Batch Updates and CDC at Scale: A Comparative Study of Iceberg and Paimon

Kirill Ievlev
Moscow Technical University of Communications and Informatics
Moscow, Russia
ievlev.k.o@yandex.ru

Vadim Surpin
Moscow, Russia
vadim.supin@mail.ru

***Abstract*—Business digitalization in data-intensive sectors such as fintech, advertising, and telecommunications has shifted early-stage storage toward extract–load–transform (ELT)-centric data lakes, while the need for transactional guarantees and SQL semantics has accelerated the rise of data lakehouse architecture. This paper evaluates the practical use of open table formats within a financial analytics platform, focusing on Apache Iceberg and Apache Paimon.**

We report two experiments. First, we model data-mart maintenance by generating a 1 TB table (~10 billion keys) and executing staged SELECT/UPDATE operations over 10–50% of keys, exposing how table format choices affect update-heavy patterns. Second, we assess ingestion via a Change Data Capture (CDC)–Kafka–Flink–Paimon pipeline that replays 1.5 billion CDC operations per day with strong recency skew and assess its viability for long-term data-mart storage and incremental updates without intermediate databases. We also benchmark a CDC–Kafka–Spark–Parquet ingestion approach to estimate the performance gain by using the open table format. Our findings surface design trade-offs that materially influence read and write performance and operational overhead, and we derive practical guidance for selecting open table formats when building lakehouse architectures in financial settings.

I. Introduction

Modern banks operate hundreds of information systems that support operational activities across various business lines. Rather than rebuilding data management from scratch, organizations favor incremental modernization and integration with existing solutions [1], [2]. These realities raise the bar for low-latency data processing and flexible integration—needs well served by open table formats (OTFs). OTFs are the foundation of the data lakehouse, which unifies data lake scalability with data warehouse semantics across batch and streaming workloads [1]. In what follows, we review prevalent OTFs and discuss which are best suited to integration workloads and to improving processing timeliness.

II. Overview of open table formats

Between 2017 and 2019, Apache Hudi, Delta Lake, and Apache Iceberg emerged as open table formats for data lakes and lakehouse architectures, adding transactional semantics and governance on top of file-based storage (typically Apache Parquet) across S3, Hadoop Distributed File System (HDFS) and other types of distributed storage systems. Their shared goal is to provide a unified tabular abstraction over raw files. All three offer table-level atomicity, consistency, isolation, durability (ACID) guarantees, snapshots, and atomic commits, enabling a wide range of enterprise workloads—from BI and ad hoc analytics to MLOps and streaming—while lowering total cost of ownership and accelerating time to insight [3], [4].

Although Delta is open source and under the Linux Foundation, its ecosystem remains closely coupled with Databricks: many key optimizations and the best user experience are delivered through the Databricks cloud (e.g., Unity Catalog, Photon, Delta Live Tables, advanced optimizers and automation), implying stronger vendor lock-in relative to alternatives.

As of August 2025 (per GitHub repository statistics; see Table I), Iceberg leads Hudi in community signals—stars/forks (7841/2730 vs 5912/2432), total commits (7336 vs 6574), and fewer open issues at larger scale—suggesting higher development velocity and broader adoption. Accordingly, Iceberg appears to be the more popular choice for cross-engine analytics, while Hudi remains active and mature in its core niche of real-time/CDC and incremental pipelines.

TABLE I. GitHub Community and Development Signals for Apache Hudi and Apache Iceberg (Aug. 2025)

OTF	Apache Hudi	Apache Iceberg
Total commits	6574	7336
Stars	5912	7841
Forks	2432	2730
Open issues	1085	610
Origins	2016	2017
Apache Foundation Top-Level Project announcement	June 4, 2020	May 20, 2020

Therefore, we chose Apache Iceberg for our study.

Since 2022, Apache Paimon has emerged as an open table format and storage engine for "streaming-lakehouse" workloads. Originating as Flink Table Store (announced by the Apache Flink community in June 2022), it was spun out as the Apache Paimon incubating project in September 2022 and graduated to Top-Level Project status in April 2024 [5]. Paimon targets primary-keyed datasets, high-frequency upserts/deletes, and continuous change processing, reflecting users' need to unify streaming and analytical processing over object storage and distributed file systems without tight coupling to a specific Database Management System (DBMS) [6]. Given its recent spin-out, we do not directly compare Paimon and Iceberg using GitHub repository metrics.

Paimon's storage is log-structured merge-tree (LSM)-inspired: records are hashed into buckets by primary key (or by full row in its absence) and persisted as ordered

Sorted String Table (SST) runs [7]. LSM-tree designs are widely used (e.g., ClickHouse, HBase) and are well suited to streaming-heavy write patterns [8]. Paimon additionally supports choosing row- or column-oriented formats per LSM level [9], a capability that could position it for Hybrid Transactional/Analytical Processing (HTAP) scenarios.

Next, we examine pipeline mechanisms using OTFs through the lens of their architectures. We first discuss the semantics and costs of UPDATE/DELETE operations, then turn to key performance aspects of streaming ingestion.

III. Architectural characteristics of Apache Iceberg and Apache Paimon

A. Drawbacks of data lake table storage

Hadoop-based data lakes typically store data in Parquet or ORC [1], [3], [10]. When updates are required, the common approach is to rewrite an entire table or partition. At scale this is compute- and I/O-intensive and, due to the absence of snapshot-level isolation, readers may observe transient errors or inconsistent states during rewrites [1, 10]. A central component of this stack is the Hive Metastore (HMS), which catalogs schemas, partitions, physical locations, and statistics for compute engines [10]. Fig. 1 shows a data storage scheme when using Hive Metastore and Parquet. In practice, HMS introduces constraints: metadata drift when files are added outside the metastore (necessitating MSCK REPAIR/ALTER PARTITION sweeps), limited scalability with very large partition counts (slow listing and planning, high load on the backing RDBMS), no snapshot consistency and limited ACID guarantees for Parquet tables, and high operational cost for bulk DDL and statistics maintenance [3], [10].

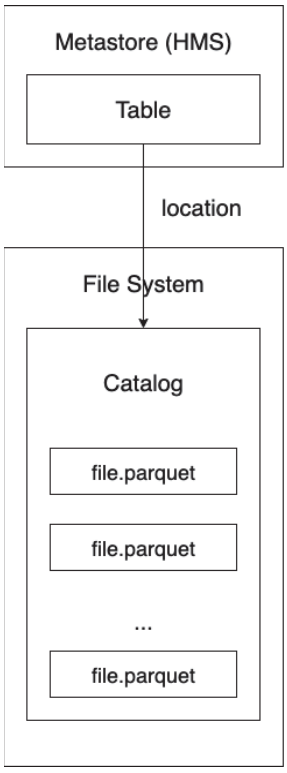

Fig. 1. Hive Metastore + Parquet storage system design

B. Snapshot abstraction in OTFs

Open table formats such as Apache Iceberg and Apache Paimon introduce the data "snapshot" abstraction. A snapshot is metadata that points to a specific set of immutable data files; on updates, only a subset of files is replaced, and the metadata atomically switches to a new snapshot [10]. This ensures an atomic version switch and immediate data consistency: a consumer reads a strictly defined snapshot without observing partially applied changes. As a result, the volume of rewritten data is reduced compared with a "full rewrite", which positively affects latency and resource consumption for client tasks. Integration for existing pipelines is generally transparent: read/write operations gain performance and functionality (atomic commits, schema evolution management, time travel, etc.) without substantial changes to user code [3, 10].

C. Divergent update design

The differences between Iceberg and Paimon are largely determined by their change-commit mechanisms and storage organization. Iceberg supports two modes for row-level changes: copy-on-write (COW) and merge-on-read (MOR). In MOR mode, updates and deletes are recorded via "delete files" (position deletes and/or equality deletes), and new/replacement rows go into new data files; readers apply delete masks when reading the corresponding snapshot. In COW mode, files containing modified rows are repackaged in full. Both approaches are committed as new metadata snapshots and provide instant version switching for readers [10]. The simplified data update scheme in Apache Iceberg is presented in Fig. 2.

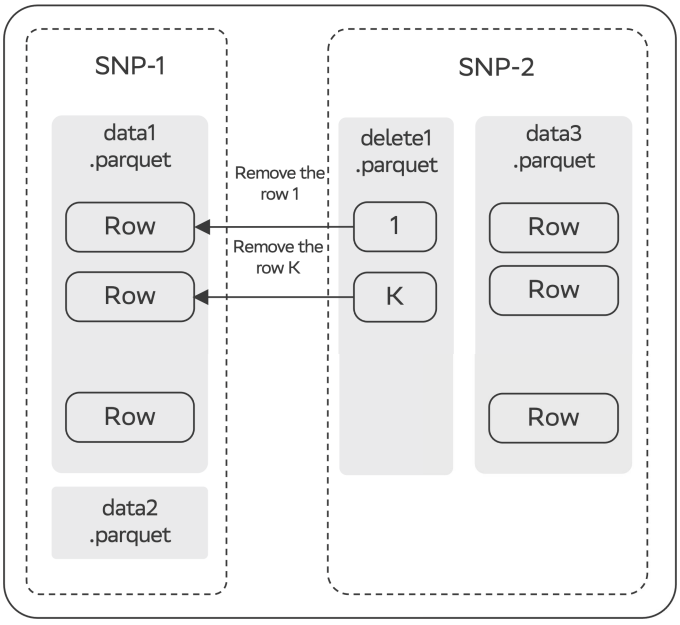

Fig. 2. Simplified data update scheme in Apache Iceberg

Paimon uses a different storage organization—based on an LSM tree with buckets—which is oriented toward key-based updates. When a primary key is present, an update is represented as an upsert: a new version of the key is added to

the next snapshot without the need to read the previous state [11]. Consolidation and ordering of versions are performed via background compactions. In typical scenarios, this approach reduces write cost and latency compared with repackaging files or extensive use of delete files. The data update scheme in Apache Paimon is presented in Fig. 3. It should be noted that updates/inserts in Paimon are appended as new runs and reconciled on read, with background compaction merging files later. This is merge-on-read by design. There is no copy-on-write storage mode Paimon.

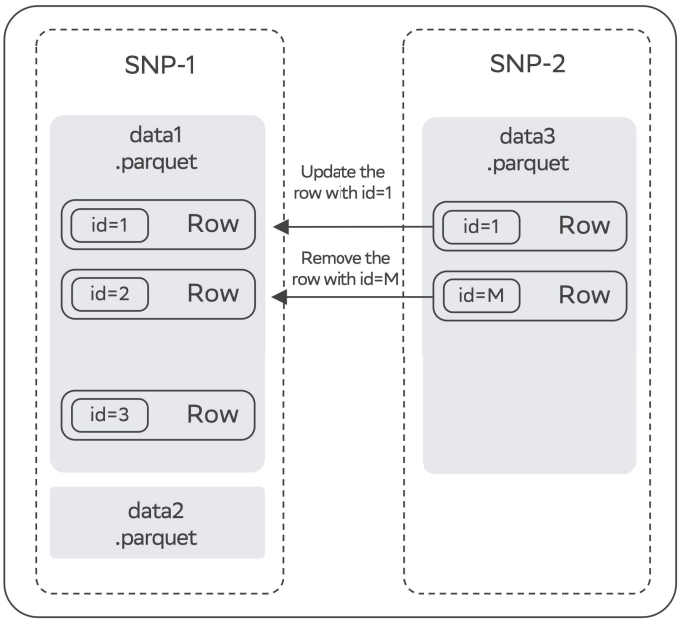

Fig. 3. Simplified data update scheme in Apache Paimon

D. Streaming ingestion patterns

When processing an incoming data stream (e.g., from Apache Kafka), Iceberg-based pipelines running on Spark commonly adopt a match-against-current-state approach. The system plans execution using table metadata, reads only the relevant partitions, and joins the arriving increment with the current snapshot to identify changed keys or rows. Despite effective partition pruning, this strategy incurs substantial read I/O and shuffle/join stages, which lowers end-to-end pipeline throughput [10]. A simplified stream processing scheme with Iceberg is described in Fig. 4.

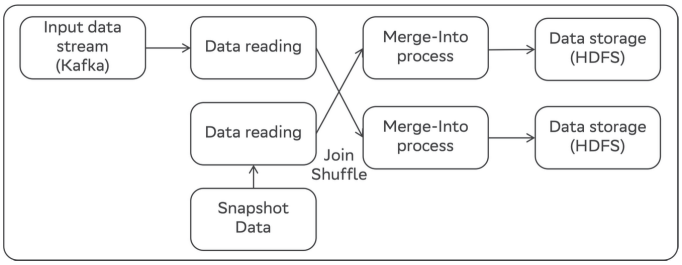

Fig. 4. Data update scheme with Apache Iceberg (MOR approach)

In Paimon, the analogous task is typically optimized on the write path. Kafka stream is ingested as a sequence of upsert operations; records are shuffled by bucket and sorted within each bucket, after which background processes compact the data asynchronously [12]. Because only the incremental batch (rather than the entire snapshot) is sorted and merged, network and disk overhead are generally lower than with full matching against the persisted state. In most practical cases, this yields an advantage for Paimon in both write latency and write cost.

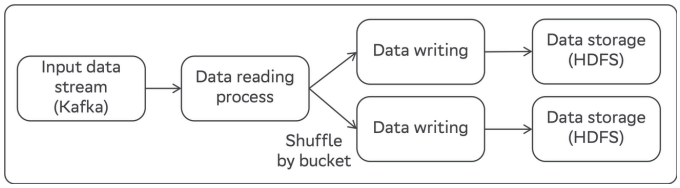

Fig. 5. Data update scheme with Apache Paimon

IV. Methodology and Experimental Design for Data Update

A. Experiment description

We conduct an experimental comparison of operation latency and throughput on the Iceberg and Paimon OTFs. To assess comparative performance across Parquet, Iceberg, and Paimon, we generate a large synthetic dataset and execute a staged workload comprising sequential SELECT and UPDATE operations. This methodology intentionally abstracts away pipeline complexity to approximate data-mart construction while isolating storage-level effects. The resulting measurements reveal practical differences in how the evaluated table formats manage data updates and reads, providing empirically grounded guidance for format selection in real-world deployments.

B. Dataset description and table structure

The following datasets are used for experiments (Table II).

TABLE II. Datasets description

Table name	Description
parquet_10b_part	Partitioned Parquet Table (1 TB)
iceberg_10b_cow	Partitioned Iceberg Table (copy-on-write mode, 1 TB)
iceberg_10b_mor	Partitioned Iceberg Table (merge-on-read mode, 1 TB)
paimon_10b_part	Partitioned Paimon Table (merge-on-read mode, 1 TB)

All datasets consist of the table of the following format (Table III):

TABLE III. Table format description

Field name	Type	Description
id	bigint	integer value from 0 to 10,000,000,000
id_str	string	string in the form "id_" + a number from 1 to 10,000,000,000, left-padded with zeros to a fixed length of 11 characters
data	string	random ASCII characters (a 100-character string)
dt	string	current datetime (string) in the format "%Y-%m-%d %H:%M:%S"
part	int	integer value from 0 to 10,000

The data sample is presented in Table IV.

TABLE IV. Table data sample

id	id_str	data	dt	part
0	id_00000000000	VTVEU…	2025-03-07 21:40:08	0
1	id_00000000001	J6TPR…	2025-03-07 21:40:08	0
…				
9999999998	id_09999999998	TVCMJ…	2025-03-07 22:02:12	99999
9999999999	id_09999999999	G4839…	2025-03-07 22:02:12	99999

Table V summarizes the hardware configuration of the cluster used for our experiments:

TABLE V. CLUSTER HARDWARE CONFIGURATION

Parameter	Specification
CPU	Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz
RAM	768 GB (32 Gb x 24) DDR4 2400MHz
Disk Info	6Tb SATA III 7200rpm 128Mb
Number of worker nodes	4

C. Component versions

Tests were carried out using the following versions of programs (Table VI):

TABLE VI. CLUSTER SOFTWARE COMPONENTS VERSIONS

Software name	Software version
Apache Parquet	1.13.1
Apache Iceberg	1.6.1
Apache Paimon	1.1.1
Apache Spark	3.5.1
Apache Flink	1.18

D. Experiment description

- Perform a read of 10% of the data using a filter on the textual copy of the identifier

$$T\_write = a + b \cdot p \quad (1)$$

```
select count(*) from {table} where id_str < 'id_01000000000'
```

- Perform an update of 10% of the data using a filter on the textual copy of the identifier

```
update {table} set dt = {current} where id_str < 'id_01000000000' where id % 3 = 0
```

- Perform a read of the previously updated 10% of the data using a filter on the textual copy of the identifier

```
select count(*) from {table} where id_str < 'id_01000000000'
```

- Perform an update of 20% of the data, with the first 10% updated twice

```
update {table} set dt = {current} where id_str < 'id_02000000000' where id % 3 = 1
```

- Perform a read of 10% of the data that were updated twice

```
select count(*) from {table} where id_str < 'id_01000000000'
```

- Perform an update of 50% of the data, with the first 10% updated three times and the first 20% updated twice

```
update {table} set dt = {current} where id_str < 'id_05000000000' where id % 3 = 2
```

- Perform a read of 10% of the data that were updated three times

```
select count(*) from {table} where id_str < 'id_01000000000'
```

It should be noted, that the "id % 3" operation is used to avoid rewriting entire data files.

IV. EXPERIMENTAL RESULTS AND ANALYSIS OF DATA UPDATE

The results of the experiment are presented in Table VII.

TABLE VII. COMMAND EXECUTION TIME PER STEP

Operation	Type	Parquet part	Iceberg (CoW)	Iceberg (MoR)	Paimon
1. SELECT 10%	read	00:57	00:33	00:47	00:37
2. UPDATE 10%	write	06:00	02:18	02:58	03:22
3. SELECT 10%	read	00:57	00:33	00:45	00:37
4. UPDATE 20%	write	12:00	03:54	05:22	06:00
5. SELECT 10%	read	00:57	00:34	00:47	00:35
6. UPDATE 50%	write	30:00	07:39	11:28	14:28
7. SELECT 10%	read	00:57	00:35	00:50	00:36

We can conclude that a simple affine model (1) explains the observed write times with high fidelity and provides actionable predictions and break-even thresholds among Parquet, Iceberg (COW/MOR), and Paimon for batch UPDATE workloads on a 1 TB table under the stated cluster configuration:

Where

- T_write is end-to-end write time for the step (minutes),
- *p* - fraction of rows updated in a step (in percent of the 1 TB table),
- *a* (minutes, fixed overhead) – engine- and system-level overheads that do not scale with p. This term captures job planning and initialization (e.g., Spark/Flink scheduling), metadata and commit operations (snapshot/transaction), file/connection setup, and the minimal I/O required even for very small updates. In the limit $p \to 0$, $T\_write \approx a$,
- *b* (minutes per percentage point) – marginal cost per additional 1% of updated data. This term reflects the scalable portion of work: file creation/rewrites, generation and handling of delete files (for MOR modes), shuffle and

sorting (and bucketing in Paimon), write amplification, and portions of compaction work. Smaller b indicates cheaper scaling of updates.

The visualization of the resulting mathematical model is presented in Fig. 6.

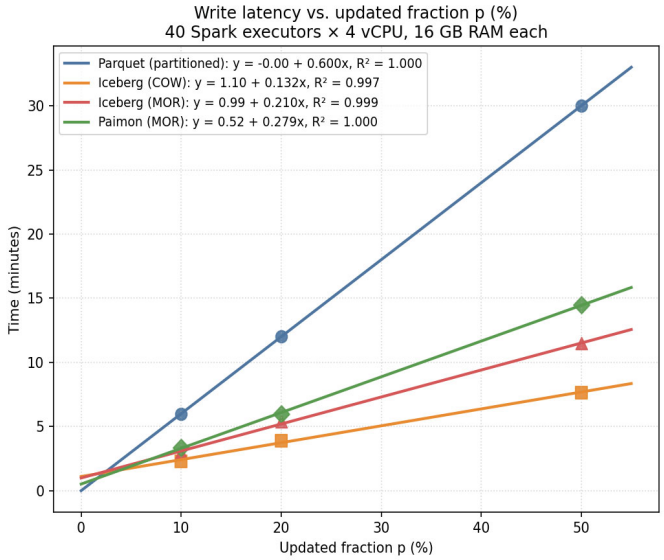

Fig. 6. Write-time model visualization for data update

Linear models for updates exhibit very high explanatory power ($R^2 \geq 0.995$) for all formats, indicating that, at the tested scales, per-step write time is well approximated by a fixed overhead plus a term linear in the updated fraction.

Empirical estimates in our setup (40 Spark executors; 4 vCPU and 16 GB RAM each) illustrate these roles:

- Parquet (partitioned): $a \approx 0$, $b \approx 36.0$ ($R^2 = 1.000$)

Interpretation: purely proportional rewriting cost.

- Iceberg (COW): $a \approx 57.8$, $b \approx 8.03$ ($R^2 \approx 0.995$)

Interpretation: non-trivial fixed overhead (planning/commit), low marginal cost per percent due to file-level rewrites of affected data only.

- Iceberg (MOR): $a \approx 50.5$, $b \approx 12.75$ ($R^2 \approx 0.998$)

Interpretation: lower fixed overhead than COW but higher marginal cost (delete files + later reconciliation).

- Paimon (MOR): $a \approx 35.5$, $b \approx 16.65$ ($R^2 \approx 0.999$)

Interpretation: small fixed overhead; higher marginal cost due to shuffle/bucketing and subsequent compactions.

V. Methodology and Experimental Design for Warehouse Data Ingestion

The prior experiment evaluated data-mart–oriented mutations (modifying records already stored in the warehouse). Here, we examine warehouse ingestion and how OTFs support this task efficiently. In banking settings, most sources are high-throughput relational Online Transaction Processing (OLTP) systems. Practically viable extraction methods are limited to:

- Change Data Capture (CDC)
- Application-level data capture

We exclude CDC replication into an intermediate database with subsequent export to the warehouse because it materially increases operational cost. In practice, CDC at the source is easier to deploy (no source code changes), whereas application-level capture requires per-system development, integration, and testing [13]. Accordingly, we focus on CDC.

Under this approach, insert/update/delete events are streamed from the OLTP database to an intermediate transport layer — typically Kafka — then consumed by a warehouse application and persisted in long-term columnar formats (Parquet or ORC) suitable for OLAP.

A. Experiment description with CDC–Kafka–Flink–Paimon pipeline

Our experiment implements a representative CDC–Kafka–Flink–Paimon pipeline (Fig. 7).

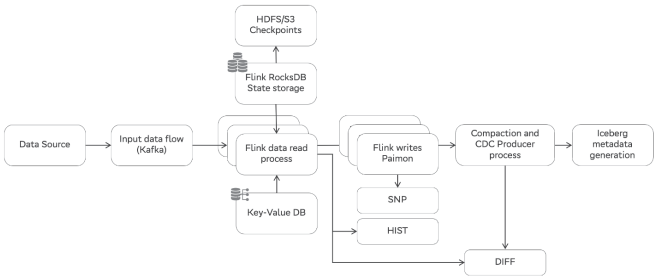

Fig. 7 Loading pipeline architecture using Flink and Paimon

We stress-test the system with a recency-biased temporal distribution that is demanding for traditional data lakes: daily partitioning with most modifications concentrated in the last three days, and the remainder uniformly distributed across the previous 360 days (Table VIII).

We additionally model long per-key CDC chains (distribution by chain length omitted here for brevity). The simulated workload updates 1 billion primary keys per day, yielding 1.5 billion CDC events per day in total.

TABLE VIII. Daily Partition Change Probabilities with Recency Skew

Date	Data change probability
Today	51.79%
Yesterday	25.89%
Two days ago	12.95%
…	…
Other days	~0.03%
…	…
1 year ago	~0.03%

B. Alternative warehouse ingestion architecture: CDC–Kafka–Spark (micro-batch)–Parquet

To benchmark against the Flink–Paimon design (Fig. 7), we also implemented a two-stage Spark pipeline (Fig. 9). The architecture separates the generation of row-level diffs from their application, thereby making the impact of micro-batch sizing and file-rewrite behavior explicit.

To ensure comparability with the preceding experiment:

Workload: same recency-biased temporal distribution (Table VIII), daily partitioning, and long per-key CDC chains; 1 billion primary keys/day yielding 1.5 billion CDC events/day, identical cluster configuration (Table V), the same Kafka CDC topics and schemas.

Concept and data flow:

- Diff producer. A Spark job in micro-batch mode consumes CDC events from Kafka and materializes row-level changes as Parquet "diff files" in a staging area. The job can optionally consult a "previous state" source—either a prior Parquet snapshot in the lake or a Key-Value store—to compute precise deltas per primary key.
- Diff apply. A second Spark micro-batch job reads the staged diffs and merges them with the previous state to produce a "new state" Parquet. A maintenance phase (compaction/file coalescing) follows to mitigate small-file proliferation. Downstream engines then query the optimized data.

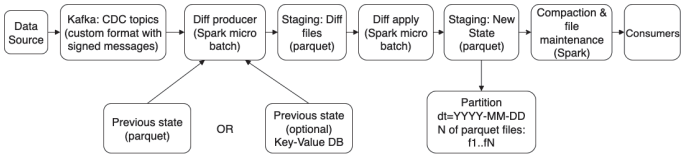

Fig. 9. Alternative warehouse-ingestion architecture: CDC–Kafka–Spark (micro-batch)–Parquet

The "new state" dataset is partitioned by date same as in the previous experiment (dt=YYYY-MM-DD), each partition comprising N Parquet files. Let B denote the micro-batch size (number of CDC messages processed per batch). Within an updated partition, the probability of rewriting k files grows with the "pressure" B/N. In particular, we observe:

- p1(B): probability that exactly 1 of N files is rewritten;
- p2(B): probability that exactly 2 of N files are rewritten;
- p1(B) + p2(B) increases with B/N because larger batches are more likely to touch multiple file ranges in the same partition.

Which files are rewritten is governed by the specific key ranges/rows affected by the batch. Even single-row updates can trigger whole-file rewrites, and compaction cannot eliminate the write amplification inherent to file-granular formats.

Micro-batch sizing induces a classical balance:

- Small batches: frequent MERGE operations cause repeated full-file rewrites, metadata growth, and frequent commits (high write amplification and coordinator overhead).
- Large batches: heavyweight MERGE operations increase memory pressure (shuffle/join/state), cause disk spills, and raise end-to-end batch latency.

VI. Experimental results and analysis of warehouse data ingestion

On a cluster with the configuration in Table V, the proposed architecture (Fig. 7) processes this feed in 1 hour 34 minutes while producing 15-minute snapshots, comfortably meeting single-source daily ingestion needs and enabling accelerated catch-up after failures.

The Spark micro-batch pipeline (Data Source-Kafka-Spark micro-batch-Parquet, including compaction, required 3 hours 14 minutes with its best-tuned micro-batch configuration, resulting in a 2 times longer end-to-end runtime.

The gap is explained by:

- File-level write amplification during MERGE, which grows with B/N and produces metadata overhead and frequent small commits for smaller batches.
- Resource pressure for larger batches (shuffle/join/state), which increases spill and latency without fully offsetting rewrite costs.

VII. Conclusion

- Across our experiments, both OTFs outperformed the partitioned Parquet baseline for updates and reads. Read-side gains arise from metadata-driven statistics pruning in Iceberg and Paimon, whereas Parquet must scan the footers of all files to enforce the id_str < 10% filter.
- Iceberg in MOR exhibits higher latency for reads (as expected) and also for writes; the latter warrants further investigation. A plausible cause is the overhead of producing and handling delete files. In both cases, the penalty scales with the fraction of modified data.
- Paimon is slower than Iceberg COW on writes (substantially) and on reads (approximately proportional to the amount of modified data), yet it reads faster than Iceberg MOR. The write-side penalty is attributable to the additional shuffle for sorting and bucketing. Notably, we used UPDATE for interpretability, which avoids shuffle; in production, MERGE INTO is more representative. Under MERGE INTO, Iceberg's advantage over Paimon is expected to narrow, and with Paimon CDC Injection the advantage may shift to Paimon.
- The CDC–Kafka–Flink–Paimon pipeline processed 1.5 billion CDC events in 1.5 hours, corresponding to an average rate of ≈2.78×10^5 events/s. The daily workload is ingested about 16 times faster than real time (24 h / 1.5 h), enabling rapid catch-up after outages.

- Producing snapshots every 15 minutes with CDC–Kafka–Flink–Paimon pipeline demonstrated manageable commit overheads and consistency preservation without intermediate databases.
- The Flink–Paimon pipeline design remains more efficient for high-volume CDC ingestion into day-partitioned lakes compared with the Spark pipeline. In our setting, it achieves the same ingestion objective in half the time, primarily by reducing write amplification and avoiding the micro-batch trade-off that forces a choice between frequent small rewrites and infrequent large, memory-intensive merges.

ACKNOWLEDGMENT

The authors thank Moscow Technical University of Communications and Informatics (MTUCI) for providing the facilities to conduct this research.

REFERENCES

[1] J. Schneider, C. Gröger, A. Lutsch, H. Schwarz, and B. Mitschang, "The lakehouse: State of the art on concepts and technologies," *SN Computer Science*, vol. 5, 2024, pp. 1-39.

[2] R. Cao and M. Iansiti, "Digital transformation, data architecture, and legacy systems," *Journal of Digital Economy*, vol. 1, 2022, pp. 1–19.

[3] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics," *11th Annual Conference on Innovative Data Systems Research (CIDR '21)*, 2021, pp. 1-8.

[4] A. R. Alleni, "AI/ML optimized lakehouse architecture: A Comprehensive framework for modern data science", *World Journal of Advanced Engineering Technology and Sciences*, 2025, pp. 2099-2104.

[5] The Apache Software Foundation Blog, Apache Software Foundation Announces New Top-Level Project Apache® Paimon, Web: https://news.apache.org/foundation/entry/apache-software-foundation-announces-new-top-level-project-apache-paimon.

[6] Apache Paimon Documentation, Apache Paimon, Web: https://paimon.apache.org/docs/1.1.

[7] Apache Paimon Documentation, Understand Files, Web: https://paimon.apache.org/docs/1.1/learn-paimon/understand-files.

[8] S. Mishra, "A survey of LSM-Tree based Indexes, Data Systems and KV-stores", *IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*, vol. 1, 2024, pp. 1-6.

[9] Apache Paimon Documentation, Configurations Web: https://paimon.apache.org/docs/master/maintenance/configurations

[10] T. T. Shiran, J. Hughes, and A. Merced, *Apache Iceberg: The Definitive Guide*. O'Reilly, 2024.

[11] A. V. Chaudhar, P. A. Charate, "Optimizing Data Lakehouse Architectures for Scalable Real-Time Analytics", *International Journal of Scientific Research in Science Engineering and Technology,* vol. 12, 2025, pp. 809-822.

[12] Apache Paimon Documentation, Bucketed | Apache Paimon, Web: https://paimon.apache.org/docs/master/append-table/bucketed.

[13] R. Sahu "Real-time Data Integration: The Evolution of CDC Architecture", *Journal of Information Systems Engineering & Management,* 2025, pp. 605-615.