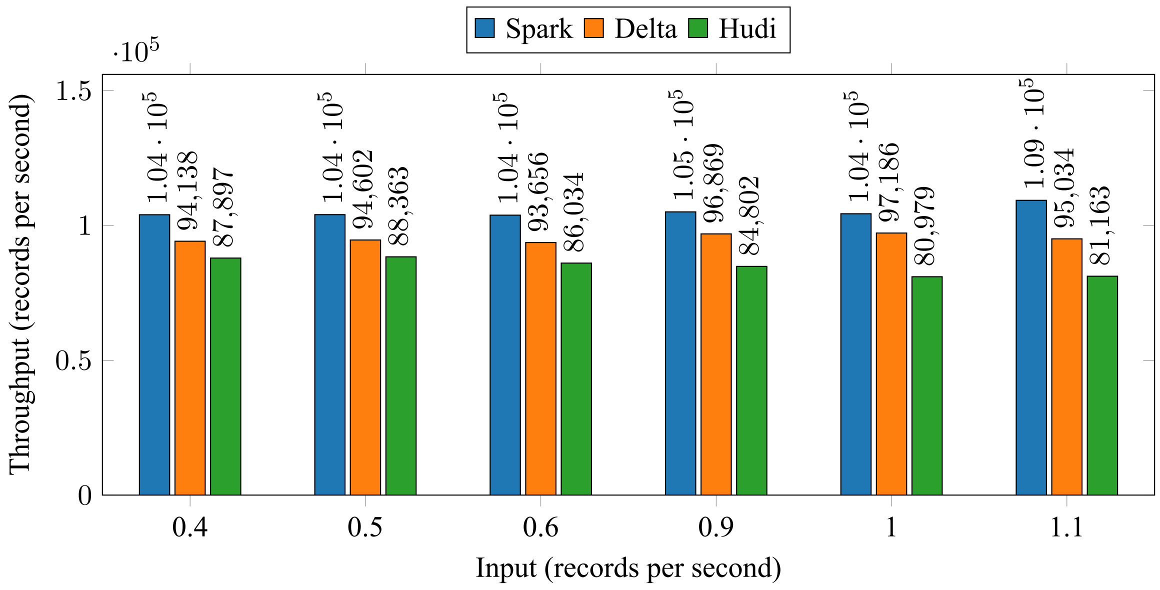


Figure 4.3: Silver Layer: Throughput for increasing input rates.

In the silver layer, while Spark continues to outperform both Delta and Hudi, the performance gaps between them have narrowed. Data is ingested from Parquet files, enabling Delta and Hudi to better leverage distributed processing during metadata collection. Metadata collection is limited to the new column, as metadata from the previous layer is preserved, significantly reducing overhead. Spark again demonstrates better scalability, with the throughput peaking at approximately 109,000 records per second. However, Delta exhibits improved scaling compared to the bronze layer, reaching a peak throughput of 97,000 records per second, followed by Hudi with a peak of 87,000 records per second. As with the previous layer, throughput declines after reaching the peak input rates due to increased system stress.

4.4.3 Gold Layer

In this section, we evaluate how each system performs for different query types, that can be applied to generate gold tables. We examine the filtering and windowed aggregation queries separately.

Filtering Queries

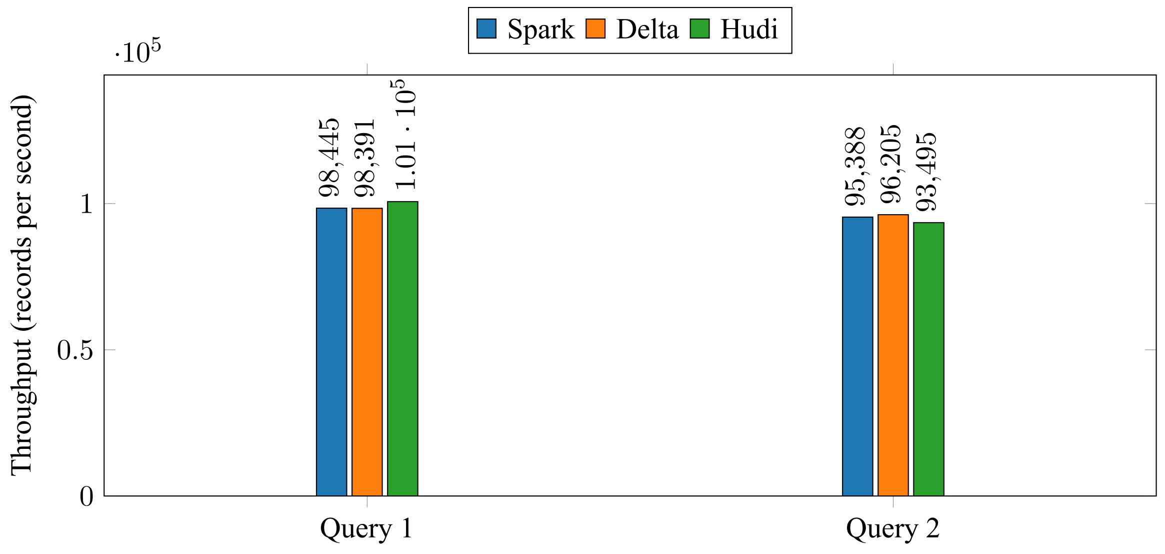


Figure 4.4: Gold Layer: Filtering queries throughput with a 90,000 records/sec input rate.

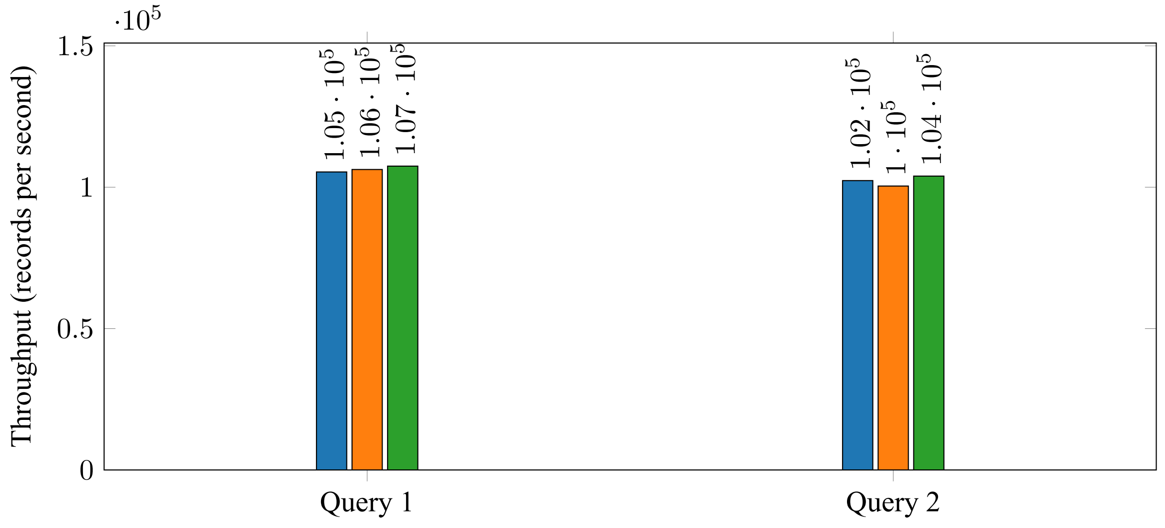


Figure 4.5: Gold Layer: Filtering queries throughput with a 100,000 records/sec input rate.

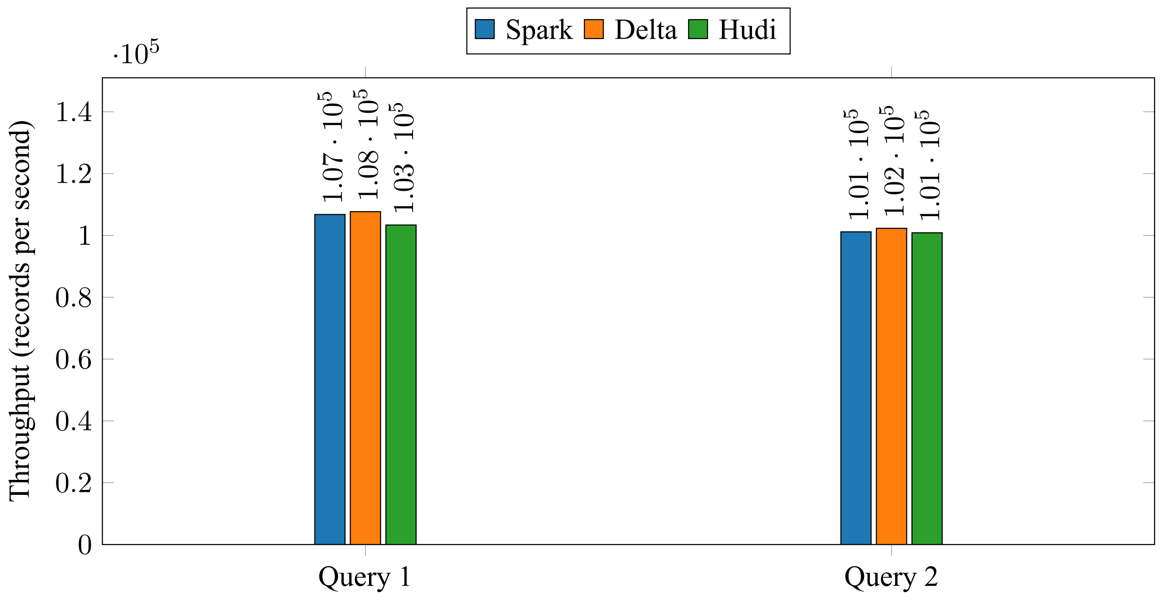


Figure 4.6: Gold Layer: Filtering queries throughput with a 110,000 records/sec input rate.

For the filtering queries, all three systems exhibit comparable performance. Delta and Hudi leverage the metadata to apply optimisations such as data skipping, which allow them to filter data more efficiently than Spark. Hudi achieves the best performance when the input rate aligns with the sustainable throughput. At higher input rates, Delta surpasses both Hudi and Spark, demonstrating better performance under the stress of backpressure.

Windowed Aggregation Queries

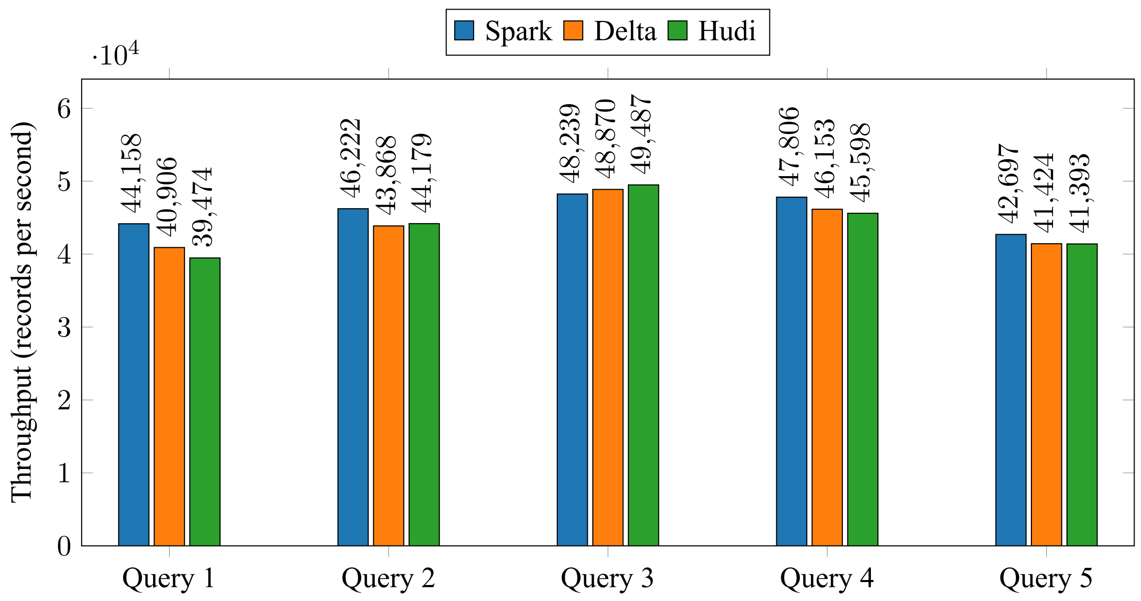


Figure 4.7: Gold Layer: Windowed aggregation queries throughput with a 40,000 records/sec input rate.

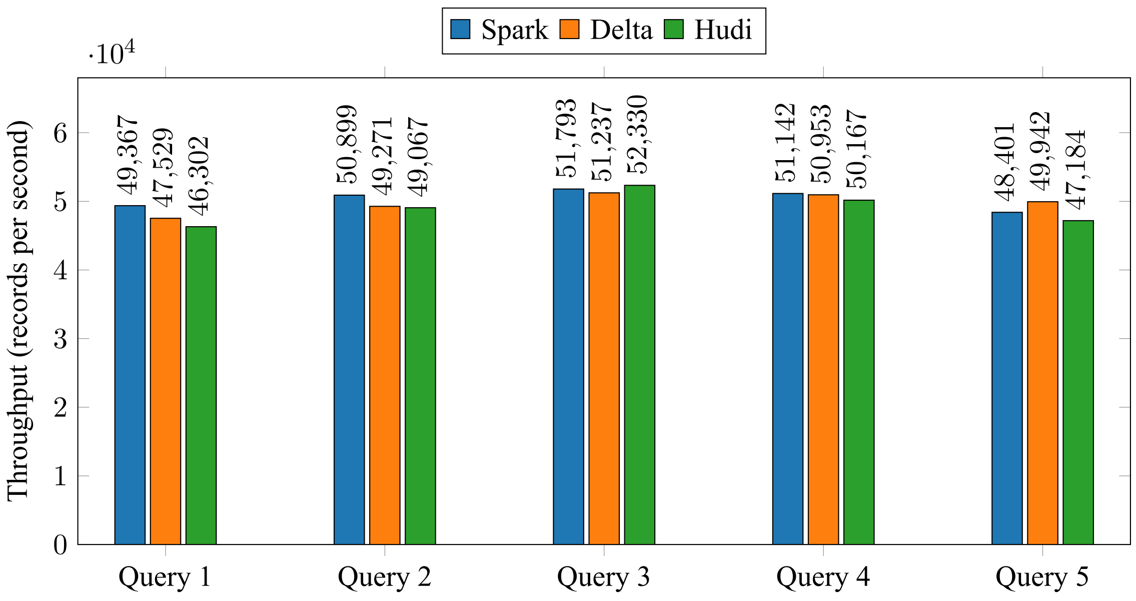


Figure 4.8: Gold Layer: Windowed aggregation queries throughput with a 50,000 records/sec input rate.

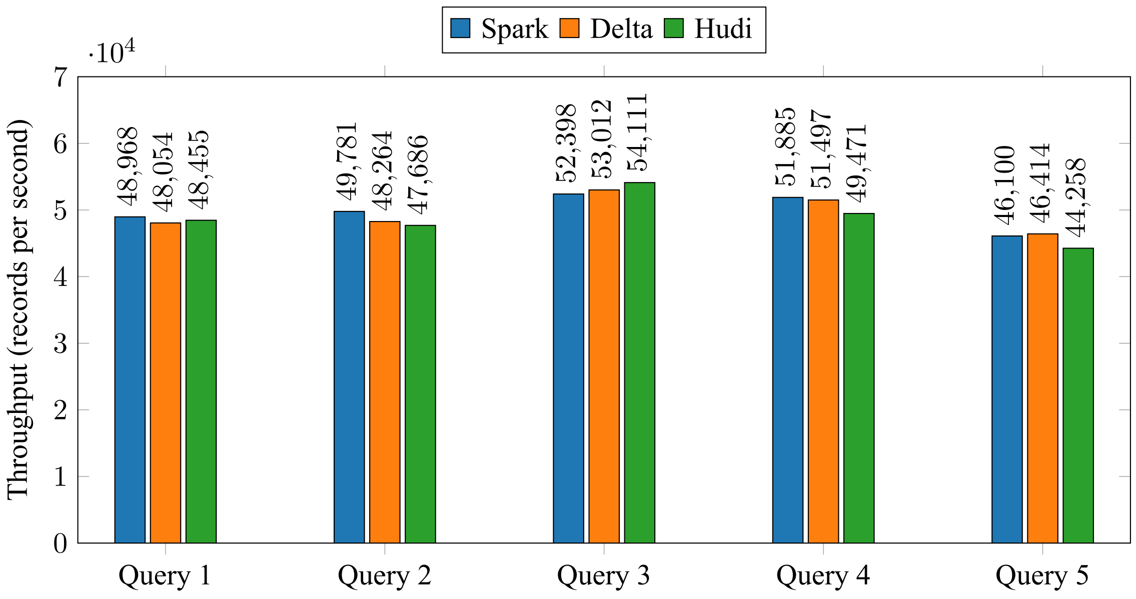


Figure 4.9: Gold Layer: Windowed aggregation queries throughput with a 60,000 records/sec input rate.

Again, the performance of all three systems is comparable. Spark outperforms both Delta and Hudi across most scenarios, with the exception of query 3, which is a tumbling window minimum aggregation, where it is outperformed by both for all input rates. Additionally, in query 5, which is a tumbling window with multiple aggregations, Delta surpasses Spark and Hudi once the input rate exceeds 50,000 records per second. Overall, Delta generally demonstrates better performance than Hudi, except in query 3, where Hudi achieves better results.

4.5 Conclusions

Based on the benchmarking results, we conclude that Spark generally outperforms Delta and Hudi during ingestion, as the additional overhead introduced by metadata collection, by Delta and Hudi, impacts their performance. Between the two, Delta generally outperforms Hudi, which can be attributed to Hudi's more complex metadata management structures that result in larger overheads.

All three systems exhibit similar performance in filtering queries, with Spark performing slightly better. Under high input rates, Delta appears to be performing better than both Spark and Hudi, while Hudi performs better when the input rate matches the sustainable throughput.

In aggregation queries, Spark generally outperforms both Delta and Hudi, with the exception of the tumbling window minimum aggregation, where both Delta and Hudi outperform Spark consistently and the tumbling window with multiple aggregations, where Delta exceeds the performance of both Spark and Hudi once the input rate surpasses 50,000 records per second.

Chapter 5

Summary and Future Directions

Lakehouses were introduced as a solution to the problems of current data lake architectures. Standard data lakes are not well suited to advanced analytics and machine learning due to their lack of features, such as data cleaning, labelling, and organisation, which leads to the need for complex ETL pipelines. Additionally, the lack core RDBMS features such as transactions, indexing, and caching, makes them inefficient for complex workloads containing inconsistent and dirty data [Hai23].

The use of two-tier architecture increases complexity and costs due to continuous ETL pipelines, duplicate storage, while also complicating migration to other platforms due to the reliance on proprietary formats. This architecture also introduces data staleness, as updates in the lake are not immediately visible in the warehouse. ETL delays and format inconsistencies reduce reliability, as new bugs and failure modes are introduced through the pipelines. Moreover, upstream warehouses only contain a small subset of the data lake data, which is not sufficient for effective machine learning applications, thus requiring additional ETL steps that complicate integration and raise costs [Armb21].

Lakehouses address the above issues through the introduction of a metadata layer over lake storage, the reliance on open file formats and SQL specific optimisations to bridge the gap between data lakes and RDBMS based solutions. By leveraging the metadata layer, lakehouses implement ACID transactions, schema enforcement and data quality assurances, through the implementation of constraints and table relationships, as well as governance features, such as access control and audit logging [Armb21]. Lakehouse's reliance on open file formats preserves the ability to integrate lakehouse specific and non-specific systems [Mazu23], while also preserving the ease of migration that characterises data lakes. The combined use of open file formats and metadata facilitate schema evolution over time, without bearing the cost of rewriting entire tables [Mazu23]. Finally, through SQL specific optimisations, such as data caching, auxiliary data collection, which enables features such as data skipping and partition pruning and data layout improvements, such as indexing, lakehouses enhance BI and analytic query performance.

Based on the batch processing benchmarks, we conclude that the performance of Delta and Hudi is comparable to that of Spark querying a plain data lake. Both systems scale effectively, both in terms of the cluster scaling out and handling growing datasets. Interestingly, for larger datasets, Delta and Hudi even outperform Spark. Lakehouse specific optimisations, such as data skipping and partition pruning, especially when used with indexed data, significantly impact performance, enabling Delta and Hudi to consistently outperform Spark.

From the results of the stream processing benchmarks, we conclude that, while Spark generally outperforms Delta and Hudi during ingestion, primarily due to the additional overhead associated with metadata management, all three systems demonstrate comparable performance during query execution.

Overall, Delta Lake and Hudi yield impressive results, exhibiting performance comparable to current solutions. The extensive range of features, designed to address common challenges and optimise conventional use cases, justifies their adoption, even in scenarios where they may slightly underperform.

5.1 Future Directions

In the final section we explore further research that could be conducted as a continuation to this thesis.

5.1.1 Dynamic Workloads and Datasets

An area not addressed in this thesis is the handling of unclean or inconsistent datasets. Lakehouse features, such as schema enforcement and schema evolution, could be benchmarked and compared against traditional data cleansing approaches, such as ETL pipelines or batch cleaning. Similarily, benchmarking the systems with dynamic workloads, characterised by data skews or fluctuating batch sizes, could provide further insights into how each system operates under stress conditions and back-pressure.

5.1.2 Multi-Engine Support and Query Federation

Contemporary big data management architectures often involve assembling complex platforms comprising different engines and data sources. This has lead to the emergence of query federation systems, which unify individually operating platforms, under a common schema, enabling querying both relational and non-relational data, using an SQL interface. Presto[1] and Trino[2] are widely adopted examples of such solutions, already offering connectors for Delta Lake, Hudi, as well as Apache Iceberg[3], another widespread lakehouse format. To further support interoperability, Delta Lake has even introduced the Universal Format (UniForm)[4], which enables Hudi and Iceberg to read Delta tables.

Furthermore, Delta and Hudi support integration with a variety of batch and stream processing engines, including Apache Storm[5] and Apache Flink[6], the latter being deeply integrated with Hudi. Benchmarking lakehouse systems with different engines could provide more granular insights about the most effective platform combinations depending on the use case.

5.1.3 Traditional RDBMS

Lakehouse systems incorporate a wide range of features traditionally supported by RDBMSs, such as ACID transactions, indexing and other SQL specific optimisations. Through continuous development and improvement, RDBMSs, such as PostgreSQL[7], have evolved to be able to handle big data workloads effectively. As a result, benchmarking lakehouse architectures against big data optimised RDBMSs could help assess their relative interchangeability and suitability for different use cases.

5.1.4 Machine Learning Applications

Throughout this thesis, we have repeatedly mentioned the characteristics that make lakehouses better suited for ML workloads compared to traditional data lake architectures. Delta and Hudi can serve as a reliable storage layer with systems such as TensorFlow[8] and PyTorch[9], in order to evaluate how lakehouses, with features that guarantee data quality and governance, compare to traditional ETL pipelines for ML.

[1] Presto: `https://prestodb.io/`
[2] Trino: `https://trino.io/`
[3] Apache Iceberg: `https://iceberg.apache.org/`
[4] Delta Lake's UniForm: `https://docs.delta.io/latest/delta-uniform.html`
[5] Apache Storm: `https://storm.apache.org/`
[6] Apache Flink: `https://flink.apache.org/`
[7] PostgreSQL: `https://www.postgresql.org/`
[8] TensorFlow: `https://www.tensorflow.org/`
[9] PyTorch: `https://pytorch.org/`

Βιβλιογραφία (References)

[Amaz25] Amazon, “What is Apache Spark?”, https://aws.amazon.com/what-is/apache-spark/, 2025. Accessed: 2025-06-12.

[Armb21] Michael Armbrust, Ali Ghodsi, Reynold Xin and Matei Zaharia, “Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics”, in *Proceedings of the 11th Annual Conference on Innovative Data Systems Research (CIDR ’21)*, January 2021.

[Aysh24] Avril Aysha, “Delta Lake vs Data Lake - What’s the Difference?”, https://delta.io/blog/delta-lake-vs-data-lake/, May 2024. Accessed: 2025-06-11.

[Benj20] Sarah Benjelloun, Mohamed El Mehdi El Aissi, Yassine Loukili, Younes Lakhrissi, Safae Elhaj Ben Ali, Hiba Chougrad and Abdessamad El Boushaki, “Big Data Processing: Batch-based processing and stream-based processing”, in *Proceedings of the 4th International Conference On Intelligent Computing in Data Sciences (ICDS’ 20)*, pp. 1–6, 2020.

[Chen17] Lingjiao Chen, Arun Kumar, Jeffrey Naughton and Jignesh M. Patel, “Towards linear algebra over normalized data”, *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1214—1225, August 2017.

[Data24] Databricks, “What is the medallion lakehouse architecture?”, https://docs.databricks.com/aws/en/lakehouse/medallion, 2024. Accessed: 2025-05-27.

[Data25] Databricks, “What is Hadoop Distributed File System (HDFS)?”, https://www.databricks.com/glossary/hadoop-distributed-file-system-hdfs, 2025. Accessed: 2025-06-12.

[Drem25] Dremio, “What Is Apache Hudi?”, https://www.dremio.com/wiki/apache-hudi/, 2025. Accessed: 2025-06-12.

[Hai23] Rihan Hai, Christos Koutras, Christoph Quix and Matthias Jarke, “Data Lakes: A Survey of Functions and Systems”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 12, pp. 12571–12590, 2023.

[Herd20] Olaf Herden, “Architectural Patterns for Integrating Data Lakes into Data Warehouse Architectures”, in *Proceedings of the 8th International Conference on Big Data Analytics (ICBDA ’20)’*, pp. 12–27, Springer International Publishing, December 2020.

[Hudi24] Apache Hudi, “Apache Hudi Documentation: Use Cases.”, https://hudi.apache.org/docs/0.14.1/use_cases, 2024. Accessed: 2025-06-11.

[IBM25] IBM, “What is Delta Lake?”, https://www.ibm.com/think/topics/delta-lake, 2025. Accessed: 2025-06-12.

[Jain23] Paras Jain, Peter Kraft, Conor Power, Tathagata Das, Ion Stoica and Matei Zaharia, “Analyzing and Comparing Lakehouse Storage Systems”, in *Proceedings of the 13th Annual Conference on Innovative Data Systems Research (CIDR ’23)*, January 2023.

[Kari18] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen and Volker Markl, “Benchmarking Distributed Stream Data Processing Systems”, in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 1507–1518, 2018.

[Kham17] Mahmoud Abo Khamis, Hung Quoc Ngo, XuanLong Nguyen, Dan Olteanu and Maximilian Schleich, “Learning Models over Relational Data Using Sparse Tensors and Functional Dependencies”, *ACM Transactions on Database Systems (TODS)*, vol. 45, pp. 1–66, 2017.

[Makr21] Nantia Makrynioti and Vasilis Vassalos Vasilis, “Declarative Data Analytics: A Survey”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 6, pp. 2392–2411, 2021.

[Mazu23] Dipankar Mazumdar, Jason Hughes and JB Onofre, “The Data Lakehouse: Data Warehousing and More”, arXiv:2310.08697 [cs.DB], 2023.

[Silb11] Abraham Silberschatz, Henry F. Korth and S. Sudrashan, *Database System Concepts*, pp. 889–891, McGraw-Hill, New York, 6th edition, 2011.

[Tran21] Transaction Processing Performance Council (TPC), *TPC-DS Standard Specification*, June 2021. Version 3.2.0.

[Zhao17] Xinwei Zhao, Saurabh Garg, Carlos Queiroz and Rajkumar Buyya, “Chapter 11 - A Taxonomy and Survey of Stream Processing Systems”, in *Software Architecture for Big Data and the Cloud*, pp. 183–206, Morgan Kaufmann, Boston, 2017.

Παράρτημα (Appendix) A

TPC-DS Queries

Listing A.1: TPC-DS Query 9.

```
select case when (select count(*)
                  from store_sales
                  where ss_quantity between 1 and 20) > 25437
            then (select avg(ss_ext_discount_amt)
                  from store_sales
                  where ss_quantity between 1 and 20)
            else (select avg(ss_net_profit)
                  from store_sales
                  where ss_quantity between 1 and 20)
                  end bucket1 ,
       case when (select count(*)
                  from store_sales
                  where ss_quantity between 21 and 40) > 22746
            then (select avg(ss_ext_discount_amt)
                  from store_sales
                  where ss_quantity between 21 and 40)
            else (select avg(ss_net_profit)
                  from store_sales
                  where ss_quantity between 21 and 40)
                  end bucket2 ,
       case when (select count(*)
                  from store_sales
                  where ss_quantity between 41 and 60) > 9387
            then (select avg(ss_ext_discount_amt)
                  from store_sales
                  where ss_quantity between 41 and 60)
            else (select avg(ss_net_profit)
                  from store_sales
                  where ss_quantity between 41 and 60)
                  end bucket3 ,
       case when (select count(*)
                  from store_sales
                  where ss_quantity between 61 and 80) > 10098
            then (select avg(ss_ext_discount_amt)
                  from store_sales
                  where ss_quantity between 61 and 80)
            else (select avg(ss_net_profit)
                  from store_sales
                  where ss_quantity between 61 and 80)
```

```
                 end bucket4 ,
      case when (select count(*)
                 from store_sales
                 where ss_quantity between 81 and 100) > 18213
           then (select avg(ss_ext_discount_amt)
                 from store_sales
                 where ss_quantity between 81 and 100)
           else (select avg(ss_net_profit)
                 from store_sales
                 where ss_quantity between 81 and 100)
                 end bucket5
from reason
where r_reason_sk = 1;
```

Listing A.2: TPC-DS Query 11.

```
with year_total as (
 select c_customer_id customer_id
       ,c_first_name customer_first_name
       ,c_last_name customer_last_name
       ,c_preferred_cust_flag customer_preferred_cust_flag
       ,c_birth_country customer_birth_country
       ,c_login customer_login
       ,c_email_address customer_email_address
       ,d_year dyear
       ,sum(ss_ext_list_price-ss_ext_discount_amt) year_total
       ,'s' sale_type
 from customer
     ,store_sales
     ,date_dim
 where c_customer_sk = ss_customer_sk
   and ss_sold_date_sk = d_date_sk
 group by c_customer_id
         ,c_first_name
         ,c_last_name
         ,c_preferred_cust_flag
         ,c_birth_country
         ,c_login
         ,c_email_address
         ,d_year
 union all
 select c_customer_id customer_id
       ,c_first_name customer_first_name
       ,c_last_name customer_last_name
       ,c_preferred_cust_flag customer_preferred_cust_flag
       ,c_birth_country customer_birth_country
       ,c_login customer_login
       ,c_email_address customer_email_address
       ,d_year dyear
       ,sum(ws_ext_list_price-ws_ext_discount_amt) year_total
       ,'w' sale_type
 from customer
```

```
      ,web_sales
      ,date_dim
 where c_customer_sk = ws_bill_customer_sk
   and ws_sold_date_sk = d_date_sk
 group by c_customer_id
         ,c_first_name
         ,c_last_name
         ,c_preferred_cust_flag
         ,c_birth_country
         ,c_login
         ,c_email_address
         ,d_year
         )
  select
                  t_s_secyear.customer_id
                 ,t_s_secyear.customer_first_name
                 ,t_s_secyear.customer_last_name
                 ,t_s_secyear.customer_email_address
 from year_total t_s_firstyear
     ,year_total t_s_secyear
     ,year_total t_w_firstyear
     ,year_total t_w_secyear
 where t_s_secyear.customer_id = t_s_firstyear.customer_id
         and t_s_firstyear.customer_id = t_w_secyear.customer_id
         and t_s_firstyear.customer_id = t_w_firstyear.customer_id
         and t_s_firstyear.sale_type = 's'
         and t_w_firstyear.sale_type = 'w'
         and t_s_secyear.sale_type = 's'
         and t_w_secyear.sale_type = 'w'
         and t_s_firstyear.dyear = 2001
         and t_s_secyear.dyear = 2001+1
         and t_w_firstyear.dyear = 2001
         and t_w_secyear.dyear = 2001+1
         and t_s_firstyear.year_total > 0
         and t_w_firstyear.year_total > 0
         and case when t_w_firstyear.year_total > 0
         then t_w_secyear.year_total/t_w_firstyear.year_total
         else 0.0 end
         case when t_s_firstyear.year_total > 0
         then t_s_secyear.year_total/t_s_firstyear.year_total
        else 0.0 end
 order by t_s_secyear.customer_id
         ,t_s_secyear.customer_first_name
         ,t_s_secyear.customer_last_name
         ,t_s_secyear.customer_email_address
limit 100;
```

Listing A.3: TPC-DS Query 28.

```
select  *
from (select avg(ss_list_price) B1_LP
            ,count(ss_list_price) B1_CNT
```

```
             ,count(distinct ss_list_price) B1_CNTD
      from store_sales
      where ss_quantity between 0 and 5
        and (ss_list_price between 11 and 11+10
             or ss_coupon_amt between 460 and 460+1000
             or ss_wholesale_cost between 14 and 14+20)) B1,
     (select avg(ss_list_price) B2_LP
             ,count(ss_list_price) B2_CNT
             ,count(distinct ss_list_price) B2_CNTD
      from store_sales
      where ss_quantity between 6 and 10
        and (ss_list_price between 91 and 91+10
          or ss_coupon_amt between 1430 and 1430+1000
          or ss_wholesale_cost between 32 and 32+20)) B2,
     (select avg(ss_list_price) B3_LP
             ,count(ss_list_price) B3_CNT
             ,count(distinct ss_list_price) B3_CNTD
      from store_sales
      where ss_quantity between 11 and 15
        and (ss_list_price between 66 and 66+10
          or ss_coupon_amt between 920 and 920+1000
          or ss_wholesale_cost between 4 and 4+20)) B3,
     (select avg(ss_list_price) B4_LP
             ,count(ss_list_price) B4_CNT
             ,count(distinct ss_list_price) B4_CNTD
      from store_sales
      where ss_quantity between 16 and 20
        and (ss_list_price between 142 and 142+10
          or ss_coupon_amt between 3054 and 3054+1000
          or ss_wholesale_cost between 80 and 80+20)) B4,
     (select avg(ss_list_price) B5_LP
             ,count(ss_list_price) B5_CNT
             ,count(distinct ss_list_price) B5_CNTD
      from store_sales
      where ss_quantity between 21 and 25
        and (ss_list_price between 135 and 135+10
          or ss_coupon_amt between 14180 and 14180+1000
          or ss_wholesale_cost between 38 and 38+20)) B5,
     (select avg(ss_list_price) B6_LP
             ,count(ss_list_price) B6_CNT
             ,count(distinct ss_list_price) B6_CNTD
      from store_sales
      where ss_quantity between 26 and 30
        and (ss_list_price between 28 and 28+10
          or ss_coupon_amt between 2513 and 2513+1000
          or ss_wholesale_cost between 42 and 42+20)) B6
limit 100;
```

Listing A.4: TPC-DS Query 33.

```
with ss as (
 select
```

```
          i_manufact_id ,sum( ss_ext_sales_price )  total_sales
 from
         store_sales ,
         date_dim ,
          customer_address ,
          item
 where
          i_manufact_id  in  ( select
  i_manufact_id
from
 item
where  i_category  in  ( 'Books'))
 and       ss_item_sk                   =  i_item_sk
 and       ss_sold_date_sk              =  d_date_sk
 and       d_year                       =  1999
 and       d_moy                        =  3
 and       ss_addr_sk                   =  ca_address_sk
 and       ca_gmt_offset                =  -5
 group  by  i_manufact_id ),
 cs  as  (
 select
          i_manufact_id ,sum( cs_ext_sales_price )  total_sales
 from
         catalog_sales ,
         date_dim ,
          customer_address ,
          item
 where
          i_manufact_id                 in  ( select
  i_manufact_id
from
 item
where  i_category  in  ( 'Books'))
 and       cs_item_sk                   =  i_item_sk
 and       cs_sold_date_sk              =  d_date_sk
 and       d_year                       =  1999
 and       d_moy                        =  3
 and       cs_bill_addr_sk              =  ca_address_sk
 and       ca_gmt_offset                =  -5
 group  by  i_manufact_id ),
 ws  as  (
 select
          i_manufact_id ,sum( ws_ext_sales_price )  total_sales
 from
         web_sales ,
         date_dim ,
          customer_address ,
          item
 where
          i_manufact_id                 in  ( select
  i_manufact_id
from
 item
where i_category in ('Books'))
 and      ws_item_sk              = i_item_sk
 and      ws_sold_date_sk         = d_date_sk
 and      d_year                  = 1999
 and      d_moy                   = 3
 and      ws_bill_addr_sk         = ca_address_sk
 and      ca_gmt_offset           = -5
 group by i_manufact_id)
  select  i_manufact_id ,sum(total_sales) total_sales
 from  (select * from ss
        union all
        select * from cs
        union all
        select * from ws) tmp1
 group by i_manufact_id
 order by total_sales
limit 100;
```

Listing A.5: TPC-DS Query 35.

```
select
  ca_state ,
  cd_gender ,
  cd_marital_status ,
  cd_dep_count ,
  count(*) cnt1 ,
  avg(cd_dep_count),
  max(cd_dep_count),
  sum(cd_dep_count),
  cd_dep_employed_count ,
  count(*) cnt2 ,
  avg(cd_dep_employed_count),
  max(cd_dep_employed_count),
  sum(cd_dep_employed_count),
  cd_dep_college_count ,
  count(*) cnt3 ,
  avg(cd_dep_college_count),
  max(cd_dep_college_count),
  sum(cd_dep_college_count)
 from
  customer c,customer_address ca,customer_demographics
 where
  c.c_current_addr_sk = ca.ca_address_sk and
  cd_demo_sk = c.c_current_cdemo_sk and
  exists (select *
          from store_sales,date_dim
          where c.c_customer_sk = ss_customer_sk and
                ss_sold_date_sk = d_date_sk and
                d_year = 1999 and
                d_qoy < 4) and
```

```
   (exists (select *
            from web_sales,date_dim
            where c.c_customer_sk = ws_bill_customer_sk and
                  ws_sold_date_sk = d_date_sk and
                  d_year = 1999 and
                  d_qoy < 4) or
    exists (select *
            from catalog_sales,date_dim
            where c.c_customer_sk = cs_ship_customer_sk and
                  cs_sold_date_sk = d_date_sk and
                  d_year = 1999 and
                  d_qoy < 4))
 group by ca_state,
          cd_gender,
          cd_marital_status,
          cd_dep_count,
          cd_dep_employed_count,
          cd_dep_college_count
 order by ca_state,
          cd_gender,
          cd_marital_status,
          cd_dep_count,
          cd_dep_employed_count,
          cd_dep_college_count
 limit 100;
```

Listing A.6: TPC-DS Query 38.

```
select  count(*) from (
    select distinct c_last_name, c_first_name, d_date
    from store_sales, date_dim, customer
    where store_sales.ss_sold_date_sk = date_dim.d_date_sk
    and store_sales.ss_customer_sk = customer.c_customer_sk
    and d_month_seq between 1212 and 1212 + 11
  intersect
    select distinct c_last_name, c_first_name, d_date
    from catalog_sales, date_dim, customer
    where catalog_sales.cs_sold_date_sk = date_dim.d_date_sk
    and catalog_sales.cs_bill_customer_sk = customer.c_customer_sk
    and d_month_seq between 1212 and 1212 + 11
  intersect
    select distinct c_last_name, c_first_name, d_date
    from web_sales, date_dim, customer
    where web_sales.ws_sold_date_sk = date_dim.d_date_sk
    and web_sales.ws_bill_customer_sk = customer.c_customer_sk
    and d_month_seq between 1212 and 1212 + 11
) hot_cust
limit 100;
```

Listing A.7: TPC-DS Query 61.

```
select  promotions,total,
        cast(promotions as decimal(15,4))/
```

```
         cast(total as decimal(15,4))*100
from
  (select sum(ss_ext_sales_price) promotions
   from  store_sales
        ,store
        ,promotion
        ,date_dim
        ,customer
        ,customer_address
        ,item
   where ss_sold_date_sk = d_date_sk
   and   ss_store_sk = s_store_sk
   and   ss_promo_sk = p_promo_sk
   and   ss_customer_sk= c_customer_sk
   and   ca_address_sk = c_current_addr_sk
   and   ss_item_sk = i_item_sk
   and   ca_gmt_offset = -7
   and   i_category = 'Books'
   and   (p_channel_dmail = 'Y'
          or p_channel_email = 'Y'
          or p_channel_tv = 'Y')
   and   s_gmt_offset = -7
   and   d_year = 1999
   and   d_moy  = 11) promotional_sales,
  (select sum(ss_ext_sales_price) total
   from  store_sales
        ,store
        ,date_dim
        ,customer
        ,customer_address
        ,item
   where ss_sold_date_sk = d_date_sk
   and   ss_store_sk = s_store_sk
   and   ss_customer_sk= c_customer_sk
   and   ca_address_sk = c_current_addr_sk
   and   ss_item_sk = i_item_sk
   and   ca_gmt_offset = -7
   and   i_category = 'Books'
   and   s_gmt_offset = -7
   and   d_year = 1999
   and   d_moy  = 11) all_sales
order by promotions, total
limit 100;
```

Listing A.8: TPC-DS Query 64.

```
with cs_ui as
 (select cs_item_sk,
         sum(cs_ext_list_price)
         as sale,
         sum(cr_refunded_cash+cr_reversed_charge+cr_store_credit)
         as refund
```

```
  from catalog_sales
      ,catalog_returns
  where cs_item_sk = cr_item_sk
    and cs_order_number = cr_order_number
  group by cs_item_sk
  having sum(cs_ext_list_price)>2*sum(cr_refunded_cash
                                      +cr_reversed_charge
                                      +cr_store_credit)),
cross_sales as
 (select i_product_name product_name
     ,i_item_sk item_sk
     ,s_store_name store_name
     ,s_zip store_zip
     ,ad1.ca_street_number b_street_number
     ,ad1.ca_street_name b_street_name
     ,ad1.ca_city b_city
     ,ad1.ca_zip b_zip
     ,ad2.ca_street_number c_street_number
     ,ad2.ca_street_name c_street_name
     ,ad2.ca_city c_city
     ,ad2.ca_zip c_zip
     ,d1.d_year as syear
     ,d2.d_year as fsyear
     ,d3.d_year s2year
     ,count(*) cnt
     ,sum(ss_wholesale_cost) s1
     ,sum(ss_list_price) s2
     ,sum(ss_coupon_amt) s3
  FROM   store_sales
        ,store_returns
        ,cs_ui
        ,date_dim d1
        ,date_dim d2
        ,date_dim d3
        ,store
        ,customer
        ,customer_demographics cd1
        ,customer_demographics cd2
        ,promotion
        ,household_demographics hd1
        ,household_demographics hd2
        ,customer_address ad1
        ,customer_address ad2
        ,income_band ib1
        ,income_band ib2
        ,item
  WHERE  ss_store_sk = s_store_sk AND
         ss_sold_date_sk = d1.d_date_sk AND
         ss_customer_sk = c_customer_sk AND
         ss_cdemo_sk= cd1.cd_demo_sk AND
         ss_hdemo_sk = hd1.hd_demo_sk AND
```

```
          ss_addr_sk = ad1.ca_address_sk and
          ss_item_sk = i_item_sk and
          ss_item_sk = sr_item_sk and
          ss_ticket_number = sr_ticket_number and
          ss_item_sk = cs_ui.cs_item_sk and
          c_current_cdemo_sk = cd2.cd_demo_sk AND
          c_current_hdemo_sk = hd2.hd_demo_sk AND
          c_current_addr_sk = ad2.ca_address_sk and
          c_first_sales_date_sk = d2.d_date_sk and
          c_first_shipto_date_sk = d3.d_date_sk and
          ss_promo_sk = p_promo_sk and
          hd1.hd_income_band_sk = ib1.ib_income_band_sk and
          hd2.hd_income_band_sk = ib2.ib_income_band_sk and
          cd1.cd_marital_status <> cd2.cd_marital_status and
          i_color in ('maroon','burnished','dim'
                     ,'steel','navajo','chocolate') and
          i_current_price between 35 and 35 + 10 and
          i_current_price between 35 + 1 and 35 + 15
group by i_product_name
        ,i_item_sk
        ,s_store_name
        ,s_zip
        ,ad1.ca_street_number
        ,ad1.ca_street_name
        ,ad1.ca_city
        ,ad1.ca_zip
        ,ad2.ca_street_number
        ,ad2.ca_street_name
        ,ad2.ca_city
        ,ad2.ca_zip
        ,d1.d_year
        ,d2.d_year
        ,d3.d_year
)
select cs1.product_name
      ,cs1.store_name
      ,cs1.store_zip
      ,cs1.b_street_number
      ,cs1.b_street_name
      ,cs1.b_city
      ,cs1.b_zip
      ,cs1.c_street_number
      ,cs1.c_street_name
      ,cs1.c_city
      ,cs1.c_zip
      ,cs1.syear
      ,cs1.cnt
      ,cs1.s1 as s11
      ,cs1.s2 as s21
      ,cs1.s3 as s31
      ,cs2.s1 as s12
      ,cs2.s2 as s22
      ,cs2.s3 as s32
      ,cs2.syear
      ,cs2.cnt
from cross_sales cs1,cross_sales cs2
where cs1.item_sk=cs2.item_sk and
      cs1.syear = 2000 and
      cs2.syear = 2000 + 1 and
      cs2.cnt <= cs1.cnt and
      cs1.store_name = cs2.store_name and
      cs1.store_zip = cs2.store_zip
order by cs1.product_name
        ,cs1.store_name
        ,cs2.cnt
        ,cs1.s1
        ,cs2.s1;
```

Listing A.9: TPC-DS Query 72.

```
select  i_item_desc
       ,w_warehouse_name
       ,d1.d_week_seq
       ,sum(case when p_promo_sk is null
            then 1 else 0 end) no_promo
       ,sum(case when p_promo_sk is not null
            then 1 else 0 end) promo
       ,count(*) total_cnt
from catalog_sales
join inventory on (cs_item_sk = inv_item_sk)
join warehouse on (w_warehouse_sk=inv_warehouse_sk)
join item on (i_item_sk = cs_item_sk)
join customer_demographics on (cs_bill_cdemo_sk = cd_demo_sk)
join household_demographics on (cs_bill_hdemo_sk = hd_demo_sk)
join date_dim d1 on (cs_sold_date_sk = d1.d_date_sk)
join date_dim d2 on (inv_date_sk = d2.d_date_sk)
join date_dim d3 on (cs_ship_date_sk = d3.d_date_sk)
left outer join promotion on (cs_promo_sk=p_promo_sk)
left outer join catalog_returns
     on (cr_item_sk = cs_item_sk
     and cr_order_number = cs_order_number)
where d1.d_week_seq = d2.d_week_seq
   and inv_quantity_on_hand < cs_quantity
   and d3.d_date > d1.d_date + 5
   and hd_buy_potential = '1001-5000'
   and d1.d_year = 2001
   and cd_marital_status = 'M'
group by i_item_desc,w_warehouse_name,d1.d_week_seq
order by total_cnt desc, i_item_desc,
         w_warehouse_name, d_week_seq
limit 100;
```

Listing A.10: TPC-DS Query 75.

```
WITH all_sales AS (
 SELECT d_year
        ,i_brand_id
        ,i_class_id
        ,i_category_id
        ,i_manufact_id
        ,SUM(sales_cnt) AS sales_cnt
        ,SUM(sales_amt) AS sales_amt
 FROM (SELECT d_year
          ,i_brand_id
          ,i_class_id
          ,i_category_id
          ,i_manufact_id
          ,cs_quantity-COALESCE(cr_return_quantity,0)
          AS sales_cnt
          ,cs_ext_sales_price-COALESCE(cr_return_amount,0.0)
          AS sales_amt
       FROM catalog_sales JOIN item ON i_item_sk=cs_item_sk
            JOIN date_dim ON d_date_sk=cs_sold_date_sk
            LEFT JOIN catalog_returns
            ON (cs_order_number=cr_order_number
            AND cs_item_sk=cr_item_sk)
       WHERE i_category='Sports'
       UNION
       SELECT d_year
              ,i_brand_id
              ,i_class_id
              ,i_category_id
              ,i_manufact_id
              ,ss_quantity-COALESCE(sr_return_quantity,0)
              AS sales_cnt
              ,ss_ext_sales_price-COALESCE(sr_return_amt,0.0)
              AS sales_amt
       FROM store_sales JOIN item ON i_item_sk=ss_item_sk
            JOIN date_dim ON d_date_sk=ss_sold_date_sk
            LEFT JOIN store_returns
            ON (ss_ticket_number=sr_ticket_number
            AND ss_item_sk=sr_item_sk)
       WHERE i_category='Sports'
       UNION
       SELECT d_year
              ,i_brand_id
              ,i_class_id
              ,i_category_id
              ,i_manufact_id
              ,ws_quantity-COALESCE(wr_return_quantity,0)
              AS sales_cnt
              ,ws_ext_sales_price-COALESCE(wr_return_amt,0.0)
              AS sales_amt
       FROM web_sales JOIN item ON i_item_sk=ws_item_sk
            JOIN date_dim ON d_date_sk=ws_sold_date_sk
```

```
                LEFT JOIN web_returns
                ON (ws_order_number=wr_order_number
                AND ws_item_sk=wr_item_sk)
          WHERE i_category='Sports') sales_detail
GROUP BY d_year, i_brand_id, i_class_id,
         i_category_id, i_manufact_id)
SELECT  prev_yr.d_year AS prev_year,
        curr_yr.d_year AS year,
        curr_yr.i_brand_id,
        curr_yr.i_class_id,
        curr_yr.i_category_id,
        curr_yr.i_manufact_id,
        prev_yr.sales_cnt AS prev_yr_cnt,
        curr_yr.sales_cnt AS curr_yr_cnt,
        curr_yr.sales_cnt-prev_yr.sales_cnt AS sales_cnt_diff,
        curr_yr.sales_amt-prev_yr.sales_amt AS sales_amt_diff,
FROM all_sales curr_yr, all_sales prev_yr
WHERE curr_yr.i_brand_id=prev_yr.i_brand_id
  AND curr_yr.i_class_id=prev_yr.i_class_id
  AND curr_yr.i_category_id=prev_yr.i_category_id
  AND curr_yr.i_manufact_id=prev_yr.i_manufact_id
  AND curr_yr.d_year=2002
  AND prev_yr.d_year=2002-1
  AND CAST(curr_yr.sales_cnt AS DECIMAL(17,2))/
      CAST(prev_yr.sales_cnt AS DECIMAL(17,2))<0.9
ORDER BY sales_cnt_diff,sales_amt_diff
limit 100;
```