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

Συγκριτική ανάλυση των αρχιτεκτονικών Data Lake: Apache Iceberg, Apache Hudi και Delta Lake

Διπλωματική Εργασία

της

ΕΛΕΝΗΣ ΤΣΕΡΤΟΥ

Επιβλέπων: Κοζύρης Νεκτάριος
Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2025

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

Συγκριτική ανάλυση των αρχιτεκτονικών Data Lake: Apache Iceberg, Apache Hudi και Delta Lake

Διπλωματική Εργασία

της

ΕΛΕΝΗΣ ΤΣΕΡΤΟΥ

Επιβλέπων: Κοζύρης Νεκτάριος
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 3η Νοεμβρίου 2025.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Κοζύρης Νεκτάριος	Κωνσταντίνου Ιωάννης	Τσουμάκος Δημήτριος
Καθηγητής Ε.Μ.Π.	Αναπληρωτής Καθηγητής Παν. Θεσσαλίας	Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2025

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

ΔΗΛΩΣΗ ΜΗ ΛΟΓΟΚΛΟΠΗΣ ΚΑΙ ΑΝΑΛΗΨΗΣ ΠΡΟΣΩΠΙΚΗΣ ΕΥΘΥΝΗΣ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ενυπογράφως ότι είμαι αποκλειστικός συγγραφέας της παρούσας Πτυχιακής Εργασίας, για την ολοκλήρωση της οποίας κάθε βοήθεια είναι πλήρως αναγνωρισμένη και αναφέρεται λεπτομερώς στην εργασία αυτή. Έχω αναφέρει πλήρως και με σαφείς αναφορές, όλες τις πηγές χρήσης δεδομένων, απόψεων, θέσεων και προτάσεων, ιδεών και λεκτικών αναφορών, είτε κατά κυριολεξία είτε βάσει επιστημονικής παράφρασης. Αναλαμβάνω την προσωπική και ατομική ευθύνη ότι σε περίπτωση αποτυχίας στην υλοποίηση των ανωτέρω δηλωθέντων στοιχείων, είμαι υπόλογος έναντι λογοκλοπής, γεγονός που σημαίνει αποτυχία στην Πτυχιακή μου Εργασία και κατά συνέπεια αποτυχία απόκτησης του Τίτλου Σπουδών, πέραν των λοιπών συνεπειών του νόμου περί πνευματικών δικαιωμάτων. Δηλώνω, συνεπώς, ότι αυτή η Πτυχιακή Εργασία προετοιμάστηκε και ολοκληρώθηκε από εμένα προσωπικά και αποκλειστικά και ότι, αναλαμβάνω πλήρως όλες τις συνέπειες του νόμου στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής άλλης πνευματικής ιδιοκτησίας.

(Υπογραφή)

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

3η Νοεμβρίου 2025

Περίληψη

Η ραγδαία αύξηση των δεδομένων τα τελευταία χρόνια έχει οδηγήσει σε σημαντική αλλαγή στον τρόπο με τον οποίο οι οργανισμοί αποθηκεύουν, διαχειρίζονται και αναλύουν τις πληροφορίες. Τα παραδοσιακά αποθετήρια δεδομένων, αν και αποτελεσματικά για δομημένα δεδομένα, δυσκολεύονται να ανταποκριθούν στις απαιτήσεις των σύγχρονων συνόλων δεδομένων που χαρακτηρίζονται από μεγάλο όγκο, ποικιλία και ταχύτητα. Για την αντιμετώπιση αυτών των προκλήσεων, εμφανίστηκαν τα data lakes ως κεντρικά αποθετήρια ικανά να αποθηκεύουν ακατέργαστα δεδομένα σε τεράστια κλίμακα. Ωστόσο, αυτές οι αρχικές αρχιτεκτονικές παρουσίαζαν περιορισμούς, όπως χαμηλή απόδοση ερωτημάτων, έλλειψη εγγυήσεων συναλλαγών και απουσία επιβολής σχήματος.

Για την αντιμετώπιση αυτών των ελλείψεων, έχει αναπτυχθεί μια νέα γενιά ανοιχτών μορφών πίνακα, συγκεκριμένα τα Apache Iceberg, Apache Hudi και Delta Lake. Αυτές οι τεχνολογίες εισάγουν χαρακτηριστικά συστημάτων σχεσιακών βάσεων δεδομένων στα data lakes, όπως μεταβολές σχήματος, συναλλαγές ACID, ερωτήματα time travel και αποτελεσματική διαχείριση αρχείων.

Η παρούσα εργασία επιχειρεί μια συγκριτική αξιολόγηση της απόδοσης και της κλιμάκωσης των Apache Iceberg, Apache Hudi και Delta Lake. Χρησιμοποιώντας το πλαίσιο LST-Bench, η μελέτη εκτελεί μια σειρά πειραμάτων benchmarking σε διάφορους τύπους φόρτιου εργασίας για να αξιολογήσει τις δυνατότητες απόδοσης και κλιμάκωσης κάθε τεχνολογίας στο πλαίσιο μιας αρχιτεκτονικής data lakehouse. Τα ευρήματα στοχεύουν να προσφέρουν πληροφορίες σχετικά με την καταλληλότητά τους για διαφορετικά σενάρια ανάλυσης δεδομένων.

Λέξεις Κλειδιά

Δομές Πινάκων με Καταγραφή, Λίμνες Δεδομένων, Διαχείριση Μεταδεδομένων, Ανάλυση Απόδοσης, Αξιολόγηση μέσω Δοκιμών, Μοτίβα Φορτίου Εργασίας

Abstract

The rapid proliferation of data in recent years has driven a significant shift in how organizations store, manage, and analyze information. Traditional data warehouses, while effective for structured data, struggle to meet the demands of modern datasets characterized by high volume, variety, and velocity. In response to these challenges, data lakes emerged as centralized repositories capable of storing raw data at a massive scale. However, these initial architectures suffered from limitations such as poor query performance, lack of transactional guarantees, and the absence of schema enforcement.

To address these shortcomings, a new generation of open table formats—specifically Apache Iceberg, Apache Hudi, and Delta Lake—has been developed. These technologies introduce database-like functionalities to data lakes, including schema evolution, ACID transactions, time-travel queries, and efficient file management over object storage.

This thesis provides a comparative evaluation of the performance and scalability of Apache Iceberg, Apache Hudi, and Delta Lake. Using the LST-Bench framework, the study executes a series of benchmark experiments across various workload types to assess the performance and scalability of each technology within a data lakehouse architecture. The findings aim to offer insights into their suitability for different data engineering and analytics scenarios.

Keywords

Log-Structured Table Formats, Data Lakes, Metadata Management, Performance Analysis, Benchmarking, Workload Patterns

Ευχαριστίες

Με την εκπόνηση της παρούσης διπλωματικής εργασίας, ολοκληρώνεται ο κύκλος των σπουδών μου στη σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου.

Ευχαριστώ τον Χριστό και την Παναγία.

Ευχαριστώ την οικογένεια μου που είναι δίπλα μου και με στηρίζει σε κάθε μου βήμα. Ευχαριστώ τους γονείς μου Αικατερίνη Κριθινάκη και Ξενοφών Τσέρτο. Ευχαριστώ την αδελφή μου, Αθανασία Τσέρτου.

Ευχαριστώ τον κ. Ιωαννή Κωνσταντίνου για την βοήθεια και την καθοδήγησή του καθ' όλη την διάρκεια της διπλωματικής μου εργασίας.

Ευχαριστώ ιδιαίτερα την φίλη μου Άρτεμις Ανδρώνη, για όλο το ταξίδι που μοιραστήκαμε μαζί όλα αυτά τα χρόνια σε αυτή την σχολή.

Ευχαριστώ τον φίλο μου Πάρη Σάλαρη για το ειλικρινές ενδιαφέρον του και την ψυχολογική στήριξη που μου προσέφερε όλο αυτόν τον καιρό.

Ένα ιδιαίτερο ευχαριστώ στον φίλο και συνεργάτη Γιάννη Οικονόμου για την πίστη του στις δυνατότητες μου και για την πολύτιμη συμβολή του με ιδέες σε κρίσιμα σημεία της εργασίας.

Ευχαριστώ όλους τους ανθρώπους που με στήριξαν σε αυτό το ταξίδι και μου έδωσαν την δύναμη να φτάσω στο τέλος του.

Αθήνα, Νοέμβριος 2025

Ελένη Τσέρτου

Table of Contents

List of Figures

Chapter 1

Εισαγωγή

1.1 Κίνητρο

Η ραγδαία αύξηση του όγκου δεδομένων τα τελευταία χρόνια έχει προκαλέσει μια σημαντική αλλαγή στον τρόπο με τον οποίο οι οργανισμοί αποθηκεύουν, διαχειρίζονται και αναλύουν πληροφορίες. Τα παραδοσιακά συστήματα (data warehouses) αποθήκευσης και επεξεργασίας δεδομένων, παρότι αποτελεσματικά για δομημένα δεδομένα, δυσκολεύονται να ανταποκριθούν στις απαιτήσεις των σύγχρονων συνόλων δεδομένων που χαρακτηρίζονται από μεγάλο όγκο, ποικιλία και ταχύτητα.

Για την αντιμετώπιση αυτών των προκλήσεων, αναπτύχθηκαν οι λεγόμενες λίμνες δεδομένων (data lakes)— κεντρικά αποθετήρια που αποθηκεύουν δεδομένα σε ακατέργαστη μορφή και σε μεγάλη κλίμακα. Αρχικά, βασίζονταν σε απλά συστήματα αντικειμενοστραφούς αποθήκευσης, τα οποία επέτρεπαν την εισαγωγή και διατήρηση δεδομένων ανεξάρτητα από τους υπολογιστικούς πόρους. Ωστόσο, με την αύξηση των απαιτήσεων για διακυβέρνηση δεδομένων, απόδοση αναλύσεων και πολυτροπική πρόσβαση, οι λίμνες δεδομένων παρουσίασαν σοβαρές αδυναμίες — όπως χαμηλή απόδοση ερωτημάτων, έλλειψη εγγυήσεων συναλλαγών και απουσία επιβολής σχημάτων.

Για την επίλυση αυτών των προβλημάτων, εμφανίστηκε μια νέα γενιά ανοιχτών μορφών πίνακα (Log Structure Table Formats), που προσθέτουν λειτουργίες τύπου βάσης δεδομένων στις λίμνες δεδομένων. Οι τεχνολογίες αυτές — όπως τα Apache Iceberg, Apache Hudi και Delta Lake — προσφέρουν δυνατότητες όπως μεταβολή σχημάτων, υποστήριξη ACID συναλλαγών, ερωτήματα "time travel" και αποδοτική διαχείριση αρχείων πάνω σε αντικειμενοστραφή αποθήκευση.

Σκοπός της παρούσας διπλωματικής εργασίας είναι η συγκριτική αξιολόγηση της απόδοσης και της κλιμάκωσης των τεχνολογιών Apache Iceberg, Apache Hudi και Delta Lake σε διαφορετικούς τύπους φορτίου εργασίας (workload), μέσω πειραμάτων benchmarking.

1.2 Σχετικές εργασίες και πλαίσιο αναφοράς

Μία σημαντική συμβολή στη μελέτη των σύγχρονων αρχιτεκτονικών data lake προέρχεται από την πρόσφατη εργασία της Microsoft με τίτλο "LST-Bench: Benchmarking Log-Structured Tables in the Cloud". Η μελέτη αυτή παρουσιάζει το LST-Bench, ένα εξειδικευμένο πλαίσιο αξιολόγησης (benchmarking framework) που έχει σχεδιαστεί για να εξετάζει την απόδοση και

λειτουργικότητα των ανοιχτών μορφών πίνακα (open table formats), όπως τα Apache Iceberg, Apache Hudi και Delta Lake, σε περιβάλλοντα cloud-based data lakes.

Το LST-Bench βασίζεται στο καθιερωμένο πρότυπο TPC-DS benchmark, το οποίο επεκτείνει με δυνατότητες προσαρμοσμένες στις ιδιαιτερότητες των log-structured table formats. Επιπλέον, συλλέγει τηλεμετρία τόσο από τη μηχανή υπολογισμού όσο και από τις υπηρεσίες αποθήκευσης στο cloud, προσφέροντας μια ολιστική εικόνα της συμπεριφοράς του συστήματος υπό ρεαλιστικά φορτία εργασίας.

Η παρούσα διπλωματική εργασία υιοθετεί τη μεθοδολογία του LST-Bench ως βάση για τη συγκριτική αξιολόγηση των Apache Iceberg, Apache Hudi και Delta Lake. Παρότι αξιοποιεί το σχεδιασμό και τις μετρικές απόδοσης της Microsoft, η μελέτη διεξάγεται σε διαφορετικό περιβάλλον εκτέλεσης — συγκεκριμένα, σε τοπική αποθήκευση HDFS και με χρήση του Apache Spark ως μοναδικής μηχανής επεξεργασίας. Αυτό αν αντανακλά μια διαφορετική αρχιτεκτονική προσέγγιση, η κοινή εστίαση στην αξιολόγηση των open table formats αναδεικνύει τη γενικότερη σημασία της μελέτης αυτών των τεχνολογιών σε ποικίλα μοντέλα ανάπτυξης.

1.3 Δομή Εργασίας

Στις επόμενες ενότητες, η διπλωματική εργασία διερευνά τα τεχνικά θεμέλια, την υλοποίηση και την αξιολόγηση των μορφών πινάκων με δομή καταγραφής (log-structured table formats) σε μια αρχιτεκτονική data lakehouse.

- Η ενότητα 1.4 παρέχει τις απαραίτητες βασικές πληροφορίες σχετικά με τα αποθετήρια δεδομένων, τις λίμνες δεδομένων και την εξέλιξη προς τα συστήματα lakehouse.
- Η ενότητα 1.5 παρουσιάζει τις λεπτομέρειες υλοποίησης του lakehouse stack, εστιάζοντας στα επίπεδα αποθήκευσης, μεταδεδομένων και επεξεργασίας, με έμφαση στα Delta Lake, Apache Iceberg και Apache Hudi.
- Η ενότητα 1.6 εισάγει το πλαίσιο αξιολόγησης LST-Bench, περιγράφοντας τον σχεδιασμό, τα πρότυπα φόρτου εργασίας και τις βελτιώσεις που έχουν προσαρμοστεί για την αξιολόγηση των log-structured formats.
- Η ενότητα 1.7 περιγράφει τη πειραματική διάταξη, συμπεριλαμβανομένων των προδιαγραφών υλικού, της δημιουργίας dataset, της αρχικοποίησης του συστήματος και της ενσωμάτωσης του benchmark.
- Τέλος, η ενότητα 1.8 παρουσιάζει τα αποτελέσματα της αξιολόγησης, συγκρίνοντας την απόδοση μεταξύ των διαφορετικών μορφών και φορτίων εργασίας, και συζητά τις επιπτώσεις των ευρημάτων.

1.4 Υπόβαθρο

Στην εποχή της λήψης αποφάσεων βάσει δεδομένων, οι οργανισμοί καλούνται να επεξεργάζονται τεράστιους όγκους δομημένων, ημιδομημένων και αδόμητων δεδομένων—συχνά σε πραγματικό χρόνο. Τα παραδοσιακά συστήματα, όπως οι σχεσιακές βάσεις δεδομένων και οι

κλασικές αποθήκες δεδομένων, δυσκολεύονται να ανταποκριθούν στις απαιτήσεις που θέτει η κλίμακα, η ποικιλομορφία και η ταχύτητα των big data. Αυτοί οι περιορισμοί έχουν οδηγήσει στην υιοθέτηση πιο επεκτάσιμων και ευέλικτων αρχιτεκτονικών: αρχικά των λιμνών δεδομένων (data lakes) και πιο πρόσφατα του μοντέλου lakehouse.

1.4.1 Αποθήκες Δεδομένων

Σύμφωνα με τον ορισμό του Bill Inmon, μια αποθήκη δεδομένων είναι μια συλλογή δεδομένων που είναι προσανατολισμένη σε θέματα, μη πτητική, ενοποιημένη και μεταβαλλόμενη στον χρόνο, σχεδιασμένη για να υποστηρίζει διαδικασίες λήψης αποφάσεων. Οι αποθήκες δεδομένων συγκεντρώνουν δεδομένα από πολλαπλές πηγές μέσω της διαδικασίας ETL (Extract, Transform, Load), μετατρέποντάς τα σε συνεπή μορφή για ανάλυση. Μία απλή αναπαράσταση τυπικής αποθήκης δεδομένων παρουσιάζεται στο Figure 1.1.

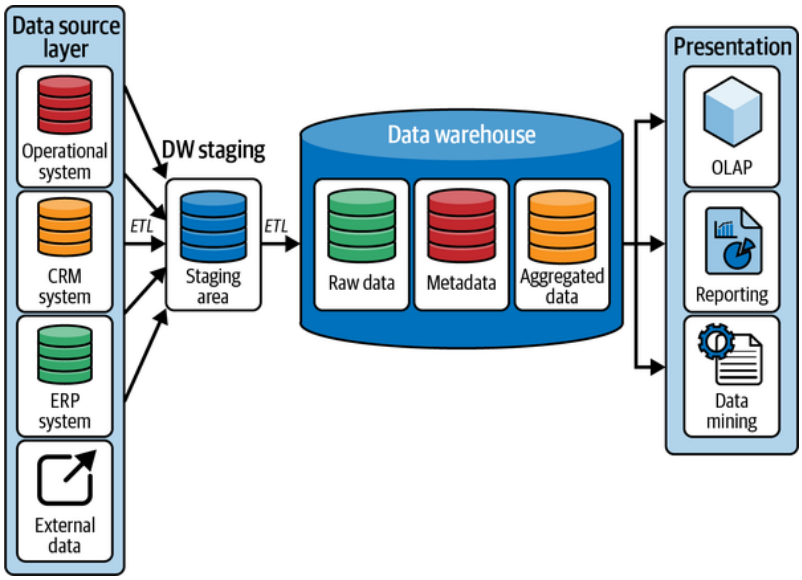

Figure 1.1. *Αποθήκη Δεδομένων, Πηγή Εικόνας: https://books.google.gr/books?id=CSfdEAAAQBAJ*

Οι αποθήκες δεδομένων έχουν προσφέρει αξιόπιστα και τυποποιημένα δεδομένα για επιχειρησιακή ανάλυση και ιστορική επισκόπηση. Ωστόσο, η ραγδαία εξάπλωση του διαδικτύου, των κοινωνικών μέσων και των πολυμέσων οδήγησε στην εμφάνιση των Big Data, που χαρακτηρίζονται από τέσσερις βασικές διαστάσεις: όγκο, ταχύτητα, ποικιλία και αξιοπιστία (Volume, Velocity, Variety, Veracity).

Οι παραδοσιακές αποθήκες δεδομένων δυσκολεύονται να ανταποκριθούν στις απαιτήσεις των Big Data. Αντιμετωπίζουν περιορισμούς στην κλιμάκωση, δεν υποστηρίζουν επεξεργασία σε πραγματικό χρόνο, και είναι ακατάλληλες για ημιδομημένα ή αδόμητα δεδομένα. Επιπλέον, παρέχουν περιορισμένα μεταδεδομένα για την ποιότητα και την προέλευση των δεδομένων, ενώ η εξάρτησή τους από ιδιόμορφες μορφές και SQL εργαλεία τις καθιστά ασύμβατες με σύγχρονα οικοσυστήματα μηχανικής μάθησης και επιστήμης δεδομένων. Αυτοί οι περιορισμοί αυξάνουν το κόστος και τον κίνδυνο αποτυχίας έργων, δυσχεραίνοντας την προσαρμογή των επιχειρήσεων στις σύγχρονες ανάγκες.

1.4.2 Λίμνες Δεδομένων

Η έννοια της λίμνης δεδομένων προτάθηκε το 2010 από τον James Dixon, ως λύση που επιτρέπει την αποθήκευση ακατέργαστων δεδομένων σε φυσική μορφή, χωρίς την ανάγκη άμεσης επεξεργασίας. Οι λίμνες δεδομένων αποτελούν οικονομικά αποθετήρια για δεδομένα κάθε μορφής και κλίμακας, βασισμένα στην αναλογία με λίμνες που δέχονται ροές από πολλαπλές πηγές σε πραγματικό χρόνο. Μία τυπική λίμνη δεδομένων παρουσιάζεται στο Figure 1.2.

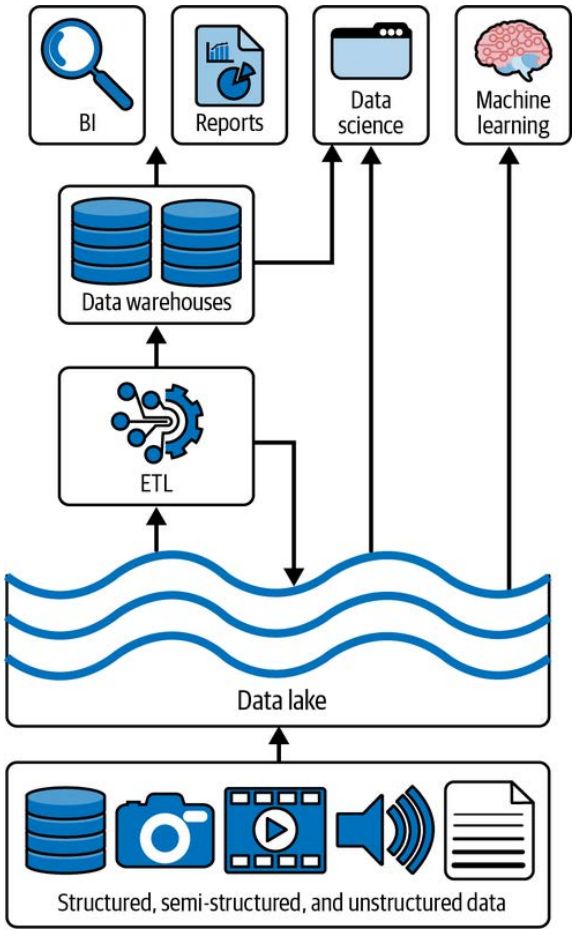

Figure 1.2. *Κανονικοποιημένη Λίμνη Δεδομένων, Πηγή Εικόνας: https://books.google.gr/books?id=CSfdEAAAQBAJ*

Αρχικά, οι λίμνες δεδομένων υλοποιούνταν σε τοπικά clusters με χρήση του Apache Hadoop, το οποίο επέτρεπε την κατανεμημένη επεξεργασία μεγάλων datasets μέσω του MapReduce. Το HDFS αποτέλεσε βασικό στοιχείο, προσφέροντας ανθεκτικότητα και χαμηλό κόστος. Από το 2015 και μετά, παρατηρείται στροφή προς cloud-based data lakes όπως τα Amazon S3, Azure ADLS και Google Cloud Storage, που προσφέρουν υψηλή αξιοπιστία, γεωγραφική αναπαραγωγή και χαμηλό κόστος αποθήκευσης.

Οι σύγχρονες ροές επεξεργασίας δεδομένων βασίζονται σε compute engines για μετασχηματισμό μεγάλων όγκων πληροφορίας, εξυπηρετώντας downstream συστήματα όπως αποθήκες δεδομένων, εργαλεία τεχνητής νοημοσύνης και μηχανικής μάθησης. Η συνεχής εισαγωγή δεδομένων και η παραγωγή αναλυτικών αποτελεσμάτων υποστηρίζονται από εφαρμογές BI και συστήματα αναφορών.

1.4.2.1 Βασικά στοιχεία

Η λειτουργία των data lakes βασίζεται σε έναν συνδυασμό τεχνολογικών συνιστωσών.

- **Υποδομή Αποθήκευσης**: η υποδομή αποθήκευσης πρέπει να είναι επεκτάσιμη και ανθεκτική, και συχνά υλοποιείται σε περιβάλλοντα cloud. Ένα σημαντικό πλεονέκτημα των λιμνών δεδομένων είναι η αρχιτεκτονική διάκριση μεταξύ αποθήκευσης και υπολογιστικής ισχύος, γεγονός που επιτρέπει την ανεξάρτητη και ελαστική κλιμάκωση των πόρων. Επιπλέον, απαιτούνται υψηλές ταχύτητες μεταφοράς δεδομένων για την υποστήριξη μαζικών φορτίων ή συνεχών ροών, όπως τηλεμετρία από IoT συσκευές και πολυμεσικό περιεχόμενο.

- **Υπολογιστική Ισχύς**: η υπολογιστική ισχύς αποτελεί επίσης κρίσιμο παράγοντα, καθώς οι λίμνες δεδομένων φιλοξενούν τεράστιους όγκους δεδομένων που πρέπει να επεξεργαστούν αποτελεσματικά. Οι σύγχρονες cloud πλατφόρμες προσφέρουν μηχανές επεξεργασίας όπως το Apache Spark, το οποίο έχει καθιερωθεί ως το πρότυπο για αναλύσεις μεγάλης κλίμακας. Το Spark μπορεί να υλοποιηθεί μέσω υπηρεσιών όπως το Databricks ή αντίστοιχες ιδιόκτητες λύσεις, αξιοποιώντας υπολογιστικά clusters που εκτελούν συνεργατικά σύνθετες εργασίες επεξεργασίας.

- **Μορφή των δεδομένων**: η μορφή των δεδομένων στο δίσκο καθορίζει το format τους. Οι λίμνες δεδομένων χρησιμοποιούν ευρέως αποδεκτά, ανοιχτά πρότυπα όπως Parquet, Avro, JSON και CSV, τα οποία εξασφαλίζουν συμβατότητα με εργαλεία ανάλυσης και αποδοτική αποθήκευση.

- **Μεταδεδομένα**: τα συστήματα αποθήκευσης στο cloud ενσωματώνουν μεταδεδομένα που προσφέρουν κρίσιμες πληροφορίες για το περιεχόμενο των δεδομένων, όπως χρονικές σημάνσεις, σχήματα δεδομένων και περιγραφικές ετικέτες που δηλώνουν ιδιοκτησία και χαρακτηριστικά χρήσης.

1.4.2.2 Πλεονεκτήματα των αρχιτεκτονικών Λιμνών Δεδομένων

Οι λίμνες δεδομένων προσφέρουν σημαντικά πλεονεκτήματα. Αποτελούν ένα ενοποιημένο αποθετήριο για όλα τα δεδομένα ενός οργανισμού, ανεξαρτήτως δομής ή προέλευσης. Η χρήση ανοιχτών μορφών δεδομένων όπως Parquet και Avro διευκολύνει την ενσωμάτωση με διάφορες πλατφόρμες και εργαλεία. Χάρη στις ώριμες αρχιτεκτονικές αποθήκευσης στο cloud, οι λίμνες δεδομένων επωφελούνται από υψηλή επεκτασιμότητα, δυνατότητες παρακολούθησης, ευκολία στην ανάπτυξη και οικονομική αποθήκευση. Επιπλέον, η υλοποίηση και η συντήρησή τους μπορούν να αυτοματοποιηθούν μέσω εργαλείων Infrastructure-as-Code, όπως το Terraform.

Σε αντίθεση με τις παραδοσιακές αποθήκες δεδομένων, οι λίμνες δεδομένων υποστηρίζουν όλους τους τύπους δεδομένων—δομημένα, ημιδομημένα και αδόμητα—καθιστώντας τα κατάλληλα για σύνθετα φορτία εργασίας όπως η επεξεργασία πολυμέσων. Οι υψηλής απόδοσης ροές δεδομένων τους υποστηρίζουν εφαρμογές σε πραγματικό χρόνο, όπως εισαγωγή δεδομένων από αισθητήρες IoT, streaming πολυμέσων και ανάλυση συμπεριφοράς από web clickstreams.

1.4.2.3 Περιορισμοί

Παρά την ευρεία υιοθέτηση και την αρχιτεκτονική ευελιξία τους, τα παραδοσιακά data lakes παρουσιάζουν αρκετές λειτουργικές και τεχνικές προκλήσεις που μπορούν να περιορίσουν την αποτελεσματικότητά τους. Αν και οι υποκείμενες λύσεις αποθήκευσης στο cloud παραμένουν οικονομικές, η δημιουργία και διαχείριση μιας πλήρους αρχιτεκτονικής data lake απαιτεί εξειδικευμένη τεχνογνωσία, γεγονός που συχνά οδηγεί σε αυξημένο λειτουργικό κόστος—είτε μέσω εξειδικευμένου προσωπικού είτε μέσω εξωτερικών συμβούλων.

Η εισαγωγή ακατέργαστων δεδομένων στο data lake είναι σχετικά απλή, όμως η μετατροπή τους σε αναλυτικά χρήσιμες μορφές απαιτεί σύνθετη επεξεργασία και αυξημένο οικονομικό κόστος. Επιπλέον, οι παραδοσιακές λίμνες δεδομένων παρουσιάζουν υψηλή καθυστέρηση στα ερωτήματα, γεγονός που τα καθιστά ακατάλληλα για διαδραστική ή σε πραγματικό χρόνο εξερεύνηση δεδομένων.

Ένα ακόμη βασικό χαρακτηριστικό των παραδοσιακών λιμνών δεδομένων είναι η αρχή schema-on-read, όπου τα δεδομένα εισάγονται χωρίς αυστηρή επιβολή σχήματος και ερμηνεύονται μόνο κατά την ανάγνωση. Αν και αυτό προσφέρει ευελιξία, ενέχει τον κίνδυνο χαμηλής ποιότητας και ασυνέπειας των δεδομένων, οδηγώντας συχνά στη μετατροπή της λίμνης δεδομένων σε βάλτο δεδομένων, δηλαδή σε ένα αποθετήριο μη διαχειρίσιμων και αναξιόπιστων δεδομένων.

Επιπλέον, οι λίμνες δεδομένων δεν παρέχουν συναλλακτικές εγγυήσεις (ACID). Τα δεδομένα μπορούν μόνο να προστεθούν και όχι να τροποποιηθούν επιτόπου, με αποτέλεσμα οι απλές ενημερώσεις να απαιτούν επανεγγραφή ολόκληρων αρχείων, κάτι που είναι υπολογιστικά δαπανηρό. Αυτό οδηγεί στο λεγόμενο πρόβλημα των μικρών αρχείων, όπου δημιουργούνται πολυάριθμα μικρά αρχεία για κάθε οντότητα δεδομένων. Αν δεν υπάρξει σωστή διαχείριση, η συσσώρευση αυτών των αρχείων υποβαθμίζει την απόδοση ανάγνωσης και αυξάνει το κόστος αποθήκευσης λόγω πλεονασμού και κατακερματισμού.

1.4.3 Εξέλιξη στα Data Lakehouses

Το lakehouse είναι ένα σύστημα διαχείρισης δεδομένων που βασίζεται σε οικονομική και άμεσα προσβάσιμη αποθήκευση, ενώ ενσωματώνει χαρακτηριστικά απόδοσης και διαχείρισης βάσεων δεδομένων όπως ACID συναλλαγές, έκδοση δεδομένων, audit logs, ευρετηρίαση, caching και βελτιστοποίηση ερωτημάτων.

Η αρχιτεκτονική lakehouse προσφέρει την ευελιξία, επεκτασιμότητα και οικονομία των λιμνών δεδομένων, ενώ ταυτόχρονα ενσωματώνει τις δυνατότητες διαχείρισης και αξιοπιστίας των data warehouses, αντιμετωπίζοντας τους περιορισμούς και των δύο μοντέλων. Είναι ιδιαίτερα κατάλληλη για περιβάλλοντα cloud όπου η αποθήκευση και η υπολογιστική ισχύς είναι διαχωρισμένες, επιτρέποντας σε διαφορετικές εφαρμογές να εκτελούνται κατά απαίτηση σε ανεξάρτητους υπολογιστικούς κόμβους (π.χ. Spark clusters), με άμεση πρόσβαση στα ίδια δεδομένα αποθήκευσης.

Η μετάβαση από τις παραδοσιακές αποθήκες δεδομένων (data warehouses), στα data lakes και τελικά στην αρχιτεκτονική lakehouse, αντιπροσωπεύει μια φυσική εξέλιξη στην προσπάθεια για πιο ενοποιημένη, αποδοτική και ευέλικτη διαχείριση δεδομένων σε σύγχρονα επιχειρησιακά περιβάλλοντα.

1.5 Υλοποιώντας ένα Data Lakehouse

Η αρχιτεκτονική του lakehouse αποτελείται από τρία επίπεδα. Το κατώτερο επίπεδο είναι το επίπεδο αντικειμενοστραφούς αποθήκευσης (object storage). Πάνω από το επίπεδο αποθήκευσης βρίσκεται το επίπεδο μεταδεδομένων (metadata layer), και τέλος το ανώτερο επίπεδο της αρχιτεκτονικής lakehouse αποτελείται από μηχανές επεξεργασίας και ερωτημάτων υψηλής απόδοσης, όπως το Apache Spark ή το Trino. Στη συνέχεια, θα εξεταστούν αναλυτικά τα επιμέρους επίπεδα της αρχιτεκτονικής lakehouse.

1.5.1 Επίπεδο Αποθήκευσης

Στη βάση της αρχιτεκτονικής lakehouse βρίσκεται το επίπεδο αποθήκευσης, το οποίο είναι υπεύθυνο για τη διατήρηση μεγάλων όγκων δεδομένων με οικονομικό και επεκτάσιμο τρόπο. Αυτό το επίπεδο μπορεί να υλοποιηθεί είτε μέσω λύσεων αντικειμενοστραφούς αποθήκευσης στο cloud, όπως το Amazon S3 και το Azure Data Lake Storage, που προσφέρουν ελαστικότητα και διαχειριζόμενη υποδομή, είτε μέσω τοπικών (on-premises) συστημάτων.

Ένα από τα πιο ώριμα και ευρέως υιοθετημένα συστήματα αποθήκευσης σε τοπικά περιβάλλοντα είναι το Hadoop Distributed File System (HDFS). Το HDFS λειτουργεί με αρχιτεκτονική master-slave, αποτελούμενη από έναν NameNode, ο οποίος διαχειρίζεται το namespace του συστήματος αρχείων και τα μεταδεδομένα, και από DataNodes, που είναι υπεύθυνα για την αποθήκευση των πραγματικών μπλοκ δεδομένων.

Ορισμένα βασικά πλεονεκτήματα του HDFS στο πλαίσιο ενός lakehouse περιλαμβάνουν:

- **Υψηλή Απόδοση και Επεκτασιμότητα**: Βελτιστοποιημένο για επεξεργασία παρτίδων με υψηλό ρυθμό μεταφοράς δεδομένων, επιτρέποντας οριζόντια κλιμάκωση καθώς αυξάνεται ο όγκος των δεδομένων.

- **Η αρχή "Γράψε Μία Φορά, Διάβασε Πολλές"**: Ευθυγραμμίζεται με αναλυτικά φορτία εργασίας όπου τα δεδομένα εισάγονται μαζικά και προσπελάζονται συχνά για ερωτήματα και αναφορές.

- **Συμβατότητα με Ανοιχτά Formats**: Υποστηρίζει μορφές όπως Apache Parquet και ORC, απαραίτητες για τη διαλειτουργικότητα με table formats όπως Delta Lake, Apache Iceberg και Apache Hudi.

- **Ενσωμάτωση με Οικοσυστήματα Big Data**: Ως βασικό στοιχείο του οικοσυστήματος Hadoop, είναι συμβατό με πλαίσια επεξεργασίας όπως Apache Spark, Hive και MapReduce.

Παρότι η αντικειμενοστραφής αποθήκευση στο cloud έχει αποκτήσει δυναμική λόγω της ελαστικότητας και των διαχειριζόμενων υπηρεσιών της, το HDFS παραμένει μια αξιόπιστη και οικονομική λύση για οργανισμούς που λειτουργούν σε αυτοδιαχειριζόμενα ή υβριδικά περιβάλλοντα, ιδιαίτερα όταν υπάρχουν απαιτήσεις για κυριαρχία στα δεδομένα, χαμηλή καθυστέρηση ή έλεγχο της υποδομής.

1.5.2 Επίπεδο Μεταδεδομένων

Έχουν εμφανιστεί αρκετά έργα ανοικτού κώδικα με σκοπό να καλύψουν το επίπεδο μεταδεδομένων και τις προηγμένες δυνατότητες διαχείρισης δεδομένων που απαιτούνται για τη λειτουργία ενός lakehouse. Μεταξύ των πιο σημαντικών είναι τα Delta Lake, Apache Iceberg και Apache Hudi. Οι τεχνολογίες αυτές επεκτείνουν τις βασικές δυνατότητες των data lakes, εισάγοντας χαρακτηριστικά όπως ACID συναλλαγές, αναδρομή στον χρόνο (time travel) και μεταβολή σχημάτων (schema evolution).

Delta Lake Το Delta Lake είναι μια πλατφόρμα ανοικτού κώδικα για τη διαχείριση αποθήκευσης δεδομένων, η οποία υποστηρίζει ACID συναλλαγές, επεκτάσιμη διαχείριση μεταδεδομένων και ενοποίηση της επεξεργασίας δεδομένων σε ροή (streaming) και παρτίδες (batch). Αρχικά σχεδιάστηκε για να λειτουργεί με το Apache Spark και φορτία εργασίας μεγάλης κλίμακας σε data lakes. Ωστόσο, με την εξέλιξή του, το Delta Lake έχει βελτιστοποιηθεί ώστε να υποστηρίζει ποικίλα φορτία (μικρά, μεσαία και μεγάλα δεδομένα) και να συνεργάζεται με πολλαπλά frameworks (όπως Apache Spark, Apache Flink, Trino, Presto, Apache Hive και Apache Druid), υπηρεσίες (όπως Athena, BigQuery, Databricks, EMR, Fabric, Glue, Starburst και Snowflake) και γλώσσες προγραμματισμού (.NET, Java, Python, Rust, Scala, SQL κ.ά.).

Ένας πίνακας Delta Lake αποτελείται από βασικά συστατικά που μαζί διαμορφώνουν ένα ισχυρό, επεκτάσιμο και υψηλής απόδοσης σύστημα διαχείρισης δεδομένων:

- **Αρχεία δεδομένων** αποθηκεύονται σε μορφή Apache Parquet, η οποία είναι ιδιαίτερα αποδοτική για μεγάλες συλλογές δεδομένων, και βρίσκονται σε κατανεμημένα συστήματα αρχείων όπως HDFS, Amazon S3, Azure Blob Storage, Google Cloud Storage ή MinIO.

- Το **αρχείο καταγραφής συναλλαγών** (transaction log), γνωστό ως Delta log, είναι θεμελιώδες για τη συμμόρφωση με τις ACID αρχές. Καταγράφει κάθε αλλαγή στον πίνακα σε μορφή JSON, περιλαμβάνοντας μεταδεδομένα όπως τον τύπο της ενέργειας, τα επηρεαζόμενα αρχεία και το σχήμα του πίνακα τη στιγμή της αλλαγής.

- Τα **μεταδεδομένα** περιλαμβάνουν ορισμούς σχημάτων, λεπτομέρειες κατατμήσεων και ρυθμίσεις παραμέτρων. Είναι ενσωματωμένα στο αρχείο καταγραφής και προσβάσιμα μέσω διεπαφών όπως SQL, Apache Spark, Rust και Python, επιτρέποντας την επιβολή σχημάτων και στρατηγικές βελτιστοποίησης.

- Το **σχήμα** καθορίζει τη δομή του πίνακα—συμπεριλαμβανομένων των στηλών και των τύπων δεδομένων—και επιβάλλεται κατά την εγγραφή. Το Delta Lake υποστηρίζει μεταβολή σχημάτων (schema evolution), επιτρέποντας αλλαγές στη δομή καθώς μεταβάλλονται οι απαιτήσεις των δεδομένων.

- Τα **checkpoints** είναι περιοδικά στιγμιότυπα του αρχείου καταγραφής, αποθηκευμένα σε μορφή Parquet, που επιταχύνουν την απόδοση των ερωτημάτων και την ανάκτηση, επιτρέποντας στα συστήματα να παρακάμπτουν την επανάληψη ολόκληρου του ιστορικού του log. Από προεπιλογή, το Delta Lake δημιουργεί checkpoints κάθε δέκα συναλλαγές.

Όσον αφορά την αρχιτεκτονική του Delta Lake, αξίζει να σημειωθεί πώς διαχειρίζεται τροποποιήσεις όπως διαγραφές. Σε συστήματα αντικειμενοστραφούς αποθήκευσης, αντί να τροποποιεί υπάρχοντα αρχεία Parquet, είναι πιο αποδοτικό να δημιουργεί νέα αρχεία που περιέχουν μόνο τις μη επηρεασμένες γραμμές. Αυτή η προσέγγιση ενισχύει την απόδοση και ευθυγραμμίζεται με την αρχή του Multiversion Concurrency Control (MVCC)—μια τεχνική που διατηρεί πολλαπλές εκδόσεις των δεδομένων, επιτρέποντας ασφαλείς και ταυτόχρονες λειτουργίες ανάγνωσης και εγγραφής. Η τεχνική αυτή επιτρέπει επίσης στο Delta Lake να προσφέρει time travel, δηλαδή τη δυνατότητα ερωτημάτων σε ιστορικές εκδόσεις των δεδομένων.

Apache Iceberg Το Apache Iceberg είναι ένα table format ανοικτού κώδικα, σχεδιασμένο για την αποδοτική διαχείριση αναλυτικών συνόλων δεδομένων σε κλίμακα petabyte, σε κατανεμημένα συστήματα και αντικειμενοστραφείς αποθηκεύσεις στο cloud. Προσφέρει προηγμένες δυνατότητες όπως μεταβολή σχημάτων (schema evolution), υποστήριξη ACID συναλλαγών, κρυφή κατατμηση (hidden partitioning) και βελτιστοποιημένη απόδοση ερωτημάτων. Το Iceberg είναι συμβατό με πληθώρα μηχανών επεξεργασίας, όπως Apache Spark, Trino, Flink και Presto, και υποστηρίζει συνεπή και ταυτόχρονη ανάγνωση και εγγραφή, καθιστώντας το ιδανικό για σύγχρονες αρχιτεκτονικές μεγάλης κλίμακας.

Το Apache Iceberg εισάγει βασικές έννοιες που υποστηρίζουν επεκτάσιμη και αποδοτική επεξεργασία δεδομένων:

- **Διαχείριση Μεταδεδομένων**: Χρησιμοποιεί πολυεπίπεδο σύστημα μεταδεδομένων—συμπεριλαμβανομένων αρχείων metadata, manifest και manifest lists (σε μορφή JSON)—για την παρακολούθηση σχημάτων, κατατμήσεων, αρχείων δεδομένων και στατιστικών. Υποστηρίζει έκδοση βάσει snapshots, επιτρέποντας συνεπή κατάσταση πίνακα στον χρόνο.

- **Στρώμα Καταλόγου (Catalog Layer)**: Ο κατάλογος διατηρεί αναφορά στο πιο πρόσφατο αρχείο μεταδεδομένων για κάθε πίνακα. Όταν γίνεται αλλαγή, δημιουργείται νέο αρχείο metadata και ο κατάλογος ενημερώνει τον δείκτη ώστε να αντικατοπτρίζει την τρέχουσα κατάσταση του πίνακα. Αυτός ο μηχανισμός επιτρέπει ACID συναλλαγές, διασφαλίζοντας ότι οι τροποποιήσεις παραμένουν απομονωμένες και γίνονται ορατές μόνο όταν ολοκληρωθούν, διευκολύνοντας συνεπή ανάγνωση και απλοποιώντας τον έλεγχο ταυτόχρονης πρόσβασης.

- **Μεταβολή Σχήματος (Schema Evolution)**: Επιτρέπει την ομαλή αλλαγή του σχήματος (π.χ. προσθήκη στηλών) χωρίς να απαιτείται δαπανηρή επανεγγραφή των δεδομένων. Τα μεταδεδομένα ενημερώνονται ώστε να αντικατοπτρίζουν τις αλλαγές, ενώ τα υπάρχοντα δεδομένα παραμένουν ανέπαφα

- **Κατάτμηση (Partitioning)**: Υποστηρίζει προηγμένες στρατηγικές κατάτμησης όπως range, hash, truncate και list partitioning. Αυτές οι μέθοδοι βελτιώνουν την απόδοση των ερωτημάτων μειώνοντας τον όγκο των δεδομένων που πρέπει να σαρωθούν.

- **Snapshots**: Κάθε αλλαγή δημιουργεί ένα νέο snapshot—μια αμετάβλητη προβολή του πίνακα σε συγκεκριμένη χρονική στιγμή. Αυτό επιτρέπει time travel και δυνατότητες επαναφοράς για πρόσβαση σε ιστορικά δεδομένα και ανάκτηση.

Apache Hudi Το Apache Hudi (Hadoop Upserts Deletes and Incrementals) είναι ένα πλαίσιο lakehouse ανοικτού κώδικα, σχεδιασμένο για να προσφέρει συναλλακτικές δυνατότητες και αποδοτική διαχείριση δεδομένων σε cloud-native data lakes. Αναπτύχθηκε αρχικά από την Uber και υποστηρίζει λειτουργίες όπως εισαγωγές, ενημερώσεις και διαγραφές σε επίπεδο εγγραφής, καθιστώντας το ιδιαίτερα κατάλληλο για επεξεργασία δεδομένων με σταδιακή ενημέρωση (incremental processing) και σενάρια καταγραφής αλλαγών (CDC – Change Data Capture).

Για να υποστηρίξει αυτές τις δυνατότητες, το Apache Hudi εισάγει βασικές έννοιες:

- **Διαχείριση Χρονολογίου και Μεταδεδομένων**: Το Hudi διατηρεί ένα χρονολόγιο όλων των ενεργειών (commits, cleanups, compactions, rollbacks) που εκτελούνται σε έναν πίνακα. Κάθε ενέργεια συνδέεται με μια χρονική στιγμή (instant time), εξασφαλίζοντας ατομικότητα και συνέπεια. Τα μεταδεδομένα αποθηκεύονται μαζί με τα αρχεία δεδομένων και παρακολουθούν το σχήμα, τις εκδόσεις αρχείων και την κατατμηση.

- **Τύποι Πινάκων**:

  Copy-on-Write (CoW): Οι ενημερώσεις δημιουργούν νέες εκδόσεις αρχείων, βελτιστοποιημένες για φορτία με έντονη ανάγνωση.

  Merge-on-Read (MoR): Οι ενημερώσεις γράφονται σε αρχεία καταγραφής διαφορών (delta logs) και συγχωνεύονται κατά την ανάγνωση, ιδανικές για φορτία με έντονη εγγραφή ή streaming.

- **Τύποι Ερωτημάτων**:

  Snapshot Queries: Παρέχουν την πιο πρόσφατη προβολή των δεδομένων.

  Incremental Queries: Ανακτούν μόνο τα δεδομένα που έχουν αλλάξει από μια συγκεκριμένη χρονική στιγμή.

  Read-Optimized Queries: Προσπελάζουν τα βασικά αρχεία χωρίς συγχώνευση των logs, προσφέροντας ταχύτερη ανάγνωση.

- **Μεταβολή Σχήματος (Schema Evolution)**: Υποστηρίζει αλλαγές στο σχήμα με την πάροδο του χρόνου, επιτρέποντας την προσθήκη νέων πεδίων χωρίς επανεγγραφή των υπαρχόντων δεδομένων.

- **Συμπύκνωση και Ομαδοποίηση (Compaction & Clustering)**: Υπηρεσίες παρασκηνίου όπως η συμπύκνωση (για MoR πίνακες) και η ομαδοποίηση (για βελτιστοποίηση της διάταξης αρχείων) βελτιώνουν την απόδοση και την αποδοτικότητα αποθήκευσης.

- **Time Travel και Rollbacks**: Μέσω του χρονολογίου, οι χρήστες μπορούν να προσπελάσουν ιστορικές εκδόσεις των δεδομένων ή να επαναφέρουν τον πίνακα σε προηγούμενη κατάσταση σε περίπτωση σφάλματος ή αλλοίωσης.

Το Hudi ενσωματώνεται με δημοφιλείς μηχανές επεξεργασίας όπως Apache Spark, Flink, Presto, Trino και Hive, και είναι συμβατό με πλατφόρμες αποθήκευσης στο cloud όπως Amazon S3, Azure Data Lake Storage και Google Cloud Storage. Οι προηγμένες δυνατότητες ευρετηρίασης, ομαδοποίησης και συμπύκνωσης ενισχύουν περαιτέρω την απόδοση τόσο για παρτίδες όσο και για ροές δεδομένων (batch & streaming workloads).

1.5.3 Επίπεδο Επεξεργασίας

Το επίπεδο επεξεργασίας στην αρχιτεκτονική lakehouse είναι υπεύθυνο για την εκτέλεση αναλύσεων, μετασχηματισμών και επεξεργασίας σε πραγματικό χρόνο. Το Apache Spark αποτελεί την κυρίαρχη μηχανή επεξεργασίας, προσφέροντας υψηλή απόδοση μέσω επεξεργασίας στη μνήμη και κατανεμημένης εκτέλεσης. Συνδέεται απευθείας με τα επίπεδα αποθήκευσης και μεταδεδομένων και συνεργάζεται άψογα με table formats όπως Delta Lake, Iceberg και Hudi. Υποστηρίζει ACID συναλλαγές, SQL ερωτήματα, schema evolution, time travel, batch και streaming επεξεργασία, καθώς και ενσωμάτωση με εργαλεία μηχανικής μάθησης. Η ευελιξία και η επεκτασιμότητά του το καθιστούν βασικό στοιχείο σε κάθε σύγχρονη lakehouse υποδομή.

1.6 Αξιολόγηση Μορφών Πινάκων με Λογική Καταγραφής: Το Πλαίσιο LST-Bench

Καθώς οι αρχιτεκτονικές των data lakes εξελίσσονται για να υποστηρίξουν πιο σύνθετα και μεταβαλλόμενα φορτία εργασίας, τα παραδοσιακά εργαλεία αξιολόγησης όπως το TPC-DS δεν επαρκούν πλέον. Το TPC-DS σχεδιάστηκε για στατικές αναλυτικές εφαρμογές και δεν αντανακλά τις σύγχρονες απαιτήσεις, όπου τα δεδομένα είναι δυναμικά και οι εργασίες έντονα ταυτόχρονες.

Οι σύγχρονες μορφές Log-Structured Tables (LSTs)—όπως τα Delta Lake, Apache Iceberg και Apache Hudi—προσφέρουν προηγμένες δυνατότητες όπως ACID συναλλαγές, μεταβολή σχήματος, μεταβολές σε επίπεδο εγγραφής και ερωτήματα "time-travel". Αυτές οι δυνατότητες απαιτούν νέα κριτήρια αξιολόγησης που δεν καλύπτονται από τα παραδοσιακά benchmarks.

Για να καλύψει αυτό το κενό, η Microsoft ανέπτυξε το LST-Bench, ένα νέο εργαλείο αξιολόγησης σχεδιασμένο ειδικά για LSTs σε περιβάλλοντα cloud. Το LST-Bench προσφέρει ευελιξία και επεκτασιμότητα, γεφυρώνοντας τις ανάγκες μεταξύ παραδοσιακών αναλυτικών φορτίων και σύγχρονων επιχειρησιακών προτύπων.

1.6.1 Σχεδιαστική Προσέγγιση του LST-Bench

Αντί να δημιουργήσουν ένα νέο benchmark από το μηδέν, οι δημιουργοί του LST-Bench επέλεξαν να βασιστούν στο TPC-DS, αξιοποιώντας τα εργαλεία δημιουργίας δεδομένων και το σύνολο ερωτημάτων SQL που ήδη παρέχει. Ωστόσο, αναδιάρθρωσαν το μοντέλο εκτέλεσης των φορτίων εργασίας του TPC-DS, το οποίο αρχικά περιελάμβανε τις βασικές λειτουργίες της **φόρτωσης (Load)**, του μοναδικού χρήστη (Single User) και της **συντήρησης δεδομένων (Data Maintenance)**, ώστε να ενσωματώσουν λειτουργίες ειδικές για LST. Συγκεκριμένα, το LST-Bench ενισχύει το βασικό φορτίο εργασίας με τις εξής επιπλέον εργασίες:

- **Βελτιστοποίηση (Optimize)**: Εκτελεί συμπύκνωση μικρών αρχείων μέσα σε έναν πίνακα για βελτίωση της απόδοσης ερωτημάτων και της αποδοτικότητας αποθήκευσης.
- **Χρονική Αναδρομή (Time Travel)**: Επιτρέπει ιστορικά ερωτήματα σε προηγούμενες εκδόσεις ενός πίνακα μέσω καθορισμένων χρονικών σημείων με σύνταξη SQL.

- **Παραμετροποιημένη Προσαρμοσμένη Εργασία (Parameterized Custom Task)**: Επιτρέπει την εκτέλεση ρουτινών ορισμένων από τον χρήστη για ευέλικτη και επεκτάσιμη μοντελοποίηση φορτίου εργασίας.

1.6.2 Δομή Φορτίου Εργασίας

Η δομική ιεραρχία του benchmark είναι η εξής:

- **Εργασία (Task)**: Αντιπροσωπεύει μια ακολουθία εντολών SQL.
- **Περίοδος (Session)**: Είναι μια λογική ομαδοποίηση εργασιών που προσομοιώνει τη συμπεριφορά ενός χρήστη ή μιας διεργασίας.
- **Φάση (Phase)**: Αποτελείται από πολλαπλές περιόδους που εκτελούνται ταυτόχρονα.

Ένα πλήρες φορτίο εργασίας αποτελείται από μια ακολουθία τέτοιων φάσεων, επιτρέποντας τη μοντελοποίηση σύνθετων και ρεαλιστικών σεναρίων χρήσης.

1.6.3 Πρότυπα Φορτίου Εργασίας

Το LST-Bench ορίζει τέσσερα εξειδικευμένα πρότυπα φορτίου εργασίας. Κάθε πρότυπο έχει σχεδιαστεί ώστε να διερευνά μια συγκεκριμένη διάσταση της συμπεριφοράς των LSTs.

- **WP1 - Μακροζωία (Longevity)**: Αξιολογεί την απόδοση του συστήματος σε βάθος χρόνου μέσω επαναλαμβανόμενων ερωτημάτων και ενημερώσεων, εντοπίζοντας φθορά, κόστος και σταθερότητα I/O.
- **WP2 - Ανθεκτικότητα (Resilience)**: Εξετάζει την ικανότητα του συστήματος να διαχειρίζεται συνεχείς εγγραφές και να διατηρεί την απόδοση μέσω εργασιών βελτιστοποίησης.
- **WP3 - Ταυτόχρονη Ανάγνωση/Εγγραφή (Read/Write Concurrency)**: Προσομοιώνει περιβάλλον πολλαπλών χρηστών με παράλληλες αναγνώσεις και εγγραφές, εστιάζοντας σε πιθανά σημεία συμφόρησης στο επίπεδο αποθήκευσης.
- **WP4 - Χρονική Αναδρομή (Time Travel)**: Αξιολογεί την ακρίβεια και αποδοτικότητα ερωτημάτων σε ιστορικές εκδόσεις δεδομένων μετά από τροποποιήσεις.

1.7 Πειραματική διάταξη και ενσωμάτωση του πλαισίου αξιολόγησης

Τα πειράματα benchmarking πραγματοποιήθηκαν σε ένα εικονικοποιημένο cluster πέντε κόμβων, φιλοξενούμενο στο Εργαστήριο Πληροφορικής (CSLab) της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του ΕΜΠ. Κάθε εικονική μηχανή (VM) διέθετε ταυτόσημες τεχνικές προδιαγραφές για να διασφαλιστεί η συνέπεια και η αντικειμενικότητα στην αξιολόγηση των διαφορετικών μορφών πινάκων. Συγκεκριμένα, κάθε VM περιλάμβανε 16 εικονικούς πυρήνες (Intel Skylake), συχνότητα ρολογιού 2.2 GHz, 16 GiB RAM και 16 GiB swap, και λειτουργικό σύστημα Ubuntu 20.04.6 LTS.

Για τη δημιουργία των δεδομένων χρησιμοποιήθηκε το εργαλείο ανοικτού κώδικα tpcds-kit της Databricks, σύμφωνα με το πλαίσιο LST-Bench της Microsoft, το οποίο βασίζεται στο dataset και το σύνολο ερωτημάτων του TPC-DS. Τα πειράματα διεξήχθησαν σε δύο επίπεδα κλίμακας δεδομένων: 100GB και 1TB. Το μικρότερο dataset χρησιμοποιήθηκε για αρχικές δοκιμές και ρύθμιση παραμέτρων, ενώ το dataset του 1TB παρείχε ένα πιο αντιπροσωπευτικό περιβάλλον για δοκιμές αντοχής σε συνθήκες παραγωγής. Σύμφωνα με τις βέλτιστες πρακτικές του TPC-DS, παρουσιάζονται μόνο τα αποτελέσματα που προέκυψαν με τον συντελεστή κλίμακας 1 TB. Ο συντελεστής κλίμακας 100 GB χρησιμοποιήθηκε αποκλειστικά για δοκιμές και ρυθμίσεις παραμέτρων.

Χρησιμοποιήθηκε ένα σύνολο εργαλείων και τεχνολογιών που συνεργάστηκαν εντός ενός κατανεμημένου συστήματος για την εξασφάλιση συνεπούς συμπεριφοράς και αξιόπιστης αξιολόγησης απόδοσης των table formats που μελετώνται. Συγκεκριμένα, αξιοποιήθηκαν οι πλατφόρμες Apache Spark, Hadoop Distributed File System (HDFS), Apache Hive Metastore και PostgreSQL, οι οποίες διαμορφώθηκαν ώστε να λειτουργούν συνδυαστικά στο cluster, υποστηρίζοντας την αποθήκευση, τη μεταδεδομένη διαχείριση και την εκτέλεση ερωτημάτων.

Η αρχιτεκτονική του συστήματος παρουσιάζεται συνοπτικά στο Σχήμα 1.3, όπου απεικονίζονται οι βασικές υπηρεσίες, τα επιμέρους συστατικά και οι μεταξύ τους συνδέσεις.

Επιπλέον, στο Σχήμα 1.4 παρατίθεται το διάγραμμα ανάπτυξης του συστήματος, το οποίο αναλύει τη φυσική κατανομή των στοιχείων στην υποδομή και τις αλληλεπιδράσεις μεταξύ των εγκατεστημένων υπηρεσιών.

Οι συγκεκριμένες εκδόσεις των λογισμικών που χρησιμοποιήθηκαν για να δημιουργηθεί το περιβάλλον για τα πειράματα, μαζί με τα βασικά αρχεία και τις παραμέτρους των configurations, παρουσιάζονται αναλυτικά στο αγγλικό κείμενο. Επιπλέον, παρουσιάζεται όλη η διαδικασία αρχικοποίησης και σύνδεσης του cluster με το LST-Bench για την εκτέλεση των διαφόρων workloads.

1.8 Αξιολόγηση

Σε αυτή την ενότητα, παρουσιάζουμε την αξιολόγηση των αποτελεσμάτων της συγκριτικής αξιολόγησης, που προέκυψαν από την εκτέλεση των φόρτων εργασίας που περιγράφηκαν προηγουμένως, χρησιμοποιώντας το πλαίσιο LST-Bench. Δεν έγινε καμία ρύθμιση στα LSTs. Εκτελέστηκαν όλα χρησιμοποιώντας τις προεπιλεγμένες παραμέτρους διαμόρφωσης. Η μόνη ρυθμίση που έγινε, αφορά την μηχανή εκτέλεσης Spark.

Το longevity workload ανέδειξε σημαντικές διαφορές στην απόδοση των LSTs. Η CoW διαμόρφωση του Delta Lake εμφάνισε πολύ χαμηλή απόδοση, με λανθάνουσες έως και τέσσερις φορές υψηλότερες από τις υπόλοιπες διαμορφώσεις, πιθανότατα λόγω I/O περιορισμών που εντείνονται σε εικονικοποιημένο περιβάλλον. Αντίθετα, η στρατηγική CoW του Apache Hudi υπερείχε της MoR, δείχνοντας ότι η δομή αρχείων της MoR εισάγει μεγαλύτερο υπολογιστικό κόστος στα queries. Το Apache Iceberg παρουσίασε τη βέλτιστη απόδοση, με τις CoW και MoR στρατηγικές να επιτυγχάνουν εξαιρετικά αποτελέσματα, ενώ η MoR εμφάνισε σταθερά τη χαμηλότερη λανθάνουσα. Τα αποτελέσματα αυτά αναδεικνύουν την υψηλή αποδοτικότητα του Iceberg, τα σχετικά πλεονεκτήματα και μειονεκτήματα του Hudi και ένα κρίσιμο σημείο αποτυχίας του Delta Lake.

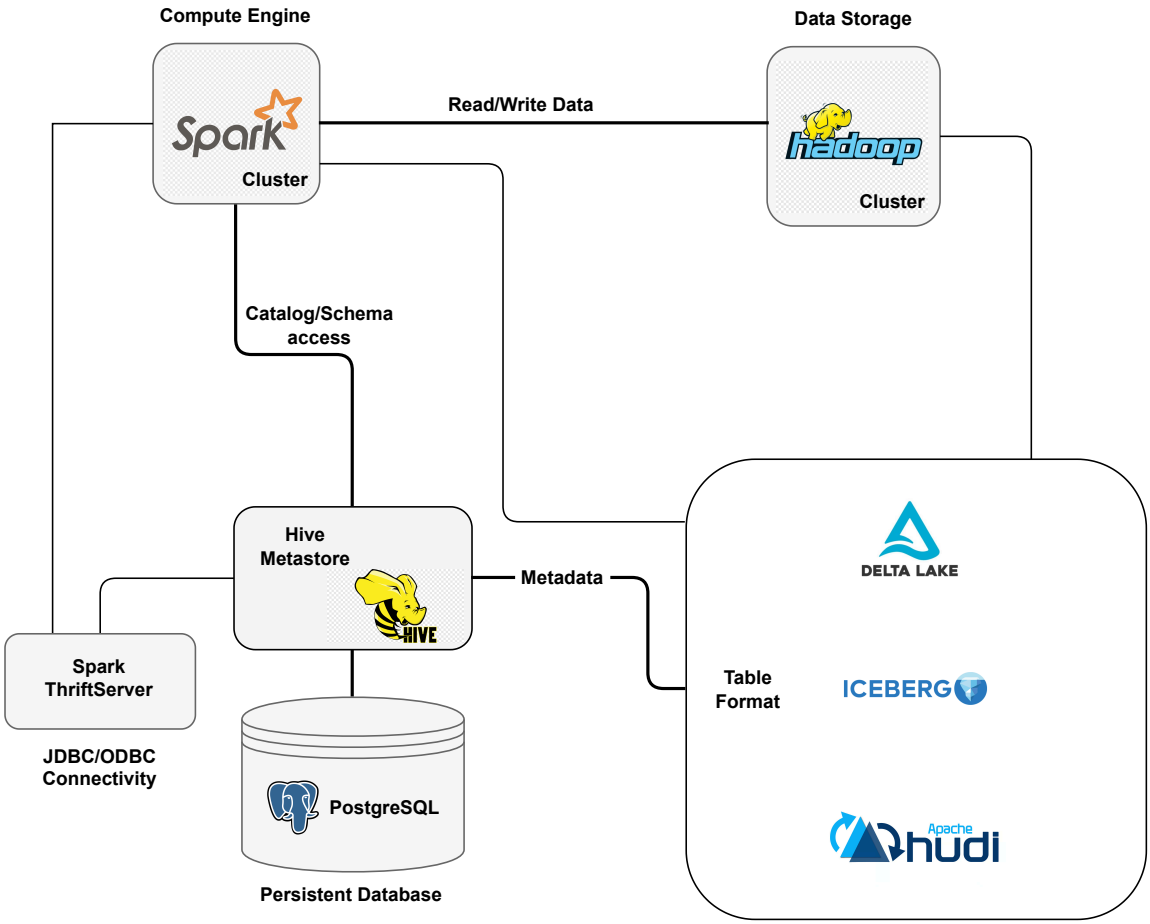

Figure 1.3. *Αρχιτεκτονική του Συστήματος*

Στη συνέχεια, αξιολογήσαμε το resilience workload. Η CoW στρατηγική του Delta Lake παρουσίασε υψηλές και αυξανόμενες καθυστερήσεις κατά τη φάση των ενημερώσεων, ενώ το Optimize μείωσε σημαντικά τις λανθάνουσες χρόνου μόνο μετά το πρώτο βήμα συμπίεσης. Το Apache Hudi εμφάνισε πολύ υψηλές λανθάνουσες τόσο στη φάση των ενημερώσεων όσο και κατά το Optimize, γεγονός που υποδηλώνει χαμηλή κλιμακωσιμότητα· ειδικά η MoR στρατηγική επιδείνωσε την απόδοση των ερωτημάτων μετά το Optimize. Αντίθετα, το Apache Iceberg επέδειξε εξαιρετικά χαμηλές και σταθερές λανθάνουσες τόσο στις ενημερώσεις όσο και στα queries, ενώ η διαδικασία Optimize ολοκληρώθηκε με ελάχιστο κόστος. Συνολικά, τα αποτελέσματα δείχνουν ότι το Iceberg προσφέρει την πιο αποδοτική και ανθεκτική λύση για workloads που απαιτούν συχνή συντήρηση δεδομένων, αποδοτική συμπίεση και σταθερή απόδοση ερωτημάτων.

Το WP-3 workload, σε αντίθεση με τα προηγούμενα, αφορούσε την ταυτόχρονη εκτέλεση

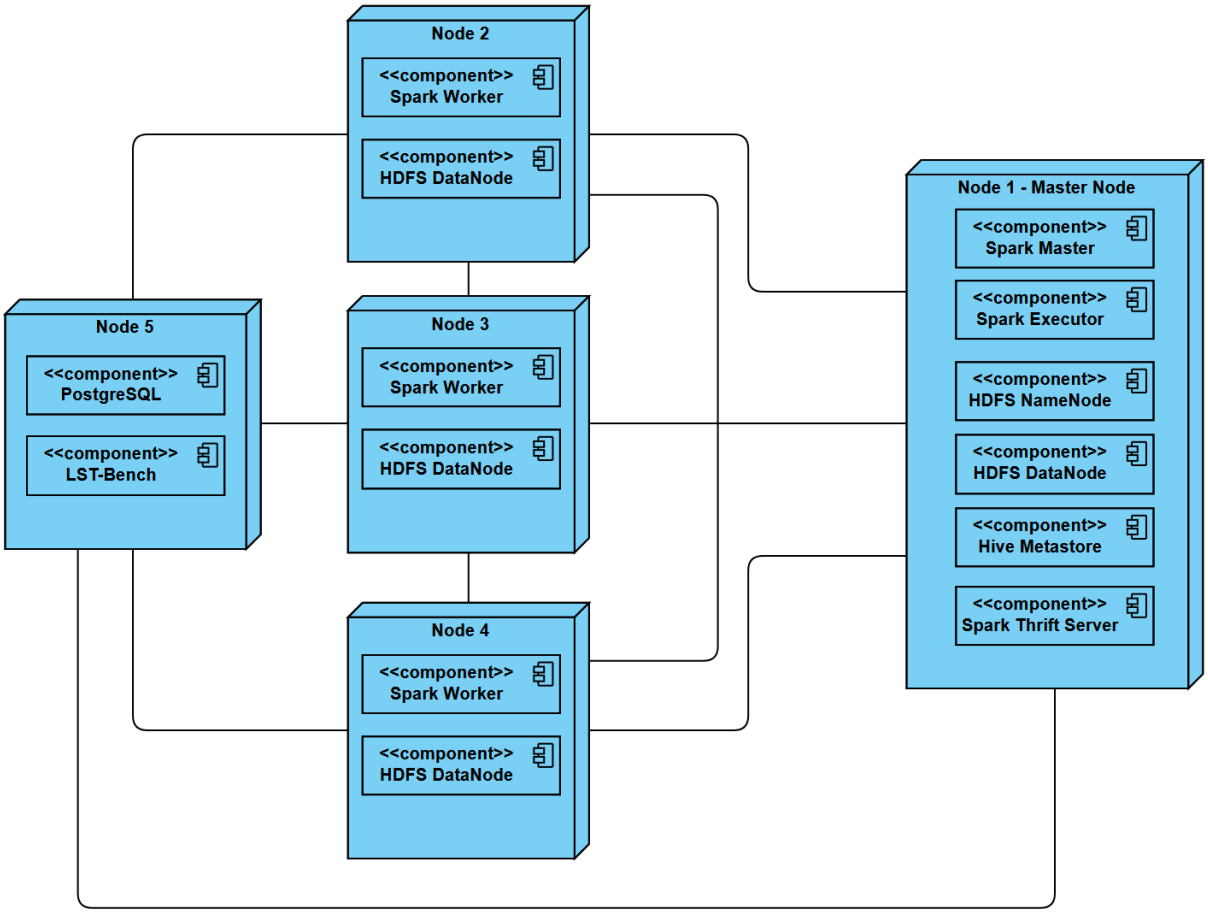

Figure 1.4. *Διάγραμμα Ανάπτυξης του Συστήματος*

queries και λειτουργιών συντήρησης. Το Apache Iceberg (CoW και MoR) κατέγραψε τη χαμηλότερη επιβάρυνση, με χρόνους ολοκλήρωσης πολύ κοντά στους αντίστοιχους διαδοχικούς, επιδεικνύοντας εξαιρετική απομόνωση ανάμεσα σε ενημερώσεις και ερωτήματα. Το Delta Lake CoW παρουσίασε μέτριες επιβαρύνσεις, που οφείλονται κυρίως στις ήδη υψηλές καθυστερήσεις ενημέρωσης, χωρίς όμως να οδηγεί σε σοβαρή υποβάθμιση της απόδοσης κατά τη σύγκλιση. Αντίθετα, το Apache Hudi εμφάνισε σημαντικά προβλήματα κλιμάκωσης υπό συνθήκες ταυτόχρονης εκτέλεσης, με το MoR να καταγράφει τις πιο σοβαρές καθυστερήσεις, φτάνοντας σε σημεία όπου ο παράλληλος χρόνος υπερέβαινε ακόμη και το άθροισμα των διαδοχικών φάσεων. Συνολικά, το Iceberg αναδεικνύεται ως η πιο αξιόπιστη και αποδοτική επιλογή για περιβάλλοντα που απαιτούν ταυτόχρονες λειτουργίες ανάγνωσης, εγγραφής και συντήρησης δεδομένων.

Τέλος, κατά την αξιολόγηση του time travel workload, το Apache Iceberg (CoW και MoR) παρουσίασε εξαιρετική απόδοση, με χαμηλές και σταθερές λανθάνουσες τόσο για την αρχικοποίηση όσο και για τα ιστορικά ερωτήματα, αποδεικνύοντας την αποτελεσματικότητα της αρχιτεκτονικής του στη διαχείριση μεταδεδομένων και στιγμιότυπων πίνακα. Αντίθετα, το Delta Lake, παρότι υποστηρίζει time travel, εμφάνισε σημαντικά υψηλότερες καθυστερήσεις, γεγονός που περιορίζει την πρακτικότητα της χρήσης του για συχνά ή διαδραστικά ιστορικά ερωτήματα. Το Apache Hudi δεν περιλαμβάνεται στη σύγκριση, καθώς η επιλεγμένη έκδοση δεν υπο-

στηρίζει time travel. Συνολικά, το Iceberg αναδεικνύεται ως η βέλτιστη λύση για περιβάλλοντα που απαιτούν ταχύ και αποδοτικό χειρισμό ιστορικών δεδομένων.

1.9 Επίλογος

Η μελέτη αξιολόγησε τρεις κορυφαίες μορφές πινάκων Data Lake (Delta Lake, Apache Hudi και Apache Iceberg), ως προς την απόδοση σε ενημερώσεις, ταυτόχρονες εργασίες και time travel queries, χρησιμοποιώντας το lst-bench σε Apache Spark πάνω σε HDFS. Το Apache Iceberg αποδείχθηκε το πιο ανθεκτικό και αποτελεσματικό σε δυναμικά περιβάλλοντα, ενώ το Delta Lake αντιμετωπίζει προβλήματα σε update-heavy και time travel σενάρια, και το Apache Hudi παρουσιάζει περιορισμούς κλιμάκωσης και συμφόρησης. Επισημαίνεται ότι οι εκδόσεις των LSTs που χρησιμοποιήθηκαν ήταν: Delta Lake 2.2.0, Apache Hudi 0.12.2 και Apache Iceberg 1.1.0, και δεδομένου του ταχύτατου ρυθμού ανάπτυξης, οι νεότερες εκδόσεις μπορεί να παρουσιάζουν διαφορετική συμπεριφορά. Παρ' όλα αυτά, η μελέτη παρέχει ένα λειτουργικό και δοκιμασμένο πλαίσιο αναφοράς, το οποίο μπορεί να χρησιμοποιηθεί για την λήψη αποφάσεων και την καθοδήγηση μελλοντικών προσπαθειών ανάπτυξης στο συνεχώς εξελισσόμενο τοπίο των εν λόγω αρχιτεκτονικών.

Chapter 2

Introduction

2.1 Motivation

The exponential growth of data in recent years has led to a paradigm shift in how organizations store, manage, and analyze information. Traditional data storage and processing systems, while effective for structured data, are increasingly being challenged by the volume, variety, and velocity of modern datasets. These challenges have driven the evolution of data lakes—centralized repositories that store raw data in its native format and at scale—as a foundational component of modern data architectures.

Initially, data lakes relied on basic object storage systems that allowed large-scale data ingestion and persistence, often decoupled from compute resources. However, as the need for sophisticated data governance, analytics performance, and multi-modal access increased, traditional data lakes exposed notable limitations—ranging from poor query performance to the absence of transactional guarantees and lack of schema enforcement. This led to operational inefficiencies and a phenomenon commonly described as the transformation of "data lakes" into "data swamps."

To address these limitations, a new class of open table formats has emerged, adding database-like functionalities to data lakes. These technologies—commonly referred to as data lake table formats—allow for schema evolution, ACID transaction support, time travel queries, and efficient file management on top of raw object storage. Among the most prominent solutions in this domain are Apache Iceberg, Apache Hudi, and Delta Lake. Each of these technologies introduces architectural innovations that improve the manageability, reliability, and performance of modern data lakes.

While Apache Iceberg, Apache Hudi, and Delta Lake all aim to solve similar problems in the realm of data lake management, their design choices, ecosystem integrations, and feature sets vary significantly. Organizations seeking to adopt one of these solutions face a complex evaluation process, often influenced by factors such as compatibility with existing data pipelines, query performance, metadata handling, and operational overhead. There is currently a lack of comprehensive comparative studies that critically assess these three technologies in a unified and practical context. This creates a knowledge gap for practitioners and decision-makers who must select an appropriate architecture for their specific data needs.

The objective of this thesis is to conduct a comparative evaluation of the technologies Apache Iceberg, Apache Hudi, and Delta Lake through benchmarking experiments, with a focus on the

performance of different workload types across varying data scales. The study aims to generate measurable and reproducible results that offer practical insights into how these table formats behave under real-world conditions. By doing so, it seeks to support organizations in making informed decisions regarding the selection of an appropriate lakehouse architecture, tailored to their operational needs and data volume.

2.2 Related Work and Benchmarking Framework

A significant contribution to the study of modern data lake architectures comes from Microsoft's recent work, *"LST-Bench: Benchmarking Log-Structured Tables in the Cloud"*[3]. This study introduces LST-Bench, a benchmarking framework specifically designed to evaluate the performance and functionality of open table formats—namely Apache Iceberg, Apache Hudi, and Delta Lake—within cloud-based data lake environments.

LST-Bench builds upon the well-established TPC-DS benchmark, extending it with capabilities tailored to the unique characteristics of log-structured table formats. It enables rigorous, repeatable testing of critical features such as data ingestion performance, query latency, schema evolution, time-travel support, and file compaction efficiency. The framework also captures telemetry from both the compute engine and cloud storage services, offering a holistic view of system behavior under realistic workloads.

This thesis adopts the LST-Bench methodology as the foundation for its comparative evaluation of Apache Iceberg, Apache Hudi, and Delta Lake. By leveraging Microsoft's benchmark design and performance metrics, the study aims to provide a practical, data-driven assessment of each format's strengths and limitations. However, it is conducted in a different execution environment—namely, on-premises HDFS storage and using Apache Spark as the sole processing engine—reflecting a distinct architectural context.

This alignment in benchmarking focus, despite differences in infrastructure and scope, underscores the broader relevance of evaluating open table formats across diverse deployment models.

2.3 Thesis Structure

In the following chapters, the thesis explores the technical foundations, implementation, and evaluation of log-structured table formats within a data lakehouse architecture. Chapter 3 provides the necessary background on data warehouses, data lakes, and the evolution toward lakehouse systems. Chapter 4 presents the implementation details of the lakehouse stack, focusing on the storage, metadata, and processing layers, with emphasis on Delta Lake, Apache Iceberg, and Apache Hudi.

Chapter 5 introduces the LST-Bench framework, outlining its design, workload patterns, and enhancements tailored for evaluating log-structured formats. Chapter 6 describes the experimental setup, including hardware specifications, dataset generation, system initialization, and benchmark integration. Finally, Chapter 7 presents the evaluation results, comparing performance across formats and workloads, and discussing the implications of the findings.

Chapter 3

Background

In the era of data-driven decision-making, organizations must process massive volumes of structured, semi-structured, and unstructured data—often in real time. Traditional systems like relational databases and classical data warehouses struggle to meet the demands posed by big data's scale, diversity, and speed. These limitations have spurred the adoption of more scalable, flexible architectures: initially data lakes, and more recently, the lakehouse model.

3.1 Data Warehouses

The concept of data warehouses (DWs) was introduced in 1988 by Barry Devlin and Paul Murphy who proposed an architectural model for business information systems in their *IBM Systems Journal* article, "An Architecture for a Business and Information System"[4]. According to the definition by Inmon, *"a data warehouse is a subject-oriented, nonvolatile, integrated, time-variant collection of data designed to support decision-making processes"*.

Formally, a DW is a large, centralized repository that stores integrated data from various sources, in a well-structured format to support enterprise-wide analytics and decision making. The process of compiling data into such a system is known as data warehousing and involves a sequence of stages typically referred to as the Extract, Transform, and Load (ETL) process. ETL tools convert data into a consistent format so that it can be efficiently analyzed and queried when it is inside the warehouse. A simple representation of a typical data warehouse is shown in Figure 3.1.

Unlike Online Transaction Processing (OLTP) systems, which support the real-time automation of operational activities such as transaction recording and customer interactions, data warehouses are optimized for Online Analytical Processing (OLAP). OLAP systems enable users to analyze data along multiple dimensions at once, enabling faster processing and more insightful analysis. Common uses of OLAP include data mining and business intelligence apps, complex analytical calculations, predictive scenarios, budgeting and forecasting[5].

While data warehouses don't generally involve OLTP systems, the data recorded in databases by OLTP systems is typically fed to the warehouse, where an OLAP system enables analysis[5].

The architecture of a data warehouse often follows a three-tier model[5]:

- The bottom tier handles data storage and cleansing, integrating inputs from multiple sources.

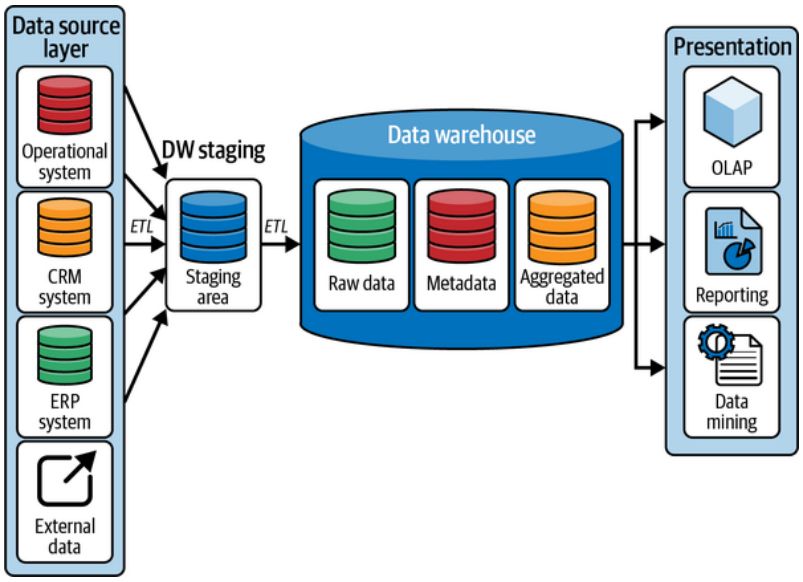

Figure 3.1. *Data warehouse, image source: https://books.google.gr/books?id=CSfdEAAAQBAJ*

- The middle tier consists of the OLAP engine responsible for analytical operations.

- The top tier provides a front-end user interface or reporting tool used for querying, visualization, and business reporting.

Data warehouses have long supported the business community by delivering reliable, high-quality, and standardized data across departments—providing a solid foundation for consistent reporting, historical trend analysis, and dependable decision-making.

However, the fast rise of the internet and social media and the availability of multimedia devices such as smartphones disrupted the traditional data landscape, giving rise to the term big data. Big data is defined as data that arrives in ever higher volumes, with more velocity, in a greater variety of formats with higher veracity. These are known as the four Vs of data[6].

Data warehouses face significant challenges when it comes to addressing these four Vs of big data. As data volumes grow exponentially into the petabyte range, these systems struggle with storage scalability without incurring high costs. Their lack of in-memory and parallel processing capabilities, limit them from scaling vertically. They are also ill-equipped to manage the high velocity of modern data streams, as they do not support real-time processing and rely on rigid ETL windows that strain the infrastructure. While data warehouses are very good at storing structured data, they are not well suited to store and query the variety of semi-structured or unstructured data. Veracity is another weak point—traditional warehouses provide minimal metadata related to data lineage or quality, making trust in the data harder to establish. Their reliance on proprietary formats and SQL-based query tools also makes them incompatible with modern data science and machine learning ecosystems. These limitations make data warehouses costly to implement and maintain, often leading to project delays or failures, and making it difficult for businesses to adapt to the fast-evolving demands of today's data-driven landscape[6].

3.2 Data Lakes

With the huge amount of data around, the need to have better solutions for storing and analyzing large amounts of semi-structured and unstructured data to gain relevant information and valuable insight became apparent. Traditional schema-on-write approaches such as the extract, transform and load (ETL) process are inefficient for such data management requirements. This gave rise to another popular modern enterprise data management scheme known as data lakes.

The concept of data lake was first coined in the industry. In 2010, it was first proposed by Pentaho CTO James Dixon, as a solution that handles raw data from one source and supports diverse user requirements[7]. In contrast to data warehouses where data must be strictly structured and its intended use clearly defined in advance, requiring extensive ETL processes, data lakes allow raw data to be stored in its native format, minimizing or postponing the need for costly preprocessing.

Formally, a data lake is a cost-effective central repository to store structured, semi-structured, or unstructured data at any scale, in the form of files and blobs. The term "data lake" came from the analogy of a real river or lake, holding the water, or in this case data, with several tributaries that are flowing the water (aka "data") into the lake in real time[6]. A canonical representation of a typical data lake is shown in Figure 3.2.

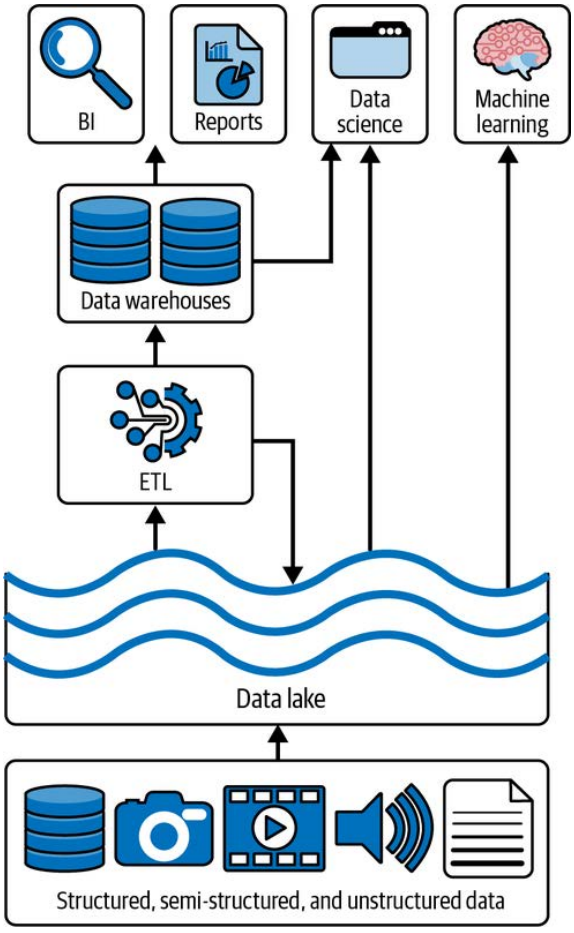

Figure 3.2. *Canonical data lake, image source: https://books.google.gr/books?id=CSfdEAAAQBAJ*

In the early days, data lakes and big data solutions were typically built using on-premises clusters powered by Apache Hadoop, an open-source framework. Hadoop made it possible to store and process massive datasets—anywhere from gigabytes to petabytes—by distributing the workload across clusters of inexpensive, standard hardware. Instead of relying on a single powerful machine, Hadoop's MapReduce framework allowed compute tasks to be performed in parallel across multiple nodes.

The Hadoop Distributed File System (HDFS), a core component, was engineered to handle vast amounts of data while being fault-tolerant and optimized for low-cost hardware.

Around 2015, a shift began toward cloud-based data lakes like Amazon S3, Azure Data Lake Storage Gen2 (ADLS), and Google Cloud Storage (GCS). These services began to replace HDFS thanks to their superior reliability (offering SLAs exceeding 99.99999999%), built-in geo-replication, and extremely low costs—including even cheaper options for archival (cold) storage.

At a higher level, data in these cloud lakes is stored as blobs—raw, unstructured chunks of data. This flexibility allows for the storage of diverse data types, from multimedia files to semi-structured formats like JSON. On top of blob storage, cloud platforms add features like file-system interfaces and fine-grained security controls, making it easy to manage structured datasets as well. Thanks to their high-speed data transfer capabilities, these modern data lakes also support real-time use cases such as continuous ingestion from IoT devices or media streaming.

Modern data processing workflows rely on compute engines to transform substantial volumes of data in a manner similar to traditional Extract, Transform, Load (ETL) pipelines. These processes serve various downstream systems, including data warehouses, artificial intelligence (AI), and machine learning (ML) tools. Streaming data can be ingested and stored in real-time databases, while analytical insights are typically generated through business intelligence (BI) and reporting applications.

3.2.1 Key Components

Data lakes are enabled through a variety of components[6].

- **Storage**: Data lakes require large, scalable and durable storage infrastructures, often provisioned in cloud environments. These storage solutions must support interoperability with third-party tools, libraries, and drivers. A key architectural advantage of data lakes is that they distinguish the concepts of storage and compute, enabling independent and elastic scalability. Additionally, high-bandwidth ingress and egress capabilities are essential for managing large-scale batch loads or continuous streams of data, such as Internet of Things (IoT) telemetry and multimedia content.

- **Compute**: Data lakes also require substantial computational power in order to process the large amount of data retained in the storage layer. Various cloud platforms offer compute engines, with Apache Spark serving as the industry-standard engine for large-scale data lake operations. Spark, an open-source unified analytics engine, can be deployed through cloud-native services like Databricks or comparable proprietary platforms. These systems utilize compute clusters—aggregations of compute nodes that collaboratively execute large-scale data processing tasks.

- **Data Formats**: The shape of the data on disk defines the format. Data lakes commonly employ standardized, open-source formats such as Parquet, Avro, JSON, and CSV. These formats ensure broad compatibility with analytical tools and efficient storage performance.

- **Metadata**: Cloud storage systems incorporate metadata to provide essential context about the data. This includes temporal information (e.g., write and access timestamps), data schemas, and descriptive tags that indicate data ownership and usage characteristics.

3.2.2 Advantages of Data Lake Architectures

Data lakes offer several compelling advantages. They provide a unified repository for all organizational data assets, independent of their structure or origin. Also, their reliance on open data formats such as Parquet and Avro facilitates seamless integration with diverse platforms and tools. By leveraging mature cloud-based storage architectures, data lakes benefit from high scalability, monitoring capabilities, ease of deployment, and cost-effective storage. Furthermore, deployment and maintenance can be automated through established Infrastructure-as-Code tools like Terraform.

Unlike conventional data warehouses, data lakes support all data types—including structured, semi-structured, and unstructured content—making them especially suited for complex workloads such as media processing. Their high-throughput data pipelines also support real-time applications such as IoT sensor data ingestion, media streaming, and behavioral analytics from web clickstreams.

3.2.3 Limitations

Despite their widespread adoption and architectural flexibility, traditional data lakes have surfaced several operational and technical challenges that can hinder their effectiveness. Although the underlying cloud-based storage solutions remain cost-efficient, establishing and managing a fully functional data lake architecture demands significant expertise. This requirement often translates into increased operational costs—either through highly skilled internal personnel or reliance on specialized consulting services.

While ingesting raw data into the data lake is relatively straightforward, transforming that data into analytically valuable formats introduces substantial processing complexity and financial overhead. Moreover, conventional data lakes exhibit high query latency, rendering them unsuitable for interactive or real-time data exploration. Consequently, organizations frequently resort to loading transformed data into a secondary analytical layer—typically a data warehouse—which extends the time to derive actionable insights. This necessity gave rise to the hybrid "data lake + data warehouse" architecture, a model that dominated enterprise data ecosystems for years but is now being gradually supplanted by more integrated paradigms like the "lakehouse" architecture.

Traditional data lakes operate on a schema-on-read principle, where data is ingested without rigid schema enforcement and interpreted only during read operations. Although this model provides flexibility, it also introduces the risk of poor data quality and inconsistency, which can degrade the data lake into what is colloquially referred to as a "data swamp."

In addition, data lakes lack transactional guarantees. Data can only be appended, not modified

in place, making simple updates computationally expensive as they often require rewriting existing data files. This limitation contributes to the so-called small file problem, where numerous tiny files are generated for individual data entities. When unmanaged, the proliferation of small files degrades read performance and inflates storage costs by introducing data redundancy and fragmentation. To counter this, administrators must perform regular compaction processes to merge small files into larger, more efficient storage units—adding further complexity to data lake maintenance[6].

3.3 Evolution to Data Lakehouses

Armbrust, Ghodsi, Xin, and Zaharia first introduced the concept of the data lakehouse in 2021. The authors define a lakehouse as "a data management system based upon low-cost and directly accessible storage that also provides analytics DBMS management and performance features such as ACID transactions, data versioning, auditing, indexing, caching and query optimization."

A lakehouse is a unified architecture that combines the best aspects of data lakes and data warehouses. It offers the flexibility, scalability, and cost-effectiveness of a data lake, while also incorporating the data management features and ACID transaction support of data warehouses, addressing the limitations of both.

Lakehouses are an especially good match for most, if not all, cloud environments with separate compute and storage resources. Different computing applications can run on demand on completely separate computing nodes, such as a Spark cluster, while directly accessing the same storage data. It is, however, conceivable that one could implement a lakehouse over an on-premises storage system such as the aforementioned HDFS[6]. An overview of the lakehouse architecture is show in Figure 3.3.

The lakehouse architecture eliminates the need to maintain separate copies of data in both a data lake and a data warehouse. Instead, data can be accessed directly from the lake offering performance on par with traditional warehouses. By continuing to use cost-efficient cloud storage and avoiding data duplication, organizations can achieve substantial savings in both infrastructure and staffing. Reduced data movement and simplified ETL processes lower the risk of data quality issues, while the lakehouse's ability to support both large-scale raw data and structured models accelerates development and shortens the time to value.

The evolution from data warehouses to data lakes to a data lakehouse architecture is shown in Figure 3.4.

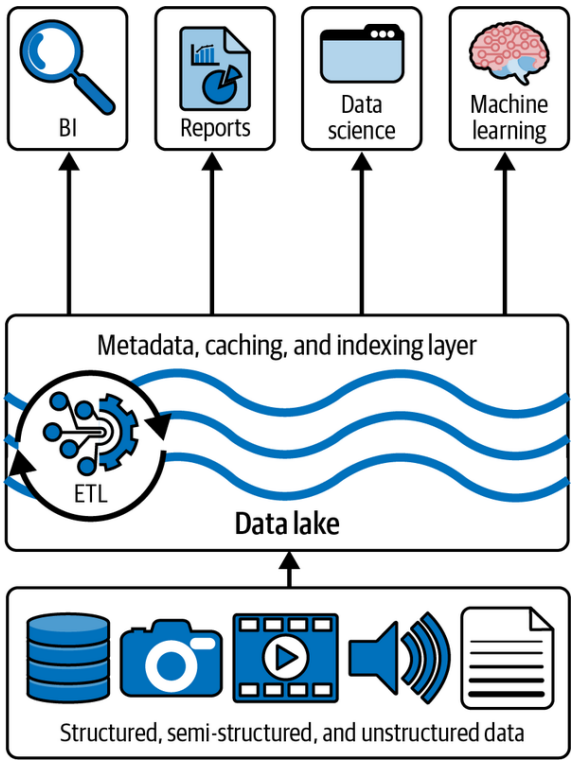

Figure 3.3. *Lakehouse architecture, image source: https://books.google.gr/books?id=CSfdEAAAQBAJ*

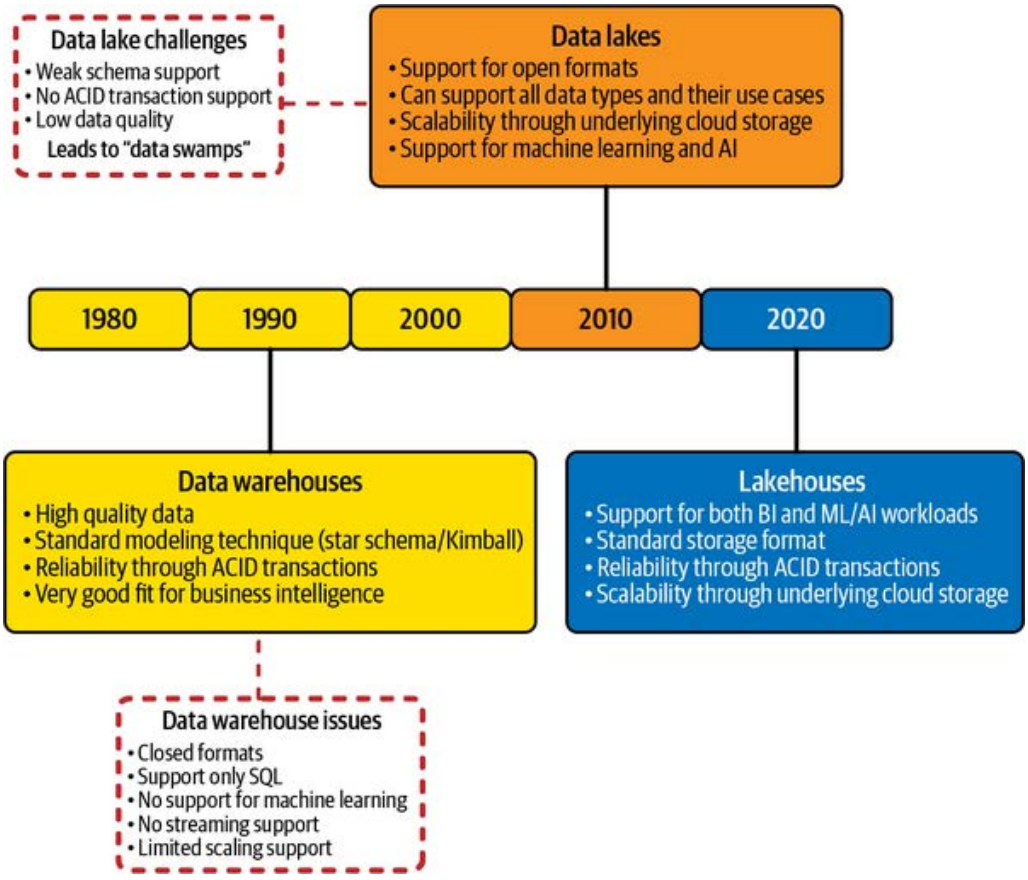

Figure 3.4. *Evolution of data architecture, image source: https://books.google.gr/books?id=CSfdEAAAQBAJ*

Chapter 4

Implementing a Data Lakehouse

A Data Lakehouse, as mentioned, is a data management system based on low cost and directly-accessible storage that also provides traditional analytical DBMS management and performance features such as ACID transactions, data versioning, auditing, indexing, caching, and query optimization.

The lakehouse architecture is made of 3-layers. The bottom layer, is the object storage layer. Lakehouses as previously mentioned, leverage cost-effective and scalable object storages, such as Amazon S3, ADLS or HDFS, while storing the data in an open, standardized format like Apache Parquet. Atop the object store, is the metadata layer. This layer, maintains a consistent and versioned definition of table contents, and ensures reliable data management, supporting ACID transactions over the underlying files[8]. Lastly, the top layer of the lakehouse architecture is made up of high-performance query and processing engines, such as Apache Spark or Trino, which leverage underlying cloud compute resources[6]. The 3-layers a lakehouse is made of, are shown in Figure 4.1.

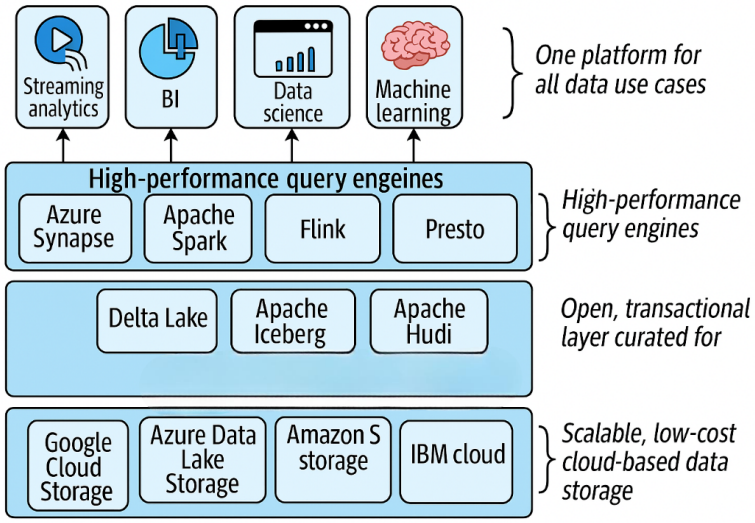

Figure 4.1. *Lakehouse layered architecture, original image source: https://books.google.gr/books?id=CSfdEAAAQBAJ*

In the remainder of this chapter, we will examine in detail the individual layers of the lakehouse architecture, beginning with the storage layer and progressing through to the processing layer that enables data querying and analytics.

4.1 Storage Layer

At the foundation of the lakehouse architecture lies the storage layer, which is responsible for persisting large volumes of data in a cost-effective and scalable manner. This layer can be implemented either using cloud-based object storage solutions, such as Amazon S3 and Azure Data Lake Storage, which offer elasticity and managed infrastructure, or through on-premises systems.

One of the most mature and widely adopted storage system in on-premises environments is the Hadoop Distributed File System (HDFS). Originally developed as part of the Apache Hadoop project, HDFS is specifically designed for handling massive volumes of data distributed across clusters of commodity hardware.

HDFS operates on a master-slave architecture, comprising a NameNode, which manages the file system namespace and metadata, and DataNodes, which are responsible for storing actual data blocks. Files are divided into blocks and replicated across multiple DataNodes to ensure fault tolerance and high availability. In the event of node failure, data can be reconstructed from replicas, making HDFS exceptionally resilient for data-intensive applications.

Some key advantages of HDFS in the context of a lakehouse include:

- **High Throughput and Scalability**: HDFS is optimized for batch processing with high data throughput, allowing lakehouse systems to scale horizontally as data volumes grow.
- **Write Once, Read Many Semantics**: This design pattern aligns well with analytical workloads where data is ingested in bulk and accessed frequently for querying and reporting.
- **Compatibility with Open Formats**: HDFS supports storage of data in open formats such as Apache Parquet and ORC, which are essential for interoperability with table formats like Delta Lake, Apache Iceberg, and Apache Hudi.
- **Integration with Big Data Ecosystems**: As a core component of the Hadoop ecosystem, HDFS is natively compatible with processing frameworks such as Apache Spark, Hive, and MapReduce.

Although cloud-based object storage has gained traction due to its elasticity and managed services, HDFS remains a robust and cost-effective solution for organizations operating in self-managed or hybrid cloud environments, particularly where data sovereignty, latency, or infrastructure control is a concern.

4.2 Metadata Layer

Several open-source projects have emerged to fulfill the metadata layer and the advanced data management capabilities required for a functional lakehouse. Among the most prominent are Delta Lake, Apache Iceberg, and Apache Hudi. These technologies extend the basic capabilities of data lakes by introducing features such as ACID transactions, time travel and schema evolution.

4.2.1 Delta Lake

Delta Lake[9] is an open source storage layer that supports ACID transactions, scalable metadata handling, and unification of streaming and batch data processing. It was initially designed to work with Apache Spark and large-scale data lake workloads. However, as it has evolved, Delta Lake has been optimally designed to work numerous workloads (small data, medium data, big data, etc.). It has also been designed to work with multiple frameworks (e.g., Apache Spark, Apache Flink, Trino, Presto, Apache Hive, and Apache Druid), services (e.g., Athena, Big Query, Databricks, EMR, Fabric, Glue, Starburst, and Snowflake), and languages (.NET, Java, Python, Rust, Scala, SQL, etc.)[10].

A Delta Lake table consists of several integral components that together enable a robust, scalable, and high-performance data management system [10].

- **Data files** are stored in the Apache Parquet format, chosen for its efficiency in handling large datasets, and reside in distributed file systems such as HDFS, Amazon S3, Azure Blob Storage, Google Cloud Storage, or MinIO.
- The **transaction log**, also known as the Delta log, is fundamental to ensuring ACID compliance. It records every change to the table in JSON format, with each entry capturing metadata such as the type of operation, affected files, and table schema at the time.
- **Metadata** includes schema definitions, partitioning details, and configuration settings, and is embedded within the transaction log. It is accessible through interfaces such as SQL, Apache Spark, Rust, and Python, enabling enforcement of schema and optimization strategies.
- The schema defines the table's structure—including columns and data types—and is enforced upon writing data. Delta Lake supports **schema evolution**, allowing structural changes as data requirements evolve.
- **Checkpoints** are periodic snapshots of the transaction log, stored in Parquet format, which accelerate query performance and recovery by allowing systems to bypass replaying the full log history. By default, Delta Lake generates checkpoints every ten transactions.

To further elaborate on the architecture of Delta Lake, it's important to highlight how it handles modifications such as deletes. On object storage systems, rather than modifying existing Parquet files, it is faster to create new files containing only the unaffected rows. This approach enhances performance and aligns with the principle of multiversion concurrency control (MVCC)—a technique that maintains multiple versions of data, enabling safe, concurrent read and write operations. This technique also allows Delta Lake to provide time travel, allowing users to query historical versions of the data[10].

4.2.2 Apache Iceberg

Apache Iceberg[11] is an open-source table format designed to efficiently manage analytical datasets at petabyte scale across distributed data systems and cloud object stores. It provides advanced capabilities such as schema evolution, ACID transaction support, hidden partitioning,

and optimized query performance. Iceberg is compatible with a wide range of processing engines—including Apache Spark, Trino, Flink, and Presto—and enables consistent, concurrent read and write operations, making it well suited for modern, large-scale data architectures.

Apache Iceberg introduces several core concepts that support scalable and efficient data processing[12].

- **Metadata Management**: Iceberg uses a layered metadata system—including metadata files, manifest files, and manifest lists (all in JSON format)—to track schemas, partitions, data files, and their statistics. It supports snapshot-based versioning, allowing consistent table states over time.
- **Catalog Layer**: The catalog maintains a reference to the most recent metadata file for each table. Whenever a change occurs, a new metadata file is created, and the catalog updates the pointer to reflect the latest table state. This mechanism enables ACID transactions by ensuring that ongoing modifications remain isolated and are only visible once committed, facilitating consistent reads and simplifying concurrency control.
- **Schema Evolution**: Iceberg allows seamless schema changes (e.g., adding columns) without requiring costly data rewrites. Metadata is updated to reflect schema changes, while existing data remains unaffected.
- **Partitioning**: Iceberg supports advanced partitioning strategies such as range, hash, truncate, and list partitioning. These methods enhance query performance by minimizing data scans.
- **Snapshots**: Every change creates a new snapshot—an immutable view of the table at a specific point in time. This enables time travel and rollback capabilities for historical data access and recovery.

4.2.3 Apache Hudi

Apache Hudi[13] (Hadoop Upserts Deletes and Incrementals) is an open-source data lakehouse framework designed to bring transactional capabilities and efficient data management to cloud-native data lakes. Originally developed at Uber, Hudi enables features such as record-level inserts, updates, and deletes, making it particularly well-suited for incremental data processing and change data capture (CDC) use cases[14].

To support these capabilities, Apache Hudi introduces several core concepts

- **Timeline and Metadata Management**: Hudi maintains a timeline of all actions (commits, cleanups, compactions, rollbacks) performed on a table. Each action is associated with an instant time, ensuring atomicity and consistency. Metadata is stored alongside data files to track schema, file versions, and partitioning.
- **Table Types**:
  - **Copy-on-Write (CoW)**: Updates create new versions of files, optimizing for read-heavy workloads.

  - **Merge-on-Read (MoR)**: Updates are written to delta logs and merged at read time, optimizing for write-heavy or streaming use cases.

- **Query Types**:
  - **Snapshot Queries**: Provide the latest view of the data.
  - **Incremental Queries**: Retrieve only data that changed since a given point in time.
  - **Read-Optimized Queries**: Access base files without merging logs, offering faster reads.

- **Schema Evolution**: Hudi supports schema changes over time, allowing new fields to be added without rewriting existing data.

- **Compaction and Clustering**: Background services like compaction (for MoR tables) and clustering (to optimize file layout) improve performance and storage efficiency.

- **Time Travel and Rollbacks**: Using the timeline, users can access historical versions of data or roll back to a previous state in case of failure or corruption.

Hudi integrates with popular processing engines such as Apache Spark, Flink, Presto, Trino, and Hive, and is compatible with cloud storage platforms like Amazon S3, Azure Data Lake Storage, and Google Cloud Storage. Its advanced indexing, clustering, and compaction features further enhance performance for both batch and streaming workloads.

4.3 Processing Layer

At the top of the lakehouse architecture resides the processing layer, which is responsible for executing analytical workloads, data transformation pipelines, and real-time processing tasks. Within this layer, while Apache Spark has become one of the most dominant and versatile distributed computing engines for large-scale data processing, alternative frameworks such as Trino and Apache Flink are also widely used, each offering distinct advantages depending on workload characteristics and system requirements.

Apache Spark is an open-source unified analytics engine designed for high-performance processing of big data across single-node machines or clustered environments. Spark's key architectural strength lies in its in-memory computing capabilities, which allow it to outperform traditional batch frameworks like Hadoop MapReduce by orders of magnitude in many scenarios.

Spark's architecture is based on a driver-executor model, where the driver orchestrates job execution across multiple worker nodes that run executors. These executors manage tasks and data caching for parallel processing. The distributed nature of Spark allows it to process terabytes or even petabytes of data efficiently across multiple nodes.

In lakehouse systems, Apache Spark serves as the core query and transformation engine interfacing directly with the storage and metadata layers. Its seamless integration with open table formats—such as Delta Lake, Apache Iceberg, and Apache Hudi—enables Spark to deliver:

- ACID-compliant reads and writes over data stored in distributed file systems

- Declarative SQL queries via Spark SQL, used for interactive and batch analytics
- Support for schema evolution and time-travel queries, depending on the table format
- Unified batch and streaming execution using Spark Structured Streaming
- Integration with machine learning libraries (MLlib) and graph processing (GraphX), enhancing analytical possibilities

In summary, Apache Spark acts as a powerful execution layer in the lakehouse architecture, bridging raw storage and transactional metadata with high-performance, distributed analytics. Its flexibility, scalability, and wide tool support make it a cornerstone technology in any modern data infrastructure.

Chapter 5

Benchmarking Log-Structured Table Formats: The LST-Bench Framework

5.1 Overview and Motivation

As data lake architectures evolve to accommodate increasingly complex, transactional, and mutable workloads, traditional benchmarking tools such as TPC-DS are no longer sufficient for evaluating their performance in full. While originally designed to assess analytical query engines in static environments, TPC-DS lacks the ability to reflect modern operational scenarios where data formats are dynamic and workloads highly concurrent.

Modern Log-Structured Table formats (LSTs)—such as Delta Lake, Apache Iceberg, and Apache Hudi—have introduced advanced data management capabilities to data lakes. These include support for ACID transactions, schema evolution, record-level mutations, and time-travel queries. Such features bring new performance considerations that are not captured by conventional benchmarks. Consequently, there is a growing need for benchmarking frameworks that can effectively model and evaluate the distinct behaviors of LSTs.

To address this gap, Microsoft proposed the LST-Bench framework [3], a benchmarking tool specifically designed to assess the performance and functionality of LSTs in realistic, cloud-based data lake environments. LST-Bench introduces a modular and extensible approach that bridges the gap between traditional analytical workloads and emerging operational patterns found in real-world deployments.

5.2 Benchmarking Limitations of TPC-DS

Although TPC-DS remains a standard benchmark for evaluating analytical query engines, it lacks support for evaluating:

- **Longevity**: referring to the need to support frequent data modifications over extended periods.
- **Resilience**: which involves managing diverse and continuous updates in an optimized table environment.
- **Read/Write Concurrency**: reflecting scenarios in which multiple concurrent sessions perform reads and writes—potentially across distributed compute clusters.

- **Time-Travel**: which enable querying data as it existed at various historical points.

These dimensions are central to evaluating LSTs, which rely on immutable file storage, metadata tracking, and snapshot isolation mechanisms to deliver transactional guarantees and consistent analytical performance.

5.3 Design Approach of LST-Bench

Rather than building a benchmark entirely from scratch, the authors of LST-Bench chose to build upon TPC-DS by reusing its data generation utilities and SQL query suite. However, they restructured the workload execution model to incorporate LST-specific operations such as compaction, schema evolution, and time-travel access.

To capture the behavioral complexity of LSTs, LST-Bench defines a set of workload patterns that simulate realistic usage scenarios. These patterns include both traditional OLAP queries and LST-specific maintenance tasks. Moreover, the benchmark introduces new performance metrics that assess ingestion throughput, metadata overhead, query latency under versioned access, and the system's response to changing data conditions.

5.4 Workloads Patterns in LST-Bench

To comprehensively assess the behavior of LSTs, the LST-Bench framework[3] builds upon the foundational TPC-DS benchmark by incorporating both standard analytical tasks and additional operations tailored to the dynamic nature of LSTs.

5.4.1 Baseline Workload Tasks

TPC-DS provides the foundational tasks upon which LST-Bench extends its evaluation logic. These baseline tasks are defined as follows:

- **Load**: Responsible for populating the benchmark tables with data to support query evaluation.
- **Single User**: Executes a series of complex analytical queries in isolation to measure the system's upper performance limits.
- **Data Maintenance**: Involves insertions and deletions, simulating operational updates that occur in dynamic datasets.

5.4.2 LST-Specific Enhancements

Recognizing that LSTs introduce operational semantics not addressed by standard OLAP benchmarks, LST-Bench augments the base workload with additional tasks to better reflect real-world use cases:

- **Optimize**: Performs compaction of small files within a table to improve query performance and storage efficiency.

- **Time Travel**: Enables historical queries against previous versions of a table by specifying temporal access points using SQL syntax.

- **Parameterized Custom Task**: Allows execution of user-defined routines to support flexible and extensible workload modeling.

5.4.3 Workload Structure

The structural hierarchy of the benchmark is depicted in Figure 5.1. A *task* represents a sequence of SQL statements; a *session* is a logical grouping of tasks intended to simulate the behavior of a single user or process; and a *phase* consists of multiple sessions executed concurrently. A full *workload* comprises a sequence of such phases, allowing for complex and realistic workload modeling.

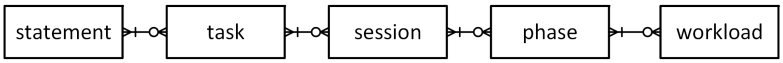

Figure 5.1. *LST Bench: Workload Components, image source: https://www.microsoft.com/en-us/research/blog/lst-bench-a-new-benchmark-tool-for-open-table-formats-in-the-data-lake/*

5.4.4 Workload Patterns

To address evaluation gaps inherent in traditional TPC-DS execution models, LST-Bench defines four specialized workload patterns. Each pattern is designed to probe a specific dimension of LST behavior:

- **WP1 — Longevity**: This pattern assesses long-term system behavior by repeating a sequence of Single User query phases and Data Maintenance operations. The objective is to observe trends in performance degradation, cost impact, and I/O stability as the table evolves over time.

- **WP2 — Resilience**: Designed to evaluate robustness, this workload gradually increases the number of write operations and applies Optimize tasks following each burst. The goal is to measure how efficiently the system reorganizes data and maintains performance under sustained update pressure.

- **WP3 — Read/Write Concurrency**: This workload simulates realistic multi-user environments by executing read (Single User) and write (Data Maintenance and Optimize) operations in parallel. The framework leverages separation of storage and compute to isolate performance bottlenecks attributable to the storage layer.

- **WP4 — Time Travel**: This pattern targets the temporal querying capabilities of LSTs. After a series of Data Maintenance operations, Time Travel tasks are used to issue queries against specific historical snapshots of the table. This enables evaluation of the efficiency and accuracy of versioned access.

The structure of each workload pattern is visualized on Figure 5.2, which provides a clearer and more intuitive representation of the operational logic and evaluation objectives defined within the LST-Bench framework. The figure also illustrates the baseline components defined on the original TPC-DS.

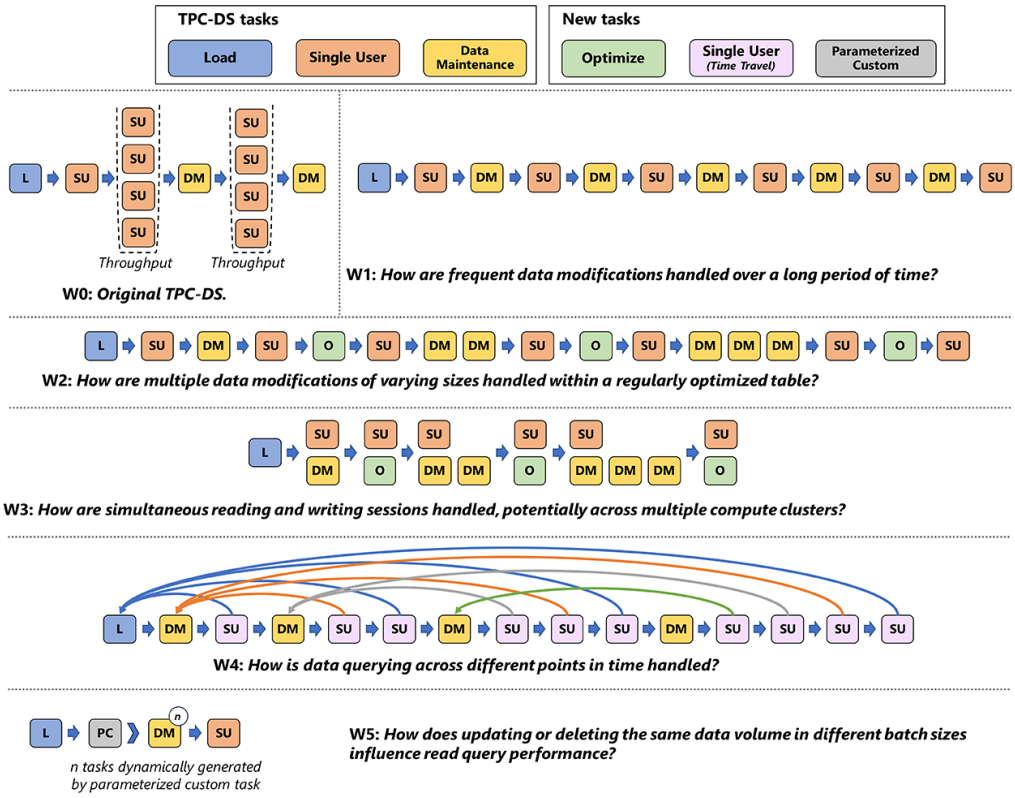

Figure 5.2. *TPC-DS and extensions to evaluate LSTs characteristics https://www.microsoft.com/en-us/research/blog/lst-bench-a-new-benchmark-tool-for-open-table-formats-in-the-data-lake/*

Chapter 6

Experimental Setup and Benchmark Integration

6.1 System Hardware

The benchmarking experiments were conducted on a five-node virtualized cluster, hosted within the Computer Science Laboratory (CSLab) of the School of Electrical and Computer Engineering (ECE), National Technical University of Athens (NTUA). Each virtual machine (VM) was provisioned with identical hardware specifications to ensure consistency and fairness in performance evaluation across the different table formats.

The specifications per VM are as follows:

- **Processor**: 16 virtual CPUs (Intel Core Processor, Skylake family, Model 94)
- **Clock Speed**: 2.2 GHz base frequency
- **Architecture**: x86_64
- **Virtualization**: Fully virtualized via KVM with VT-x support
- **Memory**: 16 GiB of RAM per node
- **Swap**: 16 GiB swap memory per node
- **Cache Hierarchy**: **L1d**: 512 KiB, **L1i**: 512 KiB, **L2**: 64 MiB, **L3**: 256 MiB
- **Operating System**: Ubuntu 20.04.6 LTS

6.2 Dataset Generation

As outlined earlier, Microsoft's LST-Bench framework relies on the TPC-DS dataset as well as its query set. In this study, data was generated using the open-source tpcds-kit tool provided by Databricks[15].

Experiments were conducted at two scale factors: 100GB and 1TB. The 100GB dataset was used primarily for preliminary testing and tuning, while the 1TB scale provided a more representative environment for stress-testing system behavior under production-like data volumes. In accordance with TPC-DS best practices, only the results obtained at the official 1 TB scale factor are presented, while runs at 100 GB were used solely for tuning and validation purposes.

Listing 6.1 illustrates the full process of acquiring and compiling the TPC-DS data generation toolkit from GitHub, followed by dataset generation at the two scale factors (100 GB and 1 TB). It also includes examples of update scenarios for the incremental data modifications.

```
git clone https://github.com/databricks/tpcds-kit.git
cd tpcds-kit/tools
make OS=LINUX

# generate data scale factor 100 and 1000
./dsdgen -scale 100 -dir /mnt/1tb/tpcds_sf_100 -TERMINATE N
./dsdgen -scale 1000 -dir /mnt/1tb/tpcds_sf_1000/ -TERMINATE N

# example to generate update dataset
./dsdgen -scale 1000 -dir /mnt/1tb/tpcds_sf_1000/update/1 -TERMINATE N
    -update 1
./dsdgen -scale 1000 -dir /mnt/1tb/tpcds_sf_1000/update/2 -TERMINATE N
    -update 2
./dsdgen -scale 1000 -dir /mnt/1tb/tpcds_sf_1000/update/3 -TERMINATE N
    -update 3
```

Listing 6.1: TPC-DS Dataset Generation

6.3 Software Environment and Configurations

This section outlines the software components and configurations that form the foundation of the experimental environment. It details the versions, roles, and integration of key systems—namely Apache Spark, Hadoop Distributed File System (HDFS), and Apache Hive Metastore—used throughout the benchmarking process. The selected tools were configured to work together within a distributed cluster to ensure consistent behavior and reliable performance evaluation across all table formats under study.

A high-level overview of the system architecture is presented in Figure 6.1, illustrating the services, the components and the connections between them.

In addition to the architectural overview, Figure 6.2 presents a deployment diagram of the system, detailing the physical distribution of components across the available infrastructure and the interactions between deployed services.

To promote reproducibility and transparency, a subset of system-level packages directly relevant to the experimental stack is summarized in Table 6.1 These include Java runtimes, PostgreSQL drivers, and commonly used development tools required for running and managing Spark, Hive, and the benchmarking framework.

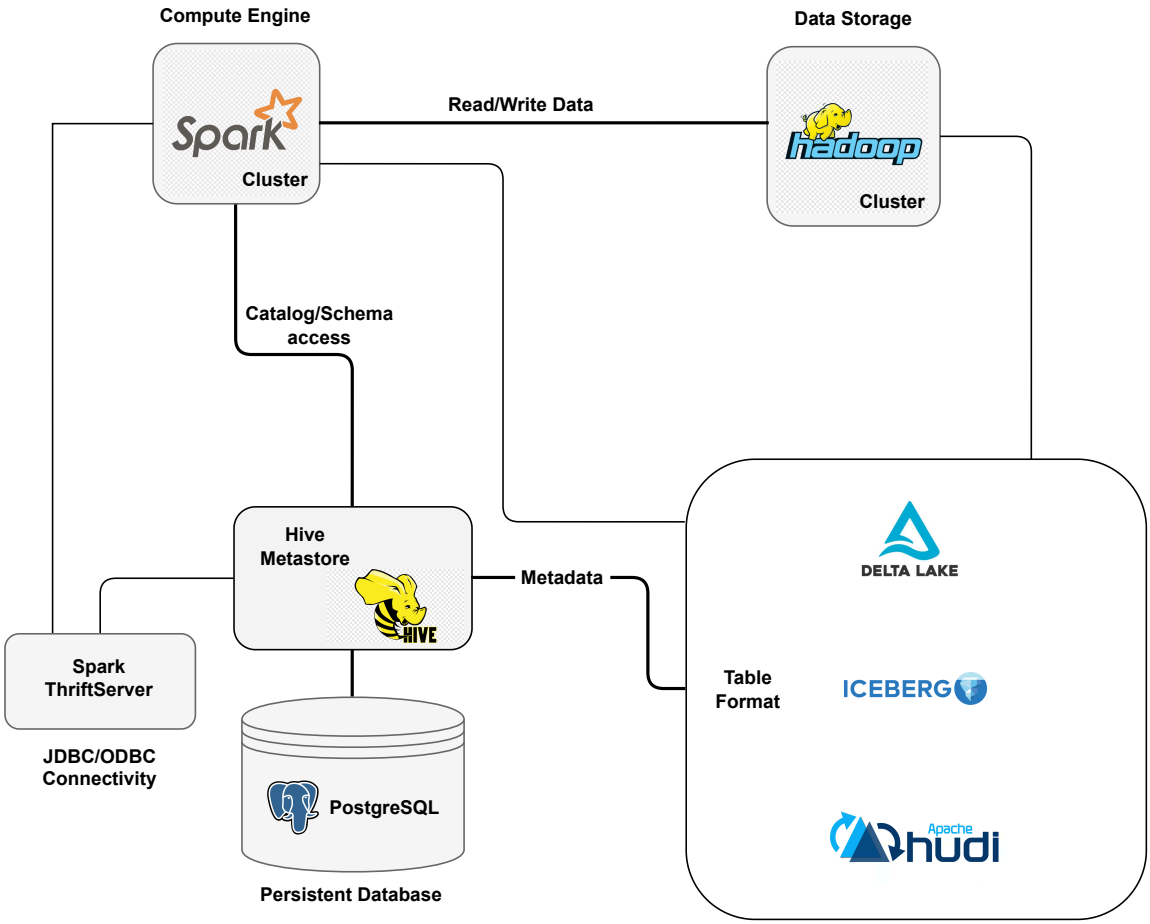

Figure 6.1. *System architecture overview*

Package	Version
openjdk-8-jdk	8u452
openjdk-11-jre	11.0.27
maven	3.6.3
libguava-java	19.0
libslf4j-java	1.7.25
libpostgresql-jdbc-java	42.2.10
postgresql-client-12	12.22
python3	3.8.2
python3-pip	20.0.2
python3-requests	2.22.0
python3-yaml	5.3.1

Table 6.1. *Core system packages relevant to the experimental environment*

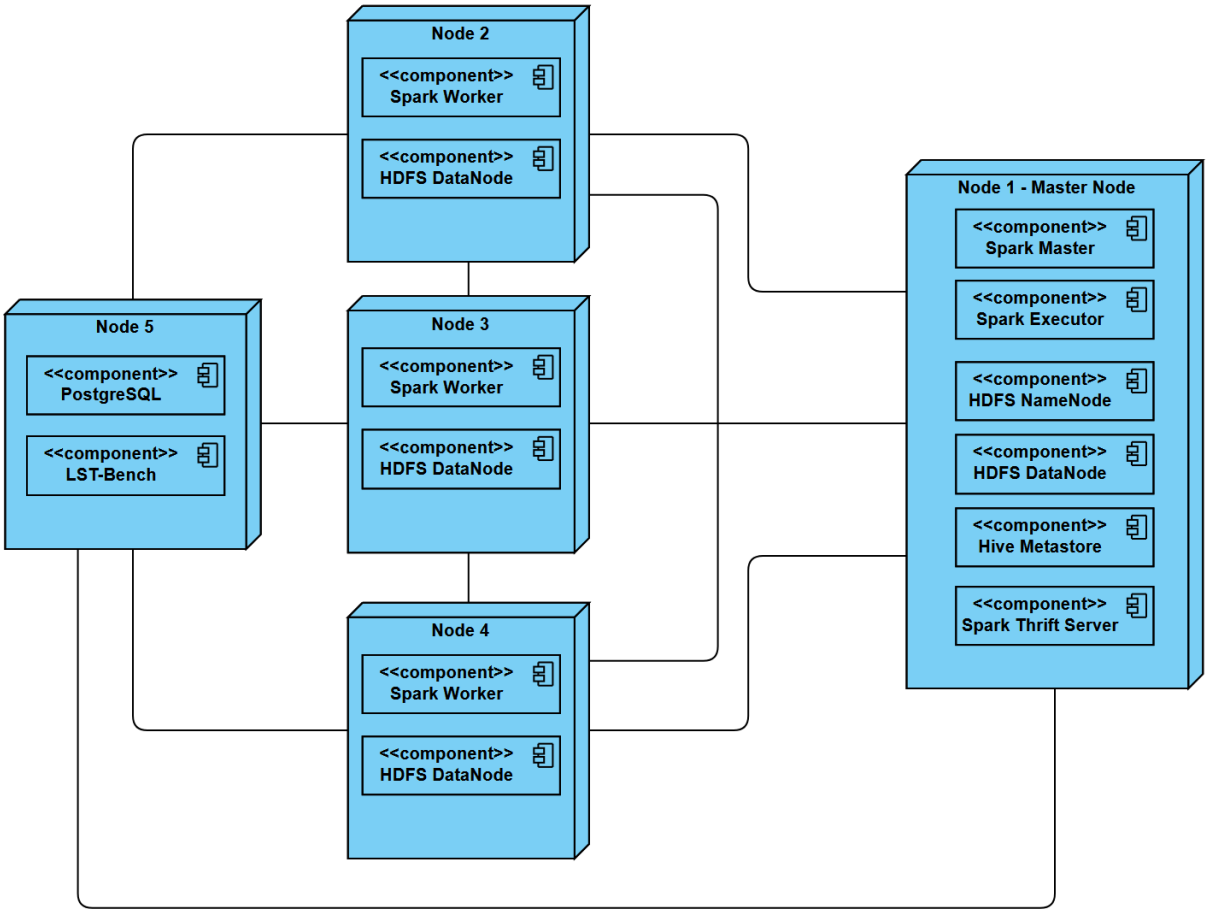

Figure 6.2. *Deployemnt diagram of the system*

6.3.1 Apache Spark

The distributed compute layer was powered by Apache Spark[16], operating in standalone cluster mode. The experiments were conducted using Apache Spark version 3.3.2, selected based on the compatibility guidelines provided by the LST-Benchmark[3] and alignment with the version requirements of the integrated systems, including Delta Lake, Apache Hudi, and Apache Iceberg.

Only four of the five total VMs were part of the Spark cluster. Each of these nodes functioned as a Spark worker, with one node additionally serving as the cluster master. Spark was tuned to strike a balance between execution throughput and resource utilization across benchmarks. The configurations specified in the `spark-defaults.conf` file, presented on Listing 6.2 were not arbitrarily chosen but rather the result of extensive iterative tuning on the 100GB scale factor dataset. Through numerous preliminary tests, these specific settings were chosen as those that significantly reduced the latency of the benchmark workloads, often by almost an order of magnitude, compared to default or other experimental configurations.

```
spark.master                        spark://10.0.1.31:7077
spark.eventLog.enabled              true
spark.eventLog.dir                  hdfs://10.0.1.31:9000/spark-logs
```

```
spark.hadoop.fs.defaultFS               hdfs://10.0.1.31:9000/
spark.hadoop.hive.metastore.uris        thrift://10.0.1.31:9083
spark.sql.catalogImplementation         hive
spark.sql.hive.metastore.version        3.1.3
spark.sql.hive.metastore.jars           path
spark.sql.hive.metastore.jars.path
    file:///home/ubuntu/spark-hdfs-metastore/apache-hive-3.1.3-bin/lib/*,
    file:///home/ubuntu/spark-hdfs-metastore/spark-3.3.2-bin-hadoop2/jars/*
spark.sql.warehouse.dir
    hdfs://10.0.1.31:9000/user/hive/warehouse/

spark.serializer
    org.apache.spark.serializer.KryoSerializer
spark.kryoserializer.buffer             1024k
spark.kryoserializer.buffer.max         1024m
spark.executor.instances           9
spark.executor.cores               5
spark.executor.memory              4g
spark.executor.memoryOverhead      400m

spark.driver.cores                 4
spark.driver.memory                4g
spark.driver.memoryOverhead        400m

spark.default.parallelism          45
spark.sql.shuffle.partitions       45
```

Listing 6.2: Spark's configuration file spark-defaults.conf

Some additional configurations for tuning the Spark cluster based on the system's hardware are shown on Listing 6.3.

```
# - SPARK_WORKER_CORES, to set the number of cores to use on this machine
SPARK_WORKER_CORES=12

# - SPARK_WORKER_MEMORY, to set how much total memory workers have to give
    executors (e.g. 1000m, 2g)
SPARK_WORKER_MEMORY=11g
```

Listing 6.3. *Spark's configuration file spark-env.sh*

6.3.2 Hadoop Distributed File System (HDFS)

The storage layer of the experimental cluster is powered by HDFS[17], which provides scalable and fault-tolerant access to data. HDFS was selected to emulate a realistic big data environment commonly seen in cloud-native architectures. Apache Hadoop version 2.7.1[18] was used. While this version is relatively dated, it was chosen deliberately due to compatibility constraints with the broader system stack.

The cluster consists of four virtual machines acting as HDFS DataNodes, while one node also serves as the NameNode, responsible for managing the filesystem metadata and block information. Key configuration details of the HDFS setup are as follows:

- **HDFS Version**: 2.7.1
- **Cluster Mode**: Multi-node deployment with 1 NameNode and 4 DataNodes
- **Replication Factor**: 1 (no data redundancy; used to minimize storage overhead during experiments)
- **Block Size**: 128 MB (default value)
- **Default Filesystem URI**: `hdfs://10.0.1.31:9000/`

While a replication factor of 1 means there is no fault tolerance at the storage layer, this configuration reduces disk usage and network overhead, allowing for faster iteration during experimental benchmarking.

Moreover, some environment variables defined in `hadoop-env.sh` and applied across the cluster were the following:

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export HADOOP_HEAPSIZE=2048

export HADOOP_NAMENODE_OPTS=
"-Dhadoop.security.logger=
 ${HADOOP_SECURITY_LOGGER:-INFO,RFAS}
-Dhdfs.audit.logger=
 ${HDFS_AUDIT_LOGGER:-INFO,NullAppender}
-Xmx3g -Xms2g
-XX:MetaspaceSize=2g
-XX:MaxMetaspaceSize=2g $HADOOP_NAMENODE_OPTS"
```

All nodes share the same configuration to ensure consistency in data access and I/O behavior across Spark jobs.

6.3.3 Apache Hive Metastore

The Apache Hive Metastore[19] serves as the central metadata catalog for all three table formats—Delta Lake, Apache Hudi, and Apache Iceberg. It facilitates schema management, table discovery, and partition tracking, providing a shared metadata layer that enables seamless interoperability across processing engines. Hive is deployed in metastore service mode, backed by a PostgreSQL[20] database for metadata persistence. The service is exposed to Spark via Thrift protocol, enabling high-performance remote access.

- **Hive Version**: 3.1.3
- **Metastore Backend**: PostgreSQL
- **Metastore URI**: `thrift://10.0.1.31:9083`
- **Warehouse Location**: `hdfs://10.0.1.31:9000/user/hive/warehouse/`

- **Spark Integration**: Configured via `spark.sql.catalogImplementation=hive`

The Hive Metastore connects to PostgreSQL via JDBC. The following properties are defined in hive-site.xml

```
<property>
 <name>javax.jdo.option.ConnectionURL</name>
 <value>jdbc:postgresql://10.0.1.35:5432/hive</value>
</property>
<property>
 <name>javax.jdo.option.ConnectionDriverName</name>
 <value>org.postgresql.Driver</value>
</property>
<property>
 <name>javax.jdo.option.ConnectionUserName</name>
 <value>hiveuser</value>
</property>
<property>
 <name>javax.jdo.option.ConnectionPassword</name>
 <value>hivepassword</value>
</property>
```

By using Hive as a unified catalog, the cluster ensures consistent schema enforcement and cross-engine metadata visibility. This setup plays a critical role in maintaining reproducibility and coherence across experiments.

6.3.4 PostgreSQL

PostgreSQL is employed as the backend database for the Hive Metastore. It provides reliable and transactional metadata storage, while supporting JDBC. PostgreSQL is installed on node 5, using the system package manager, as follows:

```
sudo apt update
sudo apt install postgresql
```

Once installed, the PostgreSQL service is enabled and started:

```
sudo systemctl enable postgresql
sudo systemctl start postgresql
```

Modified some configurations in order to allow connections from external nodes. After editing, the PostgreSQL service has to be restarted in order for the changes to be applied.

```
sudo vim /etc/postgresql/12/main/postgresql.conf
# edit below line, save and exit
listen_addresses = '*'
```

```
sudo vim /etc/postgresql/12/main/pg_hba.conf
# contents of the file are as follow
# Database administrative login by Unix domain socket
local   all             postgres                                peer

# Local connections via Unix socket
local   all             all                                     peer

# Remote access for all users from any IP
host    all             all             0.0.0.0/0               md5

# IPv4 loopback
host    all             all             127.0.0.1/32            md5

# IPv6 loopback
host    all             all             ::1/128                 md5

# Replication access
local   replication     all                                     peer
host    replication     all             127.0.0.1/32            md5
host    replication     all             ::1/128                 md5
```

The line `host all all 0.0.0.0/0 md5` is permissive and suitable for controlled benchmarking environments. In production, this should be restricted to specific IP ranges.

Moreover, PostgreSQL jar file postgresql-42.7.3.jar has to be placed in the Spark's jars directory. We created the appropriate symlink for this purpose. Then, we create a dedicated database, followed by the creation of a user hiveuser with a password hivepassword, who was granted full privileges on the database.

ollowing initialization, the HDFS daemons are started using the standard startup script:

6.4 System Initialization and Execution Sequence

The system initialization process begins by verifying that the PostgreSQL service is active and functioning correctly. This ensures that the metadata backend is available for subsequent components. Once confirmed, the Hive Metastore is launched as a background process to facilitate metadata management:

```
$ nohup hive --service metastore &
```

The HDFS must be initialized prior to use. This involves formatting the NameNode, which is required only once or when resetting the cluster:

```
$ hdfs namenode -format
```

Following initialization, the HDFS daemons are started using the standard startup script:

```
$ start-dfs.sh
```

This command launches the NameNode, SecondaryNameNode, and all DataNodes. The status of these processes can be verified using the Java Process Status tool:

```
$ jps
```

The final step involves starting the Apache Spark cluster and launching the Spark Thrift Server. Spark Thrift Server acts as a bridge between Spark SQL and external applications-here LST-Benchmark-offering a JDBC-compliant interface that facilitates SQL-based interactions with Spark's distributed processing engine.

Both the cluster and the Thrift Server are started via command-line invocations, which are dynamically configured based on the selected table format for execution. This dynamic configuration ensures that Spark is equipped with the necessary runtime components to interpret and manage the chosen format effectively.

However, the process of reaching this final stage was not without its challenges. Finding the correct configuration parameters and the necessary JAR (Java Archive) files for each of the LST formats—Delta Lake, Apache Iceberg, and Apache Hudi—proved to be a particularly demanding and time-consuming process. Significant difficulties were encountered, primarily due to version conflicts between various libraries and their dependencies, both within the Spark environment and with HDFS. Consequently, considerable time was dedicated to researching, testing, and resolving these technical issues in order for each format to operate.

6.4.1 Delta Lake

To enable support for Delta Lake, the Spark cluster must be initialized with the appropriate catalog and session extensions. This configuration ensures that Spark can interpret Delta tables and manage transactional consistency. The required JAR file, containing the Delta Lake runtime, must be explicitly included in the Spark classpath.

The following command demonstrates how the cluster is started with Delta Lake support:

```
$ ./start-all.sh
--conf "spark.sql.catalog.spark_catalog=
    org.apache.spark.sql.delta.catalog.DeltaCatalog"
--conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension"
--jars "~/spark-3.3.2-bin-hadoop2/jars/delta-core_2.12-2.2.0.jar"
```

Listing 6.4: Spark Cluster with Delta Lake support

This configuration enables the use of Delta Lake features such as ACID transactions, schema enforcement, and time travel within Spark SQL queries. The catalog is registered under the name `spark_catalog`, and the session extension activates Delta-specific optimizations during query planning and execution.

To launch the Thrift Server with Delta Lake support and optimized resource allocation, the following command can be used:

```
$ ./start-thriftserver.sh
```

```
--conf "spark.executor.instances=9"
--conf "spark.executor.cores=5"
--conf "spark.executor.memory=4g"
--conf "spark.executor.memoryOverhead=400m"
--conf "spark.driver.cores=4"
--conf "spark.driver.memory=4g"
--conf "spark.driver.memoryOverhead=400m"
--conf "spark.default.parallelism=45"
--conf "spark.sql.shuffle.partitions=45"
--conf
    "spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
--conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension"
--jars "~/spark-3.3.2-bin-hadoop2/jars/delta-core_2.12-2.2.0.jar"
```

Listing 6.5: Spark Thrift Server with Delta Lake support

This configuration ensures that the Thrift Server is equipped to handle concurrent queries efficiently while maintaining compatibility with Delta Lake. The executor and driver settings are tuned to support high-throughput workloads, and the inclusion of Delta-specific extensions allows users to query Delta tables directly through JDBC.

6.4.2 Apache Hudi

Accordingly, to enable support for Apache Hudi, the Spark cluster must be initialized with the appropriate session extensions and catalog configuration. This setup allows Spark to recognize Hudi tables and leverage features such as incremental data ingestion, upserts, and efficient storage management.

The following command demonstrates how the cluster is started with Hudi support:

```
$ ./start-all.sh
--packages org.apache.hudi:hudi-spark3.3-bundle_2.12:0.12.2
--conf 'spark.serializer=org.apache.spark.serializer.KryoSerializer'
--conf
    'spark.sql.extensions=org.apache.spark.sql.hudi.HoodieSparkSessionExtension'
--conf
    'spark.sql.catalog.spark_catalog=org.apache.spark.sql.hudi.catalog.HoodieCatalog'
--conf 'spark.hadoop.spark.sql.legacy.parquet.nanosAsLong=false'
--conf 'spark.hadoop.spark.sql.parquet.binaryAsString=false'
--conf 'spark.hadoop.spark.sql.parquet.int96AsTimestamp=true'
--conf 'spark.hadoop.spark.sql.caseSensitive=false'
```

Listing 6.6: Spark Cluster with Apache Hudi support

To deploy the Spark Thrift Server with Apache Hudi support and optimized resource allocation, the following command can be used:

```
  $ ./start-thriftserver.sh
--conf "spark.executor.instances=9"
--conf "spark.executor.cores=5"
--conf "spark.executor.memory=4g"
--conf "spark.executor.memoryOverhead=400m"
--conf "spark.driver.cores=4"
```

```
  --conf "spark.driver.memory=4g"
  --conf "spark.driver.memoryOverhead=400m"
  --conf "spark.default.parallelism=45"
  --conf "spark.sql.shuffle.partitions=45"
  --packages org.apache.hudi:hudi-spark3.3-bundle_2.12:0.12.2
  --conf 'spark.serializer=org.apache.spark.serializer.KryoSerializer'
  --conf
      'spark.sql.extensions=org.apache.spark.sql.hudi.HoodieSparkSessionExtension'
  --conf
      'spark.sql.catalog.spark_catalog=org.apache.spark.sql.hudi.catalog.HoodieCatalog'
  --conf 'spark.hadoop.spark.sql.legacy.parquet.nanosAsLong=false'
  --conf 'spark.hadoop.spark.sql.parquet.binaryAsString=false'
  --conf 'spark.hadoop.spark.sql.parquet.int96AsTimestamp=true'
  --conf 'spark.hadoop.spark.sql.caseSensitive=false'
```

Listing 6.7: Spark Thrift Server with Apache Hudi support

6.4.3 Apache Iceberg

To enable support for Apache Iceberg, the Spark cluster must be configured with the necessary session extensions and catalog definitions. This integration allows Spark to interact seamlessly with Iceberg tables, unlocking capabilities such as time travel queries, schema evolution, and efficient metadata management. The following command illustrates how the cluster is started with Iceberg support:

```
$ ./start-all.sh
--conf
    "spark.sql.catalog.spark_catalog=org.apache.iceberg.spark.SparkSessionCatalog"
--conf
    "spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions"
--jars "~spark-3.3.2-bin-hadoop2/jars/iceberg-spark-runtime-3.3_2.12-1.1.0.jar
```

Listing 6.8: Spark Cluster with Apache Iceberg support

To start the Spark Thrift Server with Apache Iceberg support and optimized resources, the following command can be used:

```
  $ ./start-thriftserver.sh
  --conf "spark.executor.instances=9"
  --conf "spark.executor.cores=5"
  --conf "spark.executor.memory=4g"
  --conf "spark.executor.memoryOverhead=400m"
  --conf "spark.driver.cores=4"
  --conf "spark.driver.memory=4g"
  --conf "spark.driver.memoryOverhead=400m"
  --conf "spark.default.parallelism=45"
  --conf "spark.sql.shuffle.partitions=45"
  --conf
      "spark.sql.catalog.spark_catalog=org.apache.iceberg.spark.SparkSessionCatalog"
  --conf
      "spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions"
```

```
    --jars
    "~/spark-3.3.2-bin-hadoop2/jars/iceberg-spark-runtime-3.3_2.12-1.1.0.jar"
```

Listing 6.9: Spark Thrift Server with Apache Iceberg support

6.5 LST-Benchmark Setup and Run Procedure

With the system components fully initialized and operational, the next phase involves configuring and executing the LST-Bench framework.

The connectivity between the benchmark driver and Apache Spark, is established through Spark Thrift Server. The required JDBC connection parameters are specified in a configuration file named `connections-config.yaml`. An excerpt of the configuration is shown below:

```
version: 1
connections:
  - id: spark_0
    driver: org.apache.hive.jdbc.HiveDriver
    url: jdbc:hive2://etsertou-1:10000
    username: hive
    password: hive
```

Listing 6.10: Connection configuration for LST-Bench

This configuration specifies a single connection identified by `spark_0`, which uses the Hive JDBC driver to connect to the Spark Thrift Server running on host `etsertou-1` at port `10000`. The benchmark framework uses these credentials to submit SQL queries and collect performance metrics during execution.

Below is the library configuration file `library-config.yaml`required for the execution of LST-Bench. This file remains unchanged from the official version provided in the LST-Bench documentation.

```
# Description: Library
---
version: 1
task_templates:
# Create external tables needed for benchmark
- id: setup
  files:
  - run/spark-3.3.1/scripts/tpcds/setup/ddl-external-tables.sql
# Create data maintenance external tables needed for benchmark
- id: setup_data_maintenance
  files:
  - run/spark-3.3.1/scripts/tpcds/setup_data_maintenance/ddl-external-tables-refresh.sql
  parameter_values_file: run/auxiliary/tpcds/setup_data_maintenance/parameter_values.dat
# Create schema and drop existing tables
- id: init
  files:
  - run/spark-3.3.1/scripts/tpcds/init/init.sql
# Create benchmark tables and load data into them
- id: build
  files:
  - run/spark-3.3.1/scripts/tpcds/build/1_create_call_center.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_catalog_page.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_catalog_returns.sql
```

```
  - run/spark-3.3.1/scripts/tpcds/build/1_create_catalog_sales.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_customer.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_customer_address.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_customer_demographics.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_date_dim.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_household_demographics.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_income_band.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_inventory.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_item.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_promotion.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_reason.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_ship_mode.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_store.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_store_returns.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_store_sales.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_time_dim.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_warehouse.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_web_page.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_web_returns.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_web_sales.sql
  - run/spark-3.3.1/scripts/tpcds/build/1_create_web_site.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_call_center.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_catalog_page.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_catalog_returns.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_catalog_sales.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_customer.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_customer_address.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_customer_demographics.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_date_dim.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_household_demographics.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_income_band.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_inventory.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_item.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_promotion.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_reason.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_ship_mode.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_store.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_store_returns.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_store_sales.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_time_dim.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_warehouse.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_web_page.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_web_returns.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_web_sales.sql
  - run/spark-3.3.1/scripts/tpcds/build/2_load_web_site.sql
# Compute statistics for tables
- id: analyze
  files:
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_call_center.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_catalog_page.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_catalog_returns.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_catalog_sales.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_customer.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_customer_address.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_customer_demographics.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_date_dim.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_household_demographics.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_income_band.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_inventory.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_item.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_promotion.sql
```

```
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_reason.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_ship_mode.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_store.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_store_returns.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_store_sales.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_time_dim.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_warehouse.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_web_page.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_web_returns.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_web_sales.sql
  - run/spark-3.3.1/scripts/tpcds/build/3_analyze_web_site.sql
# Execution of TPC-DS queries (possibly in a previous point-in-time)
- id: single_user
  files:
  - run/spark-3.3.1/scripts/tpcds/single_user/query1.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query2.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query3.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query4.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query5.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query6.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query7.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query8.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query9.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query10.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query11.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query12.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query13.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query14.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query15.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query16.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query17.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query18.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query19.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query20.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query21.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query22.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query23.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query24.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query25.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query26.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query27.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query28.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query29.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query30.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query31.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query32.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query33.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query34.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query35.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query36.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query37.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query38.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query39.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query40.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query41.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query42.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query43.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query44.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query45.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query46.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query47.sql
```

```
  - run/spark-3.3.1/scripts/tpcds/single_user/query48.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query49.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query50.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query51.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query52.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query53.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query54.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query55.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query56.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query57.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query58.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query59.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query60.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query61.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query62.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query63.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query64.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query65.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query66.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query67.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query68.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query69.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query70.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query71.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query72.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query73.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query74.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query75.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query76.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query77.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query78.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query79.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query80.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query81.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query82.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query83.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query84.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query85.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query86.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query87.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query88.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query89.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query90.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query91.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query92.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query93.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query94.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query95.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query96.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query97.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query98.sql
  - run/spark-3.3.1/scripts/tpcds/single_user/query99.sql
  permutation_orders_path: run/auxiliary/tpcds/single_user/permutation_orders/
  supports_time_travel: true
# Execution of TPC-DS data maintenance queries (Delta)
- id: data_maintenance_delta
  files:
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/DF_CS-merge.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/DF_I-merge.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/DF_SS-merge.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/DF_WS-merge.sql
```

```
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_CR.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_CS.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_I.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_SR.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_SS.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_WR.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_WS.sql
  parameter_values_file: run/auxiliary/tpcds/data_maintenance/parameter_values.dat
# Execution of TPC-DS data maintenance queries (Iceberg)
- id: data_maintenance_iceberg
  files:
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/DF_CS.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/DF_I.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/DF_SS.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/DF_WS.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_CR.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_CS.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_I.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_SR.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_SS.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_WR.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_WS.sql
  parameter_values_file: run/auxiliary/tpcds/data_maintenance/parameter_values.dat
# Execution of TPC-DS data maintenance queries (Hudi)
- id: data_maintenance_hudi
  files:
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/DF_CS-mixed.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/DF_I-mixed.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/DF_SS-mixed.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/DF_WS-mixed.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_CR.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_CS.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_I.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_SR.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_SS.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_WR.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance/LF_WS.sql
  parameter_values_file: run/auxiliary/tpcds/data_maintenance/parameter_values.dat
# Execution of optimize on all benchmark tables (Delta)
- id: optimize_delta
  files:
  - run/spark-3.3.1/scripts/tpcds/optimize/o_call_center-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_catalog_page-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_catalog_returns-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_catalog_sales-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer_address-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer_demographics-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_date_dim-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_household_demographics-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_income_band-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_inventory-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_item-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_promotion-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_reason-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_ship_mode-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_store-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_store_returns-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_store_sales-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_time_dim-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_warehouse-delta.sql
```

```
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_page-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_returns-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_sales-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_site-delta.sql
# Execution of optimize on all benchmark tables but splitting optimization
# of partitioned tables into batches by relying on dependent task executor (Delta)
- id: optimize_split_delta
  custom_task_executor: com.microsoft.lst_bench.task.custom.DependentTaskExecutor
  files:
  - run/spark-3.3.1/scripts/tpcds/optimize/o_call_center-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_catalog_page-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_returns_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_returns_IN-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_returns_NULL-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_sales_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_sales_IN-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_sales_NULL-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer_address-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer_demographics-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_date_dim-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_household_demographics-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_income_band-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_inventory_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_inventory_IN-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_inventory_NULL-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_item-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_promotion-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_reason-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_ship_mode-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_store-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_returns_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_returns_IN-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_returns_NULL-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_sales_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_sales_IN-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_sales_NULL-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_time_dim-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_warehouse-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_page-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_returns_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_returns_IN-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_returns_NULL-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_sales_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_sales_IN-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_sales_NULL-delta.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_site-delta.sql
# Execution of optimize on all benchmark tables (Iceberg)
- id: optimize_iceberg
  files:
  - run/spark-3.3.1/scripts/tpcds/optimize/o_call_center-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_catalog_page-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_catalog_returns-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_catalog_sales-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer_address-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer_demographics-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_date_dim-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_household_demographics-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_income_band-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_inventory-iceberg.sql
```

```
  - run/spark-3.3.1/scripts/tpcds/optimize/o_item-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_promotion-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_reason-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_ship_mode-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_store-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_store_returns-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_store_sales-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_time_dim-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_warehouse-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_page-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_returns-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_sales-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_site-iceberg.sql
# Execution of optimize on all benchmark tables but splitting optimization
# of partitioned tables into batches by relying on dependent task executor (Iceberg)
- id: optimize_split_iceberg
  custom_task_executor: com.microsoft.lst_bench.task.custom.DependentTaskExecutor
  files:
  - run/spark-3.3.1/scripts/tpcds/optimize/o_call_center-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_catalog_page-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_returns_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_returns_IN-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_returns_NULL-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_sales_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_sales_IN-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_sales_NULL-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer_address-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer_demographics-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_date_dim-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_household_demographics-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_income_band-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_inventory_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_inventory_IN-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_inventory_NULL-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_item-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_promotion-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_reason-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_ship_mode-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_store-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_returns_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_returns_IN-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_returns_NULL-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_sales_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_sales_IN-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_sales_NULL-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_time_dim-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_warehouse-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_page-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_returns_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_returns_IN-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_returns_NULL-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_sales_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_sales_IN-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_sales_NULL-iceberg.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_site-iceberg.sql
# Execution of optimize on all benchmark tables (Hudi)
- id: optimize_hudi
  files:
  - run/spark-3.3.1/scripts/tpcds/optimize/o_call_center-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_catalog_page-hudi.sql
```

```
  - run/spark-3.3.1/scripts/tpcds/optimize/o_catalog_returns-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_catalog_sales-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer_address-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer_demographics-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_date_dim-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_household_demographics-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_income_band-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_inventory-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_item-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_promotion-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_reason-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_ship_mode-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_store-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_store_returns-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_store_sales-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_time_dim-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_warehouse-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_page-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_returns-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_sales-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_site-hudi.sql
# Execution of optimize on all benchmark tables but splitting optimization
# of partitioned tables into batches by relying on dependent task executor (Hudi)
- id: optimize_split_hudi
  custom_task_executor: com.microsoft.lst_bench.task.custom.DependentTaskExecutor
  files:
  - run/spark-3.3.1/scripts/tpcds/optimize/o_call_center-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_catalog_page-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_returns_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_returns_IN-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_returns_NULL-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_sales_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_sales_IN-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_catalog_sales_NULL-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer_address-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_customer_demographics-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_date_dim-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_household_demographics-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_income_band-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_inventory_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_inventory_IN-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_inventory_NULL-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_item-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_promotion-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_reason-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_ship_mode-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_store-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_returns_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_returns_IN-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_returns_NULL-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_sales_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_sales_IN-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_store_sales_NULL-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_time_dim-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_warehouse-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_page-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_returns_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_returns_IN-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_returns_NULL-hudi.sql
```

```
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_sales_SELECT.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_sales_IN-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize_split/o_web_sales_NULL-hudi.sql
  - run/spark-3.3.1/scripts/tpcds/optimize/o_web_site-hudi.sql
# Execution of dependent TPC-DS data maintenance queries
- id: data_maintenance_dependent
  custom_task_executor: com.microsoft.lst_bench.task.custom.DependentTaskExecutor
  files:
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_CR_1.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_CR_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_CS_1.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_CS_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_CR_2.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_CR_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_CS_2.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_CS_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_CR_3.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_CR_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_CS_3.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_CS_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_I_1.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_I_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_I_2.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_I_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_I_3.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_I_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_SR_1.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_SR_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_SS_1.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_SS_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_SR_2.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_SR_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_SS_2.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_SS_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_SR_3.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_SR_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_SS_3.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_SS_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_WR_1.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_WR_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_WS_1.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_WS_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_WR_2.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_WR_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_WS_2.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_WS_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_WR_3.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_WR_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_WS_3.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/DF_WS_delete.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_CR_1.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_CR_2.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_CR_3.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_CR_insert.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_CS_1.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_CS_2.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_CS_3.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_CS_insert.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_I_1.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_I_2.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_I_3.sql
```

```
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_I_insert.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_SR_1.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_SR_2.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_SR_3.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_SR_insert.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_SS_1.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_SS_2.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_SS_3.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_SS_insert.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_WR_1.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_WR_2.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_WR_3.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_WR_insert.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_WS_1.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_WS_2.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_WS_3.sql
  - run/spark-3.3.1/scripts/tpcds/data_maintenance_dependent/LF_WS_insert.sql
  parameter_values_file: run/auxiliary/tpcds/data_maintenance/parameter_values.dat
```

Listing 6.11: Library configuration file for LST-Bench

LST-Bench collects detailed performance statistics across multiple levels, including experiment, phase, session, task, file, and individual SQL statements. Each event is recorded with a unique identifier—such as the statement name or phase ID as defined in the experiment configuration—alongside its execution status (success or failure) and any relevant metadata. By default, all logged events are stored in a DuckDB database for subsequent analysis. Therefore, a `telemetry-config.yaml` configuration file is also required. It establishes the DuckDB connection and specifies the paths for DDL and insert scripts necessary for telemetry logging.

```
version: 1
connection:
  id: duckdb_0
  driver: org.duckdb.DuckDBDriver
  url: jdbc:duckdb:./my-telemetry.duckdb
execute_ddl: true
ddl_file: 'src/main/resources/scripts/logging/duckdb/ddl.sql'
insert_file: 'src/main/resources/scripts/logging/duckdb/insert.sql'
parameter_values:
data_path: ''
```

Listing 6.12: Telemetry configuration for LST-Bench

A total of five experiment configuration templates were defined. One configuration template targets Delta Lake, which exclusively supports the Copy-on-Write (CoW) approach. The remaining four templates are equally divided for Apache Iceberg and Apache Hudi, both of which support dual table formats: Copy-on-Write and Merge-on-Read (MoR).

The five template configurations used in the experiments are presented below.

```
  version: 1
  id:
  repetitions: 3
  metadata:
    system: spark
    system_version: 3.3.2
    table_format: delta
```

```
  table_format_version: 2.2.0
  mode: cow
  scale_factor: "1000"
  cluster_size: "4"
  machine: cslab.16xIntelSkylake
parameter_values:
  external_catalog: spark_catalog
  external_database: "external_tpcds_sf_1000"
  external_table_format: csv
  external_data_path:
      'hdfs://etsertou-1:9000/user/hive/warehouse/tpcds_sf_1000/'
  external_options_suffix: ',header="false",nullValue="",delimiter="|"'
  external_tblproperties_suffix: ''
  catalog: spark_catalog
  database: "delta_cow"
  table_format: delta
  data_path: 'hdfs://etsertou-1:9000/user/hive/warehouse/delta/sf_1000/'
  options_suffix: ''
  tblproperties_suffix: ''
```

Listing 6.13: Experiment configuration for Delta Lake (CoW)

```
version: 1
id:
repetitions: 3
metadata:
  system: spark
  system_version: 3.3.2
  table_format: hudi
  table_format_version: 0.12.2
  mode: cow
  scale_factor: "1000"
  cluster_size: "4"
  machine: cslab.16xIntelSkylake
parameter_values:
  external_catalog: spark_catalog
  external_database: "external_tpcds_sf_1000"
  external_table_format: csv
  external_data_path:
      'hdfs://etsertou-1:9000/user/hive/warehouse/tpcds_sf_1000/'
  external_options_suffix: ',header="false",nullValue="",delimiter="|"'
  external_tblproperties_suffix: ''
  catalog: spark_catalog
  database: "cow_hudi"
  table_format: hudi
  data_path: 'hdfs://etsertou-1:9000/user/hive/warehouse/hudi/sf_1000/'
  options_suffix: ''
  tblproperties_suffix: ', "type"="cow"'
```

Listing 6.14: Experiment configuration for Apache Hudi (CoW)

```
version: 1
id:
```

```
repetitions: 3
metadata:
  system: spark
  system_version: 3.3.2
  table_format: hudi
  table_format_version: 0.12.2
  mode: mor
  scale_factor: "1000"
  cluster_size: "4"
  machine: cslab.16xIntelSkylake
parameter_values:
  external_catalog: spark_catalog
  external_database: "external_tpcds_sf_1000"
  external_table_format: csv
  external_data_path:
      'hdfs://etsertou-1:9000/user/hive/warehouse/tpcds_sf_1000/'
  external_options_suffix: ',header="false",nullValue="",delimiter="|"'
  external_tblproperties_suffix: ''
  catalog: spark_catalog
  database: "mor_hudi"
  table_format: hudi
  data_path: 'hdfs://etsertou-1:9000/user/hive/warehouse/hudi/sf_1000/'
  options_suffix: ''
  tblproperties_suffix: ', "type"="mor"'
```

Listing 6.15: Experiment configuration for Apache Hudi (MoR)

```
version: 1
id:
repetitions: 3
metadata:
  system: spark
  system_version: 3.3.2
  table_format: iceberg
  table_format_version: 1.1.0
  mode: cow
  scale_factor: "1000"
  cluster_size: "4"
  machine: cslab.16xIntelSkylake
parameter_values:
  external_catalog: spark_catalog
  external_database: "external_tpcds_sf_1000"
  external_table_format: csv
  external_data_path:
      'hdfs://etsertou-1:9000/user/hive/warehouse/tpcds_sf_1000/'
  external_options_suffix: ',header="false",nullValue="",delimiter="|"'
  external_tblproperties_suffix: ''
  catalog: spark_catalog
  database: "cow_iceberg"
  table_format: iceberg
  data_path: 'hdfs://etsertou-1:9000/user/hive/warehouse/iceberg/sf_1000/'
  options_suffix: ''
  tblproperties_suffix: ', "format-version"="2",
```

```
      "write.delete.mode"="copy-on-write",
      "write.update.mode"="copy-on-write",
      "write.merge.mode"="copy-on-write"'
```

Listing 6.16: Experiment configuration for Apache Iceberg (Cow)

```
version: 1
id:
repetitions: 3
metadata:
  system: spark
  system_version: 3.3.2
  table_format: iceberg
  table_format_version: 1.1.0
  mode: mor
  scale_factor: "1000"
  cluster_size: "4"
  machine: cslab.16xIntelSkylake
parameter_values:
  external_catalog: spark_catalog
  external_database: "external_tpcds_sf_1000"
  external_table_format: csv
  external_data_path:
      'hdfs://etsertou-1:9000/user/hive/warehouse/tpcds_sf_1000/'
  external_options_suffix: ',header="false",nullValue="",delimiter="|"'
  external_tblproperties_suffix: ''
  catalog: spark_catalog
  database: "mor_iceberg"
  table_format: iceberg
  data_path: 'hdfs://etsertou-1:9000/user/hive/warehouse/iceberg/sf_1000/'
  options_suffix: ''
  tblproperties_suffix: ', "format-version"="2",
      "write.delete.mode"="merge-on-read",
      "write.update.mode"="merge-on-read",
      "write.merge.mode"="merge-on-read"'
```

Listing 6.17: Experiment configuration for Apache Iceberg (MoR)

Although the configurations presented above are defined with a scale factor of 1000, each of them has also been executed with a scale factor of 100 for validation and tuning purposes. The execution logic and structure remain identical across both scale factors.

Lastly, as mentioned in subsection 5.4.4, the four workloads corresponding to each table format are presented below.

```
# Description: W0: Original TPC-DS sequence
---
version: 1
id: w0_tpcds
phases:
- id: setup
  sessions:
  - tasks:
    - template_id: setup
```

```
- id: setup_data_maintenance
  sessions:
  - tasks:
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
- id: init
  sessions:
  - tasks:
    - template_id: init
- id: build
  sessions:
  - tasks:
    - template_id: build
- id: single_user
  sessions:
  - tasks:
    - template_id: single_user
- id: throughput_1
  sessions:
  - tasks:
    - template_id: single_user
      permute_order: true
  - tasks:
    - template_id: single_user
      permute_order: true
  - tasks:
    - template_id: single_user
      permute_order: true
  - tasks:
    - template_id: single_user
      permute_order: true
- id: data_maintenance_1
  sessions:
  - tasks:
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
- id: throughput_2
  sessions:
  - tasks:
    - template_id: single_user
      permute_order: true
  - tasks:
    - template_id: single_user
      permute_order: true
  - tasks:
    - template_id: single_user
      permute_order: true
  - tasks:
    - template_id: single_user
      permute_order: true
```

```
- id: data_maintenance_2
  sessions:
  - tasks:
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
```

Listing 6.18: W0: Original TPC-DS sequence, Delta Lake

```
# Description: W0: Original TPC-DS sequence
---
version: 1
id: w0_tpcds
phases:
- id: setup
  sessions:
  - tasks:
    - template_id: setup
- id: setup_data_maintenance
  sessions:
  - tasks:
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
- id: init
  sessions:
  - tasks:
    - template_id: init
- id: build
  sessions:
  - tasks:
    - template_id: build
    replace_regex:
    - pattern: '(?i)varchar\(.*\)|char\(.*\)'
       replacement: 'string'
- id: single_user
  sessions:
  - tasks:
    - template_id: single_user
- id: throughput_1
  sessions:
  - tasks:
    - template_id: single_user
    permute_order: true
  - tasks:
    - template_id: single_user
    permute_order: true
  - tasks:
    - template_id: single_user
    permute_order: true
  - tasks:
      - template_id: single_user
       permute_order: true
```

```
- id: data_maintenance_1
  sessions:
  - tasks:
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
- id: throughput_2
  sessions:
  - tasks:
    - template_id: single_user
    permute_order: true
  - tasks:
    - template_id: single_user
    permute_order: true
  - tasks:
    - template_id: single_user
      permute_order: true
  - tasks:
    - template_id: single_user
    permute_order: true
- id: data_maintenance_2
  sessions:
  - tasks:
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
```

Listing 6.19: W0: Original TPC-DS sequence, Apache Hudi

```
# Description: W0: Original TPC-DS sequence
---
version: 1
id: w0_tpcds
phases:
- id: setup
  sessions:
  - tasks:
    - template_id: setup
- id: setup_data_maintenance
  sessions:
  - tasks:
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
- id: init
  sessions:
  - tasks:
    - template_id: init
- id: build
  sessions:
  - tasks:
    - template_id: build
- id: single_user
  sessions:
```

```
  - tasks:
    - template_id: single_user
- id: throughput_1
  sessions:
  - tasks:
    - template_id: single_user
      permute_order: true
  - tasks:
    - template_id: single_user
      permute_order: true
  - tasks:
    - template_id: single_user
      permute_order: true
  - tasks:
    - template_id: single_user
      permute_order: true
- id: data_maintenance_1
  sessions:
  - tasks:
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
- id: throughput_2
  sessions:
  - tasks:
    - template_id: single_user
      permute_order: true
  - tasks:
    - template_id: single_user
      permute_order: true
  - tasks:
    - template_id: single_user
      permute_order: true
   - tasks:
    - template_id: single_user
      permute_order: true
- id: data_maintenance_2
  sessions:
  - tasks:
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
```

Listing 6.20: W0: Original TPC-DS sequence, Apache Iceberg

```
# Description: WP1: Longevity
---
version: 1
id: wp1_longevity
phases:
- id: setup
  sessions:
  - tasks:
    - template_id: setup
- id: setup_data_maintenance
```

```
    sessions:
    - tasks:
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
  - id: init
    sessions:
    - tasks:
      - template_id: init
  - id: build
    sessions:
    - tasks:
      - template_id: build
  - id: single_user_1
    sessions:
    - tasks:
      - template_id: single_user
  - id: data_maintenance_1
    sessions:
    - tasks:
      - template_id: data_maintenance_delta
      - template_id: data_maintenance_delta
  - id: single_user_2
    sessions:
    - tasks:
      - template_id: single_user
  - id: data_maintenance_2
    sessions:
    - tasks:
      - template_id: data_maintenance_delta
      - template_id: data_maintenance_delta
  - id: single_user_3
    sessions:
    - tasks:
      - template_id: single_user
  - id: data_maintenance_3
    sessions:
    - tasks:
      - template_id: data_maintenance_delta
      - template_id: data_maintenance_delta
  - id: single_user_4
    sessions:
    - tasks:
      - template_id: single_user
  - id: data_maintenance_4
```

```
  sessions:
  - tasks:
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
- id: single_user_5
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_5
  sessions:
  - tasks:
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
- id: single_user_6
  sessions:
  - tasks:
    - template_id: single_user
```

Listing 6.21: WP1: Longevity, Delta Lake

```
# Description: WP1: Longevity
---
version: 1
id: wp1_longevity
phases:
- id: setup
  sessions:
  - tasks:
    - template_id: setup
- id: setup_data_maintenance
  sessions:
  - tasks:
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
- id: init
  sessions:
  - tasks:
    - template_id: init
- id: build
  sessions:
  - tasks:
    - template_id: build
      replace_regex:
      - pattern: '(?i)varchar\(.*\)|char\(.*\)'
        replacement: 'string'
```

```
- id: single_user_1
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_1
  sessions:
  - tasks:
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
- id: single_user_2
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_2
  sessions:
  - tasks:
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
- id: single_user_3
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_3
  sessions:
  - tasks:
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
- id: single_user_4
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_4
  sessions:
  - tasks:
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
- id: single_user_5
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_5
  sessions:
  - tasks:
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
- id: single_user_6
  sessions:
  - tasks:
    - template_id: single_user
```

Listing 6.22: WP1: Longevity, Apache Hudi

```
# Description: WP1: Longevity
```

```
---
version: 1
id: wp1_longevity
phases:
- id: setup
  sessions:
  - tasks:
    - template_id: setup
- id: setup_data_maintenance
  sessions:
  - tasks:
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
- id: init
  sessions:
  - tasks:
    - template_id: init
- id: build
  sessions:
  - tasks:
    - template_id: build
- id: single_user_1
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_1
  sessions:
  - tasks:
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
- id: single_user_2
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_2
  sessions:
  - tasks:
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
- id: single_user_3
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_3
```

```
    sessions:
    - tasks:
      - template_id: data_maintenance_iceberg
      - template_id: data_maintenance_iceberg
  - id: single_user_4
    sessions:
    - tasks:
      - template_id: single_user
  - id: data_maintenance_4
    sessions:
    - tasks:
      - template_id: data_maintenance_iceberg
      - template_id: data_maintenance_iceberg
  - id: single_user_5
    sessions:
    - tasks:
      - template_id: single_user
  - id: data_maintenance_5
    sessions:
    - tasks:
      - template_id: data_maintenance_iceberg
      - template_id: data_maintenance_iceberg
  - id: single_user_6
    sessions:
    - tasks:
      - template_id: single_user
```

Listing 6.23: WP1: Longevity, Apache Iceberg

```
# Description: WP2: Resilience
---
version: 1
id: wp2_resilience
phases:
- id: setup_data_maintenance
  sessions:
  - tasks:
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
- id: init
  sessions:
  - tasks:
    - template_id: init
```

```
- id: build
  sessions:
  - tasks:
    - template_id: build
- id: single_user_1
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_1
  sessions:
  - tasks:
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
- id: single_user_2
  sessions:
  - tasks:
    - template_id: single_user
- id: optimize_1
  sessions:
  - tasks:
    - template_id: optimize_delta
- id: single_user_2o
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_2
  sessions:
  - tasks:
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
- id: single_user_3
  sessions:
  - tasks:
    - template_id: single_user
- id: optimize_2
  sessions:
  - tasks:
    - template_id: optimize_delta
- id: single_user_3o
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_3
  sessions:
  - tasks:
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
```

```
    - template_id: data_maintenance_delta
- id: single_user_4
  sessions:
  - tasks:
    - template_id: single_user
- id: optimize_3
  sessions:
  - tasks:
    - template_id: optimize_delta
- id: single_user_4o
  sessions:
  - tasks:
    - template_id: single_user
```

Listing 6.24: WP2: Resilience, Delta Lake

```
# Description: WP2: Resilience
---
version: 1
id: wp2_resilience
phases:
- id: setup_data_maintenance
  sessions:
  - tasks:
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
- id: init
  sessions:
  - tasks:
    - template_id: init
- id: build
  sessions:
  - tasks:
    - template_id: build
      replace_regex:
      - pattern: '(?i)varchar\(.*\)|char\(.*\)'
        replacement: 'string'
- id: single_user_1
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_1
  sessions:
```

```
  - tasks:
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
- id: single_user_2
  sessions:
  - tasks:
    - template_id: single_user
- id: optimize_1
  sessions:
  - tasks:
    - template_id: optimize_hudi
- id: single_user_2o
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_2
  sessions:
  - tasks:
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
- id: single_user_3
  sessions:
  - tasks:
    - template_id: single_user
- id: optimize_2
  sessions:
  - tasks:
    - template_id: optimize_hudi
- id: single_user_3o
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_3
  sessions:
  - tasks:
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
    - template_id: data_maintenance_hudi
- id: single_user_4
  sessions:
  - tasks:
    - template_id: single_user
- id: optimize_3
  sessions:
  - tasks:
    - template_id: optimize_hudi
- id: single_user_4o
```

```
    sessions:
    - tasks:
      - template_id: single_user
```

Listing 6.25: WP2: Resilience, Apache Hudi

```
# Description: WP2: Resilience
---
version: 1
id: wp2_resilience
phases:
- id: setup_data_maintenance
  sessions:
  - tasks:
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
- id: init
  sessions:
  - tasks:
    - template_id: init
- id: build
  sessions:
  - tasks:
    - template_id: build
- id: single_user_1
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_1
  sessions:
  - tasks:
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
- id: single_user_2
  sessions:
  - tasks:
    - template_id: single_user
- id: optimize_1
  sessions:
  - tasks:
    - template_id: optimize_iceberg
- id: single_user_2o
  sessions:
```

```
  - tasks:
    - template_id: single_user
- id: data_maintenance_2
  sessions:
  - tasks:
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
- id: single_user_3
  sessions:
  - tasks:
    - template_id: single_user
- id: optimize_2
  sessions:
  - tasks:
    - template_id: optimize_iceberg
- id: single_user_3o
  sessions:
  - tasks:
    - template_id: single_user
- id: data_maintenance_3
  sessions:
  - tasks:
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
- id: single_user_4
  sessions:
  - tasks:
    - template_id: single_user
- id: optimize_3
  sessions:
  - tasks:
    - template_id: optimize_iceberg
- id: single_user_4o
  sessions:
  - tasks:
    - template_id: single_user
```

Listing 6.26: WP2: Resilience, Apache Iceberg

```
# Description: WP3: R/W concurrency
---
version: 1
id: wp3_rw_concurrency
phases:
- id: setup_data_maintenance
  sessions:
  - tasks:
```

```
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
      - template_id: setup_data_maintenance
  - id: init
    sessions:
    - tasks:
      - template_id: init
  - id: build
    sessions:
    - tasks:
      - template_id: build
  - id: single_user_1_data_maintenance_1
    sessions:
    - tasks:
      - template_id: single_user
    - tasks:
      - template_id: data_maintenance_delta
      - template_id: data_maintenance_delta
  - id: single_user_2_optimize_1
    sessions:
    - tasks:
      - template_id: single_user
    - tasks:
      - template_id: optimize_delta
  - id: single_user_2o_data_maintenance_2
    sessions:
    - tasks:
      - template_id: single_user
    - tasks:
      - template_id: data_maintenance_delta
      - template_id: data_maintenance_delta
      - template_id: data_maintenance_delta
      - template_id: data_maintenance_delta
  - id: single_user_3_optimize_2
    sessions:
    - tasks:
      - template_id: single_user
    - tasks:
      - template_id: optimize_delta
  - id: single_user_3o_data_maintenance_3
    sessions:
    - tasks:
      - template_id: single_user
```

```
  - tasks:
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
- id: single_user_4_optimize_3
  sessions:
  - tasks:
    - template_id: single_user
  - tasks:
    - template_id: optimize_delta
```

Listing 6.27: WP3: R/W concurrency, Delta Lake

```
# Description: WP3: R/W concurrency
---
version: 1
id: wp3_rw_concurrency
phases:
- id: setup_data_maintenance
  sessions:
  - tasks:
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
- id: init
  sessions:
  - tasks:
    - template_id: init
- id: build
  sessions:
  - tasks:
    - template_id: build
      replace_regex:
    - pattern: '(?i)varchar\(.*\)|char\(.*\)'
      replacement: 'string'
- id: single_user_1_data_maintenance_1
  sessions:
  - tasks:
    - template_id: single_user
  - tasks:
    - template_id: data_maintenance_hudi
```

```
      - template_id: data_maintenance_hudi
  - id: single_user_2_optimize_1
    sessions:
    - tasks:
      - template_id: single_user
    - tasks:
      - template_id: optimize_hudi
  - id: single_user_2o_data_maintenance_2
    sessions:
    - tasks:
      - template_id: single_user
    - tasks:
      - template_id: data_maintenance_hudi
      - template_id: data_maintenance_hudi
      - template_id: data_maintenance_hudi
      - template_id: data_maintenance_hudi
  - id: single_user_3_optimize_2
    sessions:
    - tasks:
      - template_id: single_user
    - tasks:
      - template_id: optimize_hudi
  - id: single_user_3o_data_maintenance_3
    sessions:
    - tasks:
      - template_id: single_user
    - tasks:
      - template_id: data_maintenance_hudi
      - template_id: data_maintenance_hudi
      - template_id: data_maintenance_hudi
      - template_id: data_maintenance_hudi
      - template_id: data_maintenance_hudi
      - template_id: data_maintenance_hudi
  - id: single_user_4_optimize_3
    sessions:
    - tasks:
      - template_id: single_user
    - tasks:
      - template_id: optimize_hudi
```

Listing 6.28: WP3: R/W concurrency, Apache Hudi

```
# Description: WP3: R/W concurrency
---
version: 1
id: wp3_rw_concurrency
phases:
- id: setup_data_maintenance
  sessions:
  - tasks:
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
```

```
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
- id: init
  sessions:
  - tasks:
    - template_id: init
- id: build
  sessions:
  - tasks:
    - template_id: build
- id: single_user_1_data_maintenance_1
  sessions:
  - tasks:
    - template_id: single_user
  - tasks:
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
- id: single_user_2_optimize_1
  sessions:
  - tasks:
    - template_id: single_user
  - tasks:
    - template_id: optimize_iceberg
- id: single_user_2o_data_maintenance_2
  sessions:
  - tasks:
    - template_id: single_user
  - tasks:
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
- id: single_user_3_optimize_2
  sessions:
  - tasks:
    - template_id: single_user
  - tasks:
    - template_id: optimize_iceberg
- id: single_user_3o_data_maintenance_3
  sessions:
  - tasks:
    - template_id: single_user
  - tasks:
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
```

```
      - template_id: data_maintenance_iceberg
      - template_id: data_maintenance_iceberg
      - template_id: data_maintenance_iceberg
      - template_id: data_maintenance_iceberg
  - id: single_user_4_optimize_3
    sessions:
    - tasks:
      - template_id: single_user
    - tasks:
      - template_id: optimize_iceberg
```

Listing 6.29: WP3: R/W concurrency, Apache Iceberg

```
# Description: WP4: Time travel
---
version: 1
id: wp4_time_travel
phases:
- id: setup
  sessions:
  - tasks:
    - template_id: setup
- id: setup_data_maintenance
  sessions:
  - tasks:
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
- id: init
  sessions:
  - tasks:
    - template_id: init
- id: build
  sessions:
  - tasks:
    - template_id: build
- id: data_maintenance_1
  sessions:
  - tasks:
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
- id: single_user_2_0
  sessions:
  - tasks:
```

```
    - template_id: single_user
      time_travel_phase_id: build
- id: data_maintenance_2
  sessions:
  - tasks:
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
- id: single_user_3_1
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: data_maintenance_1
- id: single_user_3_0
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: build
- id: data_maintenance_3
  sessions:
  - tasks:
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
- id: single_user_4_2
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: data_maintenance_2
- id: single_user_4_1
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: data_maintenance_1
- id: single_user_4_0
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: build
- id: data_maintenance_4
  sessions:
  - tasks:
    - template_id: data_maintenance_delta
    - template_id: data_maintenance_delta
- id: single_user_5_3
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: data_maintenance_3
- id: single_user_5_2
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: data_maintenance_2
```

```
- id: single_user_5_1
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: data_maintenance_1
- id: single_user_5_0
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: build
```

Listing 6.30: WP4: Time travel, Delta Lake

```
# Description: WP4: Time travel
---
version: 1
id: wp4_time_travel
phases:
- id: setup
  sessions:
  - tasks:
    - template_id: setup
- id: setup_data_maintenance
  sessions:
  - tasks:
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
    - template_id: setup_data_maintenance
- id: init
  sessions:
  - tasks:
    - template_id: init
- id: build
  sessions:
  - tasks:
    - template_id: build
- id: data_maintenance_1
  sessions:
  - tasks:
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
- id: single_user_2_0
  sessions:
  - tasks:
```

```
    - template_id: single_user
      time_travel_phase_id: build
- id: data_maintenance_2
  sessions:
  - tasks:
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
- id: single_user_3_1
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: data_maintenance_1
- id: single_user_3_0
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: build
- id: data_maintenance_3
  sessions:
  - tasks:
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
- id: single_user_4_2
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: data_maintenance_2
- id: single_user_4_1
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: data_maintenance_1
- id: single_user_4_0
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: build
- id: data_maintenance_4
  sessions:
  - tasks:
    - template_id: data_maintenance_iceberg
    - template_id: data_maintenance_iceberg
- id: single_user_5_3
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: data_maintenance_3
- id: single_user_5_2
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: data_maintenance_2
```

```
- id: single_user_5_1
  sessions:
  - tasks:
    - template_id: single_user
      time_travel_phase_id: data_maintenance_1
- id: single_user_5_0
  sessions:
  - tasks:
    - template_id: single_user
```

Listing 6.31: WP4: Time travel, Apache Iceberg

Apache Hudi was excluded from the execution of the `wp4 time travel` workload due to a known limitation in its SQL extension. Specifically, time travel queries using the `TIMESTAMP AS OF` syntax result in an `AnalysisException`, preventing successful execution. This issue is documented under HUDI-7274 in the Apache Hudi issue tracker. As a result, the wp4 workload is not presented for Hudi in this study.

Chapter 7

Evaluation

In this chapter, we present the evaluation of the benchmarking results, obtained from the execution of the workloads described in Section 6.5, using the LST-Bench framework with Delta Lake, Apache Hudi and Apache Iceberg, all deployed on Apache Spark. System-level tuning was applied exclusively to the Spark execution engine, while the evaluated log-structured table formats were executed using their default configuration parameters. The purpose of this evaluation is to highlight the practical performance differences and scalability characteristics of each format under varying workload patterns.

The remainder of this chapter is structured around the individual evaluation of each workload pattern. As mentioned on the previous chapter, each workload was ran against 5 variations of the LSTs; both CoW and MoR configurations for Apache Iceberg and Apache Hudi, and only CoW for Delta Lake as it does not support MoR. For each workload, we present and analyze the performance results across the three table formats highlighting key differences in performance, scalability, and overall efficiency.

7.1 Longevity

The aim of the longevity workload, consisting of six Single User and five interleaved Data Maintenance phases (as illustrated on Figure 5.2), is to evaluate an LST's ability to maximize performance, efficiency, and stability metrics in scenarios that involve frequent data updates.

Upon initial inspection of Figure 7.1, a clear performance dichotomy emerges across the tested configurations. Delta Lake's CoW configuration consistently exhibits significantly higher latency, averaging approximately 11 minutes per single user phase. This contrasts sharply with the remaining four configurations, which form a tightly clustered group demonstrating substantially lower latencies, typically ranging between 3 and 5 minutes. The consistency of these performance profiles across all six single user runs underscores the inherent stability of these configurations.

The most striking observation is the exceptionally poor performance of Delta Lake's CoW configuration. Its latency is nearly three to four times greater than the next slowest configuration. Given that the base hardware and Spark configurations were identical across all tests, this rules out hardware limitations as the root cause. The elevated latency for this Delta Lake configuration suggests a critical performance bottleneck or inefficiency specifically inherent to this particular combination, and potentially exacerbated by the virtualized environment. Delta Lake

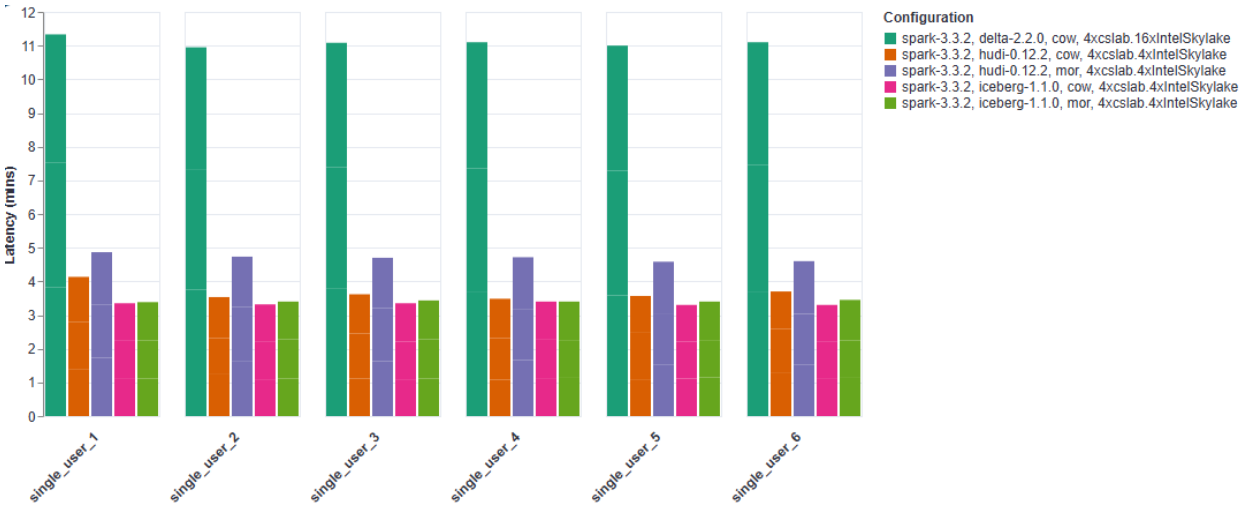

Figure 7.1. *Performance of WP-1 Single User phases (SF1000)*

2.2.0's CoW strategy may be I/O-intensive or may generate a high volume of small write/rewrite operations. These overheads could be disproportionately magnified compared to the other configurations. The virtualization layer might introduce additional latency for each I/O request, making highly chatty or rewrite-heavy operations significantly slower. It is also plausible that this particular Delta Lake version (2.2.0) or its CoW mechanism may interact unfavorably with the underlying HDFS architecture, potentially leading to suboptimal file system operations or metadata management overheads. Additionally, while base Spark configurations were consistent, specific Spark tuning parameters related to worker allocation, memory management, or task scheduling (e.g., executor cores, memory, parallelism settings) might not be optimally aligned for Delta Lake's specific write patterns on HDFS within this virtualized context, thus exacerbating its observed performance challenges compared to the other LSTs.

In contrast to the Delta Lake outlier, both Apache Hudi and Apache Iceberg configurations demonstrate highly competitive and significantly lower latencies.

Apache Hudi's CoW configuration consistently achieved query latencies lower than its MoR counterpart. This solid performance indicates that Hudi's CoW strategy, when used for data maintenance in version 0.12.2, effectively manages data, resulting in a well-organized dataset conducive to efficient query execution. This finding is particularly insightful for query performance. While MoR is often designed to optimize write/update performance, it might achieve this by maintaining a more complex file structure (base files + delta log files) that requires more effort during the read (query) phase to merge data on the fly.

Apache Iceberg's CoW configuration is among the top performers, consistently delivering faster query times than any Hudi configuration and closely rivaling Iceberg's MoR. Iceberg's MoR achieved the lowest query latencies across all tested scenarios, outperforming all other configurations in every single-user phase. This indicates that Iceberg's MoR implementation is exceptionally optimized to handle updates efficiently without significantly compromising read performance, suggesting superior file layout or metadata handling.

In summary, the results provide crucial insights into how the chosen update strategy (CoW vs. MoR) impacts subsequent query performance. For Apache Hudi 0.12.2, the CoW strategy

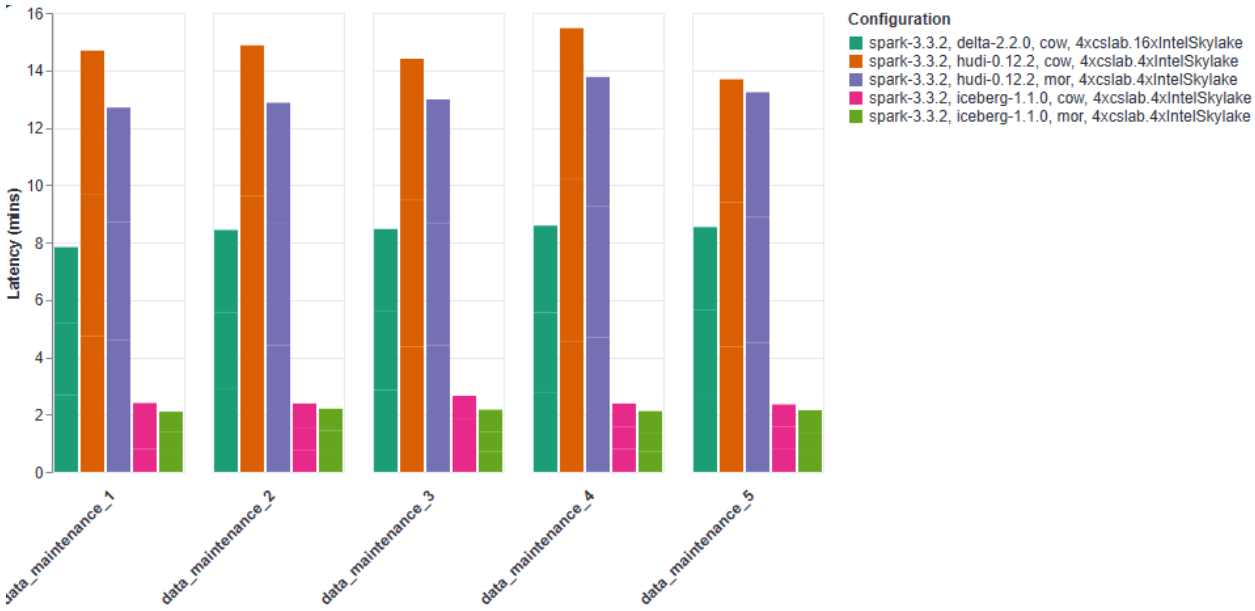

Figure 7.2. *Performance of WP-1 Data Maintenance phases (SF1000)*

led to significantly lower query latencies than its MoR strategy, suggesting that while MoR is intended to optimize updates, its file structure imposes a higher computational cost on the query engine. On the other hand, both Iceberg's CoW and MoR strategies result in excellent query performance. Iceberg's MoR strategy consistently delivered the absolute best query performance, with Iceberg's CoW being a very close second. This implies that Iceberg 1.1.0 has highly optimized implementations for both strategies, with MoR providing marginal but consistent additional benefits in query latency for this specific workload. Lastly, the severely underperforming Delta Lake CoW configuration for queries underscores that its update strategy, when repeatedly applied in a longevity test, results in a data state that is highly detrimental to query efficiency. This is a critical failure point for Delta Lake 2.2.0 under these specific conditions.

7.2 Resilience

Subsequently, we examined how the performance of LSTs is affected when maintenance operations are incorporated into the workload. Specifically, we employed the resilience workload, which assesses the impact of the Optimize phase, i.e., the compaction of small data files into larger ones to improve efficiency. This workload iteratively executes a sequence of Single User (labeled SU-i), Data Maintenance (with an increasing number of tasks), Optimize, and Single User (labeled SU-i-O) tasks. The analysis leverages three distinct charts, each focusing on a specific phase.

Figure 7.3 presents the latency incurred during the data maintenance phase, representing the time taken for data updates to be applied. Given that these operations include updates with an increasing number of tasks across iterations, an increase in latency in subsequent phases is a logical expectation. The critical metric for evaluation, therefore, becomes the rate and magnitude of this latency increase, indicating the scalability and efficiency of each format's update mechanism.

Delta Lake's CoW configuration exhibits a significant increase in latency across the data

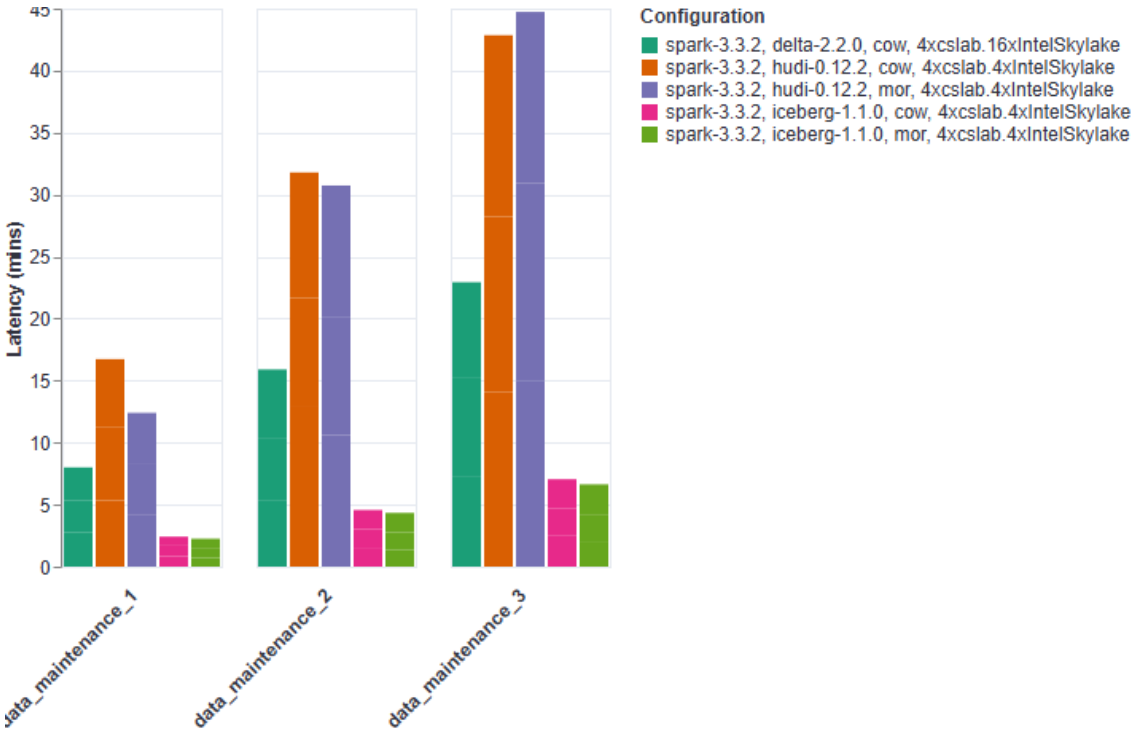

Figure 7.3. *Performance of WP-2 Data Maintenance (SF1000)*

maintenance iterations. While some increase is expected, this substantial escalation points to notable inefficiencies in Delta Lake 2.2.0's CoW update mechanism under a growing update load.

Hudi's both MoR and CoW strategies demonstrate high and rapidly escalating latencies during the data maintenance phase. This steep and substantial increase across both strategies strongly suggests that Hudi 0.12.2, regardless of CoW or MoR, struggles severely with the increasing update patterns of the resilience workload, indicating poor scalability for update-heavy operations in this specific setup. This finding is particularly critical for MoR, which often aims to reduce write-time overhead, indicating significant overheads even with deferred merging in this context.

In stark contrast to its counterparts, both Iceberg's CoW and MoR strategies demonstrate exceptionally low and remarkably stable latencies for data maintenance. Consistently remaining low across all iterations, and showing only a minimal, almost unnoticeable increase as the number of update tasks grows, Iceberg exhibits superior scalability and efficiency in handling increasing update tasks. This highlights Iceberg 1.1.0's highly optimized implementations for both update strategies, effectively managing the underlying file operations and metadata even under a growing workload with minimal performance degradation.

Figure 7.4 presents the query latency for single user phases both before (SU-i) and after (SU-i-O) the optimize phase. Delta Lake's CoW configuration consistently exhibits high query latencies. The optimize phase appears to offer minimal benefit for Delta Lake's subsequent query performance as it maintains almost stably high across initial and post-optimization query phases.

Hudi CoW consistently delivers good initial query performance, and shows a clear trend of improving query performance after each optimize phase. This indicates that Hudi CoW's data layout benefits from compaction, allowing it to maintain competitive query speeds.

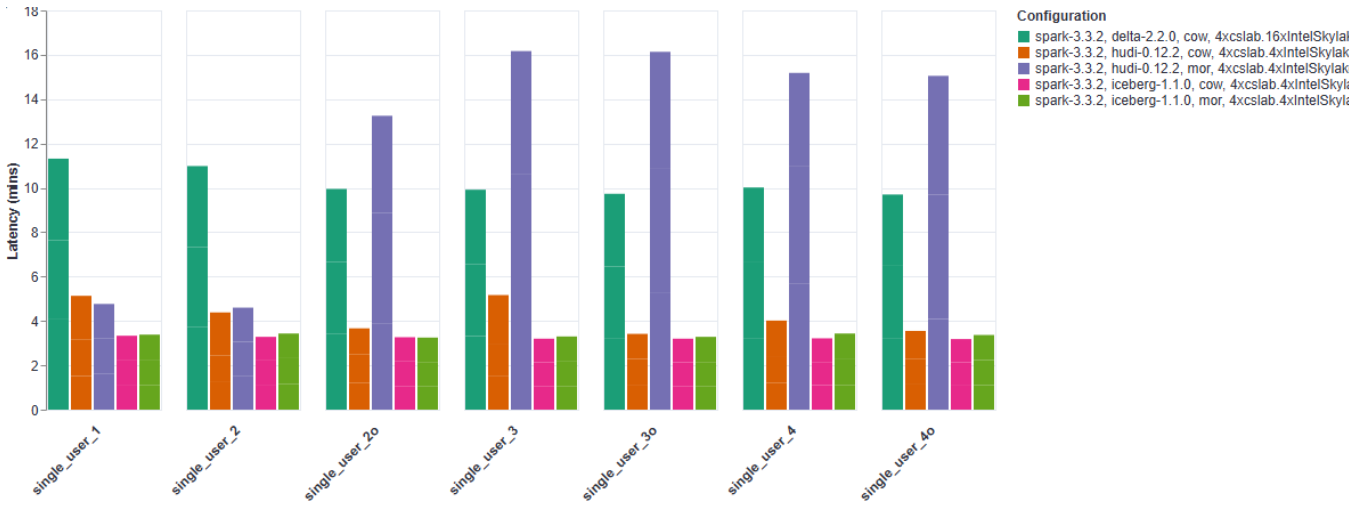

Figure 7.4. *Performance of WP-2 Single User phases before and after optimization (SF1000)*

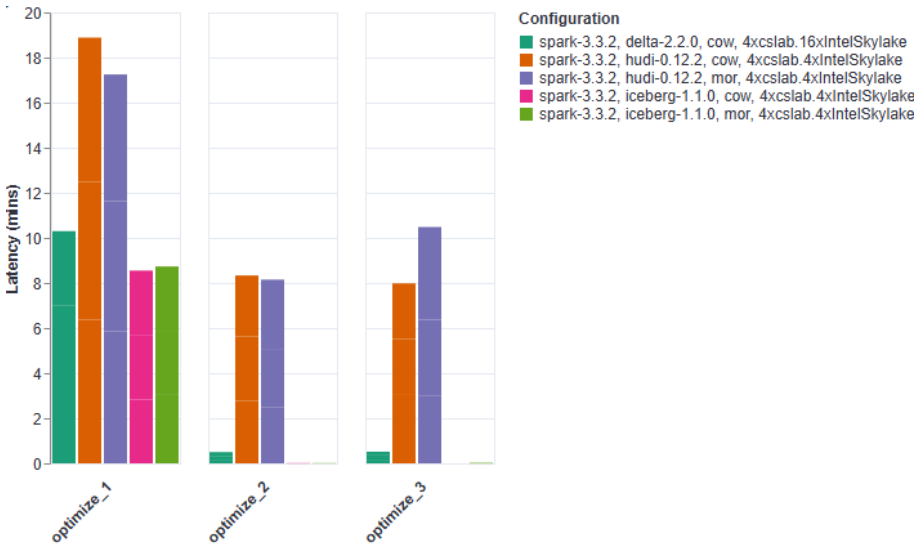

Figure 7.5. *Performance of WP-2 Optimize phases (SF1000)*

Hudi's MoR query latency experiences a substantial increase after the first optimize phase and continues to increase afterwards and staying high. This dramatic degradation in query performance post-optimization for Hudi MoR suggests that its "merge on read" strategy, rather than being aided by compaction, might be burdened by it, or that the resulting file structure from its maintenance and optimization phases becomes increasingly complex for read operations, leading to persistent and escalating overhead.

Both Iceberg configurations demonstrate consistently low and stable query latencies from the start, establishing them as the top performers for read operations. This near-perfect stability and efficiency, both before and after optimization, highlight Iceberg's robust ability to handle updates efficiently without compromising read performance.

Figure 7.5 displays the latency for the optimize phase, which compacts small files into larger ones to improve query efficiency. Delta Lake's configuration exhibits a remarkable and significant

decrease in latency across the optimize phases. It starts with a higher latency in optimize_1 and then drastically reduces its optimize time for optimize_2 and optimize_3. This suggests that while Delta Lake's data maintenance operations might initially produce a lot of small files or fragmentation, its explicit optimize command is exceptionally effective and efficient at cleaning up this state, achieving a highly optimized file layout quickly after the initial larger compaction.

Both Hudi's CoW and MoR strategies demonstrate very high initial latencies for optimize_1. While optimize_2 and optimize_3 show an improvement compared to optimize_1 for both, their latencies remain high compared to Delta Lake's and Iceberg's optimize times. This indicates that Hudi, regardless of strategy, requires more substantial and time-consuming compaction efforts to maintain an efficient file layout. Hudi MoR's optimize latency even shows an increase again in optimize_3, suggesting its complexity might grow over time.

Both Iceberg configurations show a behavior that is similar to Delta Lake's but consistently better in latency. Both Iceberg's CoW and MoR drop to negligible levels of latency for optimize_-2 and optimize_3 phases, highlighting their robust internal data organization and compaction efficiency.

In summary, for workloads requiring continuous data maintenance, efficient compaction, and stable query performance, Apache Iceberg provides a superior and more consistently resilient solution. Its architecture seems best equipped to handle the combined demands of this type of iterative workload.

7.3 Read/Write Concurrency

Up to this point, the sessions have been executed sequentially, utilizing all available resources. However, LSTs are designed with concurrency in mind. In this section, we investigate the effects of executing read and write sessions concurrently by employing WP3. Both WP2 and WP3 contain the same set and sequence of tasks; the only difference is that in WP3, the phases execute Single User tasks concurrently with either Data Maintenance or Optimize. Consequently, WP2 serves as the baseline for evaluating execution speedup and overheads.

The core of this evaluation involves contrasting the observed concurrent completion time in WP3 for a given phase against two critical WP2 baselines:

- Ideal Concurrent Baseline (MAX(sequential_query_time, sequential_other_task_time)): This theoretical minimum represents the best-case scenario where concurrent tasks execute in perfect isolation, and the overall duration is simply dictated by the longest individual task. Any duration in WP3 exceeding this value quantifies the overhead due to contention, locking, resource starvation, or inefficiencies in concurrency management. A smaller positive difference indicates better isolation, while a larger positive difference points to significant performance penalties.

- Total Sequential Baseline (sequential_query_time + sequential_other_task_time): This sum represents the time taken if the two concurrent tasks were to run strictly one after another. Comparing the WP3 concurrent time against this sum helps determine if the concurrent execution provides any efficiency gain over a purely serial approach (i.e., Concurrent_WP3 < Total_Seq_Time), or if the overheads are so severe that concurrency actually results in a

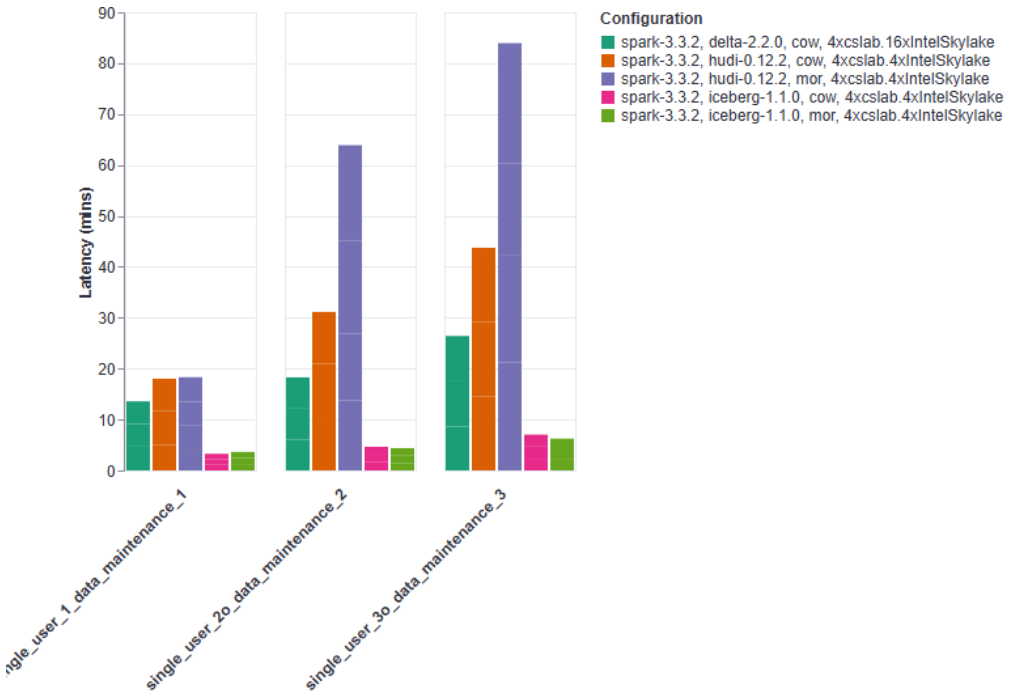

Figure 7.6. *Performance of WP-3 Concurrent Data Maintenance and Single User phases (SF1000)*

longer total duration than sequential execution (i.e., Concurrent_WP3 > Total_Seq_Time), signifying a critical failure in concurrency management.

By applying this comparative framework across all configurations and phases, we can accurately identify which LSTs are genuinely "concurrency-performant." Formats that exhibit minimal positive overhead relative to the ideal concurrent baseline (i.e., Concurrent_WP3 is close to MAX(sequential_times)) demonstrate superior architectural design for parallelism, effectively isolating concurrent operations. Conversely, LSTs showing significant positive overheads, especially when Concurrent_WP3 approaches or exceeds the Total_Seq_Time, indicate substantial bottlenecks and resource contention issues that severely limit their practical utility in concurrent environments. This rigorous comparison thus serves as the definitive metric for evaluating an LST's ability to maintain performance, predictability, and stability under the shared resource demands of modern data architectures.

The evaluation of concurrent Single User queries and Data Maintenance operations is depicted in Figure 7.6. A general trend observed across all configurations is that true "speedup" (where concurrent execution is significantly faster than the longest individual sequential task) is not achieved. Instead, the focus shifts to how effectively each LST minimizes the overhead introduced by running these demanding tasks in parallel.

Apache Iceberg (both CoW and MoR) stands out as the most robust performer. Both configurations consistently demonstrate exceptionally low overheads, with concurrent completion times only marginally exceeding the duration of their longest sequential component. This indicates highly effective isolation between concurrent update and query operations, allowing both to progress with minimal mutual interference. Iceberg's architectural design appears well-suited to handle the inherent resource contention of concurrent data maintenance and query workloads.

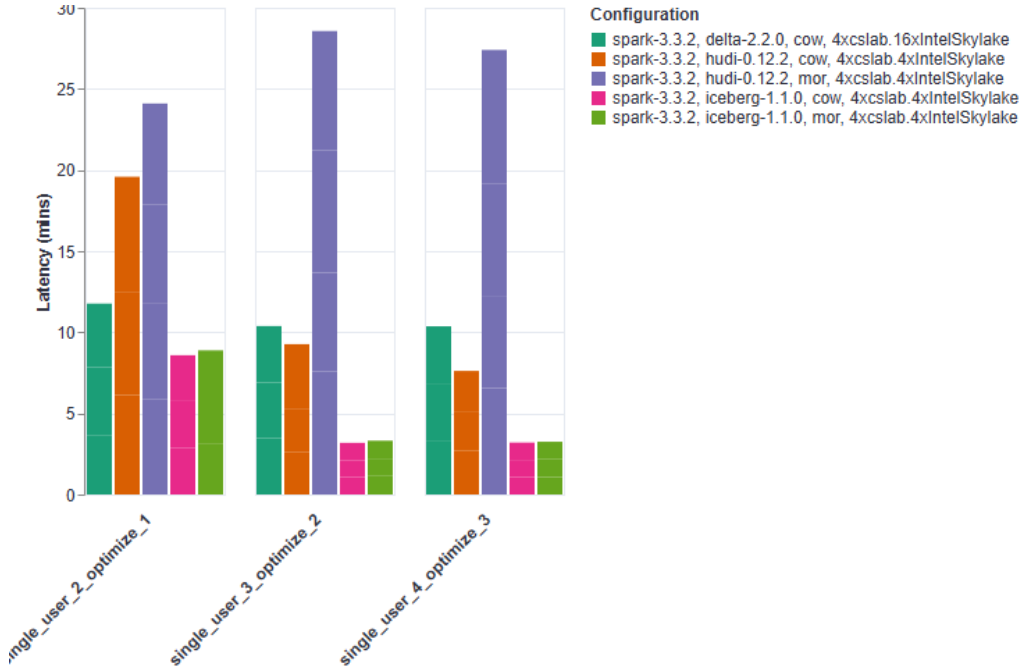

Figure 7.7. *Performance of WP-3 Concurrent Optimize and Single User phases (SF1000)*

In contrast, both Delta Lake CoW and Apache Hudi (CoW and MoR) exhibit considerable challenges in this concurrent scenario. For Delta Lake CoW, the concurrent completion times are often similar to or slightly higher than its already substantial sequential Data Maintenance latency. This suggests that its inherent inefficiencies in updating data, identified in WP2, become the primary bottleneck even when queries run concurrently, introducing noticeable overhead.

The most severe degradation is observed with Apache Hudi, particularly its MoR configuration. Both Hudi CoW and MoR show rapidly escalating concurrent latencies. Crucially, for Hudi MoR, the concurrent completion times in later iterations are not only significantly higher than the longest individual sequential task but also exceed the sum of their sequential execution times. This indicates a critical failure in concurrency management, where parallel execution introduces massive contention, leading to performance that is worse than if the update and query tasks were simply run one after another. Hudi CoW also experiences high concurrent latencies, primarily tracking its already high sequential Data Maintenance costs, but without the extreme compounding overhead seen in Hudi MoR.

The concurrent execution of Single User queries and Optimize phases is shown in Figure 7.7. Apache Iceberg (both CoW and MoR) again demonstrates superior efficiency. Following an initial, moderate overhead for the first optimize phase, both Iceberg configurations achieve near-ideal concurrent performance in subsequent iterations. Their concurrent completion times drop to levels very close to their already minimal sequential query latencies. This signifies that once the initial compaction is managed, Iceberg's subsequent lightweight optimize operations have virtually no discernible impact on concurrent query performance, showcasing excellent isolation.

Delta Lake CoW exhibits a pragmatic approach to concurrency during optimization. After an initial moderate overhead, its subsequent concurrent optimize phases complete in times that are only slightly higher than its sequential query performance, and well below the total sequential

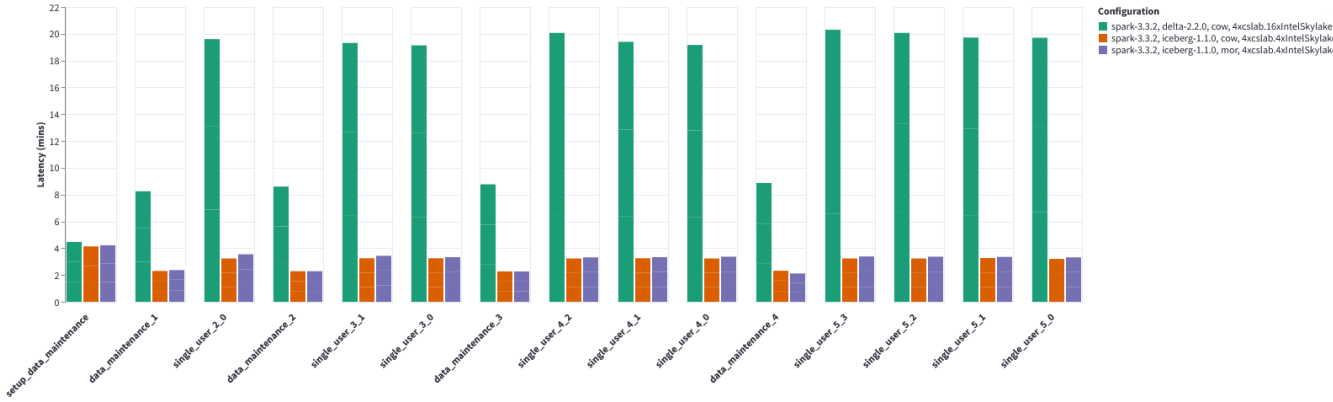

Figure 7.8. *Performance of WP-4 (SF1000)*

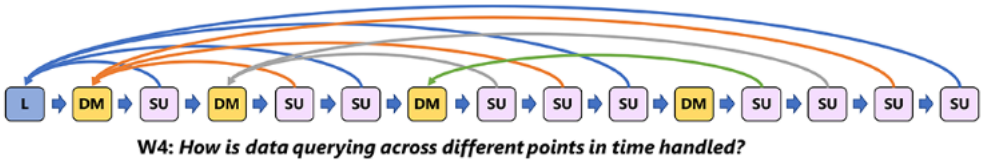

Figure 7.9. *Time Travel workload structure*

sum. This implies that while its optimize phase, once highly efficient in WP2, does introduce a small, measurable overhead to concurrent queries, it is generally manageable and does not lead to severe degradation.

Conversely, Apache Hudi (both CoW and MoR) struggles significantly with concurrent optimization. Both configurations exhibit substantial overheads, with concurrent completion times often considerably higher than the longest individual sequential task. For Hudi MoR, this overhead is particularly pronounced and escalating, where later concurrent optimize phases again take longer than the sum of their sequential counterparts. This highlights persistent contention and inefficient resource sharing, making concurrent optimization a bottleneck that severely impacts query performance. Hudi CoW also faces notable overheads, although not as extreme as MoR, indicating that its compaction process, even when run concurrently, is resource-intensive and prone to contention.

In essence, for environments where reads, writes, and maintenance tasks must run in parallel, Apache Iceberg provides the most reliable and efficient foundation, ensuring stable performance and minimal degradation.

7.4 Time Travel

A critical feature of modern Data Lake Table Formats is the ability to perform "time travel" queries, allowing users to access historical versions of data, reconstruct past states, or debug changes. This section evaluates the performance of such queries. To better clarify which data are queried at each phase, Figure 7.9 illustrates the structure of the Time Travel workload. It is important to note that Apache Hudi, as mentioned in a previous chapter, is not represented in these time travel benchmarks, as the selected version does not support this feature.

All the phases of the Time Travel workload results are shown on Figure 7.8. Apache Iceberg (both CoW and MoR) demonstrably offers vastly superior time travel performance. Its low and stable latencies for both initial data setup and subsequent historical queries make time travel a highly practical and efficient feature. This strength can be attributed to its metadata management, which is optimized for quick access to various table snapshots, regardless of the data's age.

Delta Lake, while supporting time travel, incurs a substantial and consistent performance penalty. Its query latencies for historical versions are significantly higher than Iceberg, suggesting that accessing past states introduces considerable overhead. This might make frequent or interactive time travel queries less viable for performance-critical applications using Delta Lake in this configuration.

In summary, for use cases demanding efficient and fast access to historical data states, Apache Iceberg proves to be the leading solution, offering predictable, low-latency time travel capabilities that are essential for auditing, debugging, and advanced analytics.

Chapter 8

Conclusion

In this paper, we presented a comprehensive performance evaluation of three leading Data Lake Table Formats (LSTs): Delta Lake, Apache Hudi, and Apache Iceberg. The study leveraged the lst-bench benchmarking suite, executed on Apache Spark over HDFS, to systematically assess these formats across critical dimensions: longevity during frequent data updates, resilience during sequential data operations, performance under concurrent read/write workloads, and the efficiency of time travel queries. The experiments were conducted in a virtualized cloud environment, reflecting common modern deployment scenarios.

The study underscores that while LSTs fundamentally enhance data lake capabilities, their implementations vary dramatically in their efficiency and architectural resilience. Apache Iceberg consistently demonstrated the most robust and efficient performance across all critical dimensions, proving to be the most "concurrent-ready" and resilient solution for dynamic data environments. Delta Lake, while functionally rich, faces performance challenges in update-heavy and time travel scenarios that demand careful workload analysis. Apache Hudi, particularly in concurrent and update-intensive workloads, exhibited significant scalability and contention issues, indicating areas for further optimization to meet the demands of modern data architectures. These findings are crucial for architects and practitioners facing the complex decision of selecting an appropriate lakehouse architecture, tailored to their operational needs and data volume, providing the practical insights necessary for informed decision-making.

It is important to acknowledge that the versions of the LSTs utilized in this study—Delta Lake 2.2.0, Apache Hudi 0.12.2, and Apache Iceberg 1.1.0—correspond to those supported by LST-Bench. Given the rapid pace of development in the open-source data lake ecosystem, these technologies continually evolve, introducing new optimizations, features, and performance improvements with each release. Therefore, while our findings provide a valuable and rigorously established snapshot of their performance characteristics under the tested conditions, it is plausible that more recent versions may exhibit different behaviors. Nevertheless, this work offers crucial architectural insights into the design trade-offs and underlying performance drivers of these prominent LSTs. It highlights fundamental strengths and weaknesses that are often indicative of their core design principles, thereby furnishing practitioners and researchers with a foundational understanding necessary for informed decision-making and guiding future development efforts in the ever-evolving landscape of data lake architectures. The rapid evolution of these data lake table formats ensures that continuous evaluation and adaptation remain paramount for optimal data management strategies.

References

[1] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin and Larisa Safina. *Microservices: yesterday, today, and tomorrow. Present and ulterior software engineering*, sel'ides 195--216, 2017.

[2] Jesús Camacho-Rodríguez, Ashvin Agrawal, Anja Gruenheid, Ashit Gosalia, Cristian Petculescu, Josep Aguilar-Saborit, Avrilia Floratou, Carlo Curino and Raghu Ramakrishnan. *LST-Bench: Benchmarking Log-Structured Tables in the Cloud. Proceedings of the ACM on Management of Data*, 2(1):1--26, 2024.

[3] Jesús Camacho-Rodríguez, Ashvin Agrawal, Anja Gruenheid, Ashit Gosalia, Cristian Petculescu, Josep Aguilar-Saborit, Avrilia Floratou, Carlo Curino and Raghu Ramakrishnan. *LST-Bench: Benchmarking Log-Structured Tables in the Cloud. Proceedings of the ACM on Management of Data*, 2(1):1–26, 2024.

[4] Barry Devlin. *Thirty Years of Data Warehousing*. https://www.9sight.com/pdfs/Thirty_Years_of_DW.pdf, 2018.

[5] Jim Holdsworth and Matthew Kosinski. *What is a Data Warehouse?* https://www.ibm.com/think/topics/data-warehouse, 2024.

[6] B. Haelen and D. Davis. *Delta Lake: Up and Running*. O'Reilly Media, 2023.

[7] James Dixon. *Pentaho, hadoop, and data lakes | james dixon's blog*. https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/, 2010.

[8] Matei A. Zaharia, Ali Ghodsi, Reynold Xin and Michael Armbrust. *Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. Conference on Innovative Data Systems Research*, 2021.

[9] *Delta Lake*. https://delta.io/, 2025.

[10] D. Lee, T. Wentling, S. Haines and P. Babu. *Delta Lake: The Definitive Guide: Modern Data Lakehouse Architectures with Data Lakes*. O'Reilly Media, 2024.

[11] *Apache Iceberg*. https://iceberg.apache.org/, 2025.

[12] Srujana Maddula. *Apache Iceberg Explained: A Complete Guide for Beginners*. https://www.datacamp.com/tutorial/apache-iceberg, 2024.

[13] *Apache Hudi*. https://hudi.apache.org/, 2025.

[14] *Apache Hudi Docs*. https://hudi.apache.org/docs/overview/.

[15] Databricks. *tpcds-kit: Databricks fork of the TPC-DS data generation kit*. https://github.com/databricks/tpcds-kit, 2020.

[16] *Apache Spark*. https://spark.apache.org/, 2025.

[17] *Apache Hadoop*. https://hadoop.apache.org/, 2025.

[18] *Apache Hadoop 2.7.1*. https://hadoop.apache.org/docs/r2.7.1/, 2015.

[19] *Apache Hive*. https://hive.apache.org/, 2025.

[20] *PostgreSQL*. https://www.postgresql.org/, 2025.