Proactive concurrency control for data lakehouse: a meta-scheduling framework for urban construction data pipelines[⋆]

Olga Solovei[1,*]

[1] *Kyiv National University of Construction and Architecture, 31, Air Force Avenue, Kyiv, 03037, Ukraine*

Abstract

Transactional conflicts caused by concurrent operations on shared Delta Lake partitions pose significant challenges to reliability and predictability in modern data lakehouse environments. This paper introduces a dynamic, partition-aware meta-scheduling framework that operates above existing platform schedulers (e.g., Databricks Jobs) to proactively prevent such conflicts. The core of the framework involves analyzing SQL task semantics to infer resource access patterns, constructing a conflict graph, and applying a greedy graph coloring algorithm to produce conflict-free execution plans. Unlike static schedulers or ML-based approaches relying on historical data, the proposed method dynamically triggers scheduling decisions only when material changes occur, such as new task arrivals or task completions. Experimental results demonstrate the scalability of the algorithm, with execution time following an expected $O(n^2)$ trend based on task count, and reveal performance sensitivity to conflict density. While performance anomalies were observed at higher task volumes due to structural graph complexity, the framework remains computationally feasible for real-world engineering pipelines. Future extensions will incorporate multi-objective optimization to account for factors such as cost, deadlines, and priority in the scheduling process.

Keywords

Delta table, concurrent transaction conflict, scheduling algorithm, resource aware conflict graph

1. Introduction

The Internet of Things (IoT) has emerged as a core technology applied across all five key stages of modern construction projects: investment, planning, construction, operations, and demolition [1]. Empirical studies have statistically validated a strong and significant correlation between IoT adoption and measurable improvements in critical areas such as environmental monitoring, equipment administration, predictive maintenance, and on-site safety monitoring [2]. Despite these documented benefits, the adoption of IoT for managing complex urban construction projects remains slower than anticipated, hindered by several significant barriers [3]. Key among these are technical challenges are the demand for substantial computing power, issues with scalability, and the underperformance of technologies when deployed in real-world conditions [4].

⋆*5th International Conference on Information Technologies: Theoretical and Applied Problems (ITTAP2025), Ternopil, Ukraine, Opole, Poland, October 22-24, 2025*
[1,*] Corresponding author.
solovey.ol@knuba.edu.ua (O. Solovei)
0000-0001-8774-7243 (O. Solovei)

In this context, the Data Lakehouse architecture is well-suited to serve as a centralized platform for managing the vast amounts of structured and unstructured data generated by these systems [5-7]. A foundation of the Data Lakehouse is built on Apache Spark and cloud storage provides three fundamental advantages: 1) the decoupling of compute and storage, which enables cost efficiency and elastic scaling; 2) distributed processing, ensuring high performance on large datasets; 3) distributed storage, offering immense scalability.

However, while this architecture effectively addresses challenges of scale and performance, it introduces new operational complexities, particularly in managing transactional concurrency. In a dynamic construction environments, numerous tasks attempt to read and write to the same Delta Lake tables simultaneously. This concurrency can lead to transaction failures and cascading job retries, especially in high-velocity streaming scenarios [8].

For instance, Figure 1 illustrates a common multi-task data engineering job designed to implement a typical Extract Transform and Load (ETL) pattern. The job consists of parallel tasks to ingest IoT sensor data from different device types (Tasks 1-2), followed by parallel data validation tasks (Tasks 3-4). The validated streams are then combined in an enrichment task (Task 5), after which two final tasks are triggered concurrently: Task 6 to update a status table and Task 7 to clean up the now-processed raw data. Under a standard, resource-agnostic scheduler, both Task 6 and Task 7 would be executed in parallel. This creates a race condition and a high probability of a transactional conflict: if the DELETE operation from Task 7 modifies a table partition while Task 6 is attempting to read or update that same partition, the latter task will fail.

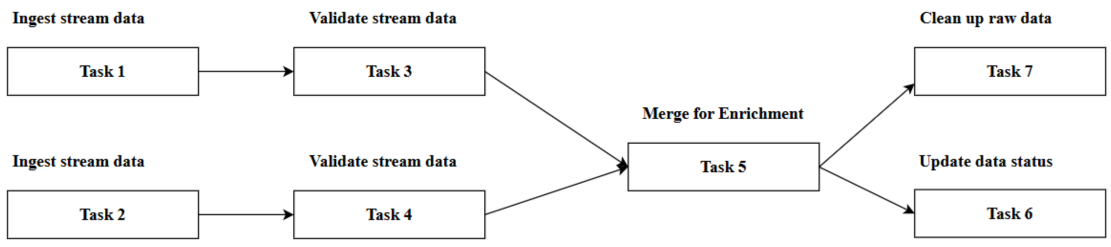


Figure 1: A single Job to implement ETL with streaming data

The root causes of this conflict lies in the interaction between parallel execution model and the optimistic concurrency control mechanism of Delta Lake. As shown in Figure 2, Spark's ability to run tasks in concurrent threads allows both Task 6 and Task 7 to proceed independently until the final commit stage. The conflict is only detected at the storage layer when one thread attempts to commit its transaction to a table version that has already been modified by the other, leading to an exception.

Existing solutions for managing concurrent transactions in Delta Lake, such as partitioning strategies and the built-in Optimistic Concurrency Control, have notable limitations. While partitioning can isolate operations and reduce contention, its effectiveness diminishes significantly in the presence of skewed workloads where many operations target the same partition [9]. Optimistic Concurrency, by design, is a reactive mechanism. It assumes conflicts are rare, allowing jobs to proceed and only checking for conflicts at the commit stage. When a conflict occurs, the failing job must be rolled back and retried, leading to wasted computational resources, increased data latency, and unpredictable job completion times [10].

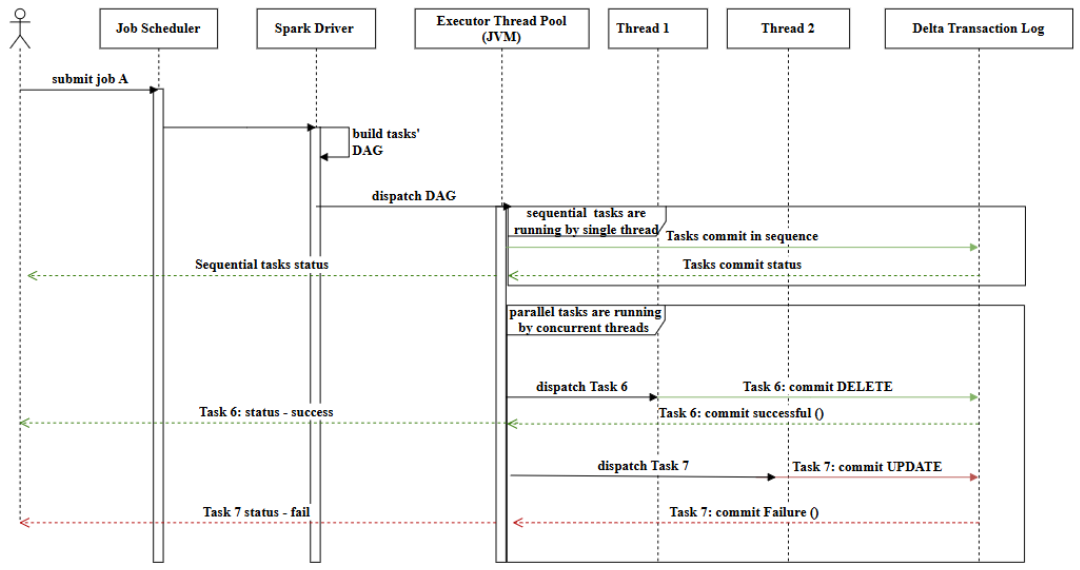


Figure 2: Parallel task execution model in Delta Lake

To address the issue of transactional conflicts, several studies have explored the integration of machine learning models with transaction schedulers. These approaches typically predict conflicts based on learned patterns from historical data and generate a fixed execution plan. However, such solutions assume a static snapshot of tasks and resources, lacking support for dynamic task arrivals. Consequently, these static methods exhibit limited adaptability to the variability and elasticity inherent in cloud computing environments.

Hence, an enhanced solution is required, involving the design of a dynamic scheduling layer positioned above the data platform. By constructing and analyzing a partition-aware conflict graph, the proposed meta-scheduler can generate and continuously adapt a conflict-free execution plan. This paradigm shifts concurrency control from a reactive, storage-level mechanism to a proactive, orchestration-level strategy, thereby mitigating the need for costly transactional rollbacks and yielding significant improvements in performance, cost efficiency, and reliability of critical data pipelines.

This paper is structured as follows: following the Introduction and Literature Review, the Materials and Methods section provides a formal definition of the proposed dynamic scheduling framework, including the scheduling algorithm and an analysis of its computational complexity. The Experiment Preparation section outlines the test scenario and procedure. The Results and Discussion section presents and interprets the practical outcomes of the experiments. Finally, the Conclusions section summarizes the key findings and specifies the next work focus.

2. Literature Review

Transactional conflicts and efficient resource scheduling are critical challenges in cloud-based data processing systems, especially under optimistic concurrency control models such as those employed by Delta Lake. Several recent studies have proposed solutions that combine machine learning and graph-based algorithms to address these challenges, each with distinct strengths and limitations.

In [11], the authors introduce a classifier-based system to proactively predict hot key access patterns for transaction scheduling. This system is trained on historical transaction traces to identify common operational footprints. The training pipeline involves: (i) encoding transactions as integer-based metadata vectors based on transaction type and known hot keys; (ii) clustering these vectors using Euclidean distance, with the optimal number of clusters determined through validation error minimization; and (iii) identifying canonical hot key access patterns for each cluster. At runtime, the system applies a K-Nearest Neighbors (KNN) classifier to assign incoming transactions to a cluster, thereby predicting their access patterns. However, the method assumes that transactions of the same type exhibit a single, predictable access pattern—a premise that does not hold in complex systems where the same transaction type may produce diverse access patterns depending on its parameters.

The “CCaaLF” model proposed in [12] (Concurrency Control as a Learnable Function) takes a different approach. It is a workload-adaptive concurrency control mechanism that predicts optimal waiting times for transactions using a machine learning model trained on historical traces. An oracle is used during training to generate conflict-free scheduling sequences. At runtime, for each transaction, a feature vector is extracted and passed through the CCaaLF model, which predicts a wait time to minimize conflicts. Similarly, [13] presents a binary classification model that estimates the likelihood of a conflict between two transactions based on past interactions. The output of this model informs the transaction scheduler, which reorders the execution queue accordingly. Despite their predictive accuracy, both [12] and [13] depend heavily on historical data and are limited in environments where workloads and data access patterns evolve rapidly, as is typical in cloud-based systems.

In [14], a graph-based approach is explored for optimizing task scheduling in a general-purpose cloud environment. The methodology involves constructing a conflict graph, where each node represents a task, and an edge denotes contention over a shared resource. A greedy graph coloring algorithm is applied to assign time slots or execution batches to tasks such that no two conflicting tasks run concurrently. While effective in reducing contention, the proposed solution assumes a static set of tasks and available resources, performing scheduling once prior to execution. This static planning fails to address the need for runtime adaptability and dynamic task arrival, which are common in cloud-native data pipelines.

In summary, while prior works have made significant progress toward proactive scheduling, they tend to rely on assumptions of static workloads and deterministic access patterns. These limitations highlight the need for a dynamic, runtime-aware scheduling mechanism capable of adapting to transactional variability and concurrency in modern cloud environments.

3. Materials and methods

To address the challenge of transactional conflicts inherent in the optimistic concurrency model of Delta Lake, this study proposes a novel scheduling framework. As illustrated in Figure 3, this framework to be placed before "Job Scheduler" (Figure 3) and has a capability to:

- Analyze the SQL queries for each task of the submitted Job to infer its resource access patterns. This includes identifying the target tables, the access mode (e.g., READ, WRITE, MERGE), and, the specific partitions being modified.
- Construct a partition-aware conflict graph based on the discovered resource access patterns.

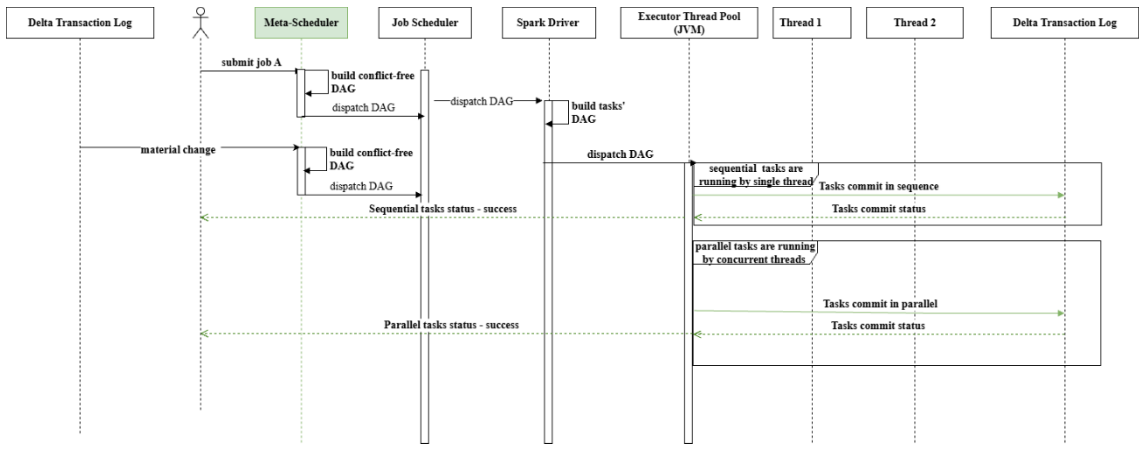


Figure 3: A novel meta-scheduling framework to dynamically address concurrent transaction conflicts

To maintain computational efficiency, the graph coloring algorithm is triggered under conditions that indicate a material change in the conflict graph. These include:

- The arrival of a new task t_{new} whose resource's access overlaps with existing running task t_i, defined as:

$$\exists t_i \epsilon T_r,\ \exists (r, m_{new}) \in (r, m_i), \text{ such that } C(m_i, m_{new})=1 \tag{1}$$

where r is a resource which is requested, m_i, m_{new} – resource access mode by running and new task.

- The completion of a task that previously introduced exclusive access constraints:

$$\Delta E \neq \emptyset \wedge \exists (r, m) \in A(t_k), m \in \{MERGE, DELETE, UPDATE\} \quad (2)$$

where $A(t_k)$ is a set of resource accesses made by the completed task t_k; ΔE represents the set of edges removed from the conflict graph G due to the task's completion; the access modes MERGE, DELETE, UPDATE are treated as exclusive and may have blocked other tasks.

The following subsections (3.1-3.3) provide a formal mathematical model and a detailed specification of the scheduling algorithm. The proposed approach is designed to be general, applying to any data platform that utilizes Delta Lake or a similar storage format employing optimistic concurrency control.

3.1. Mathematical model of tasks conflict graph

Let a workload be defined by a set of jobs $J=\{j_1,j_2,\ldots,j_m\}$. Each job is composed of a set of tasks T, such that the set of all tasks in the workload is V_t. Each task $t \in V_t$ is defined by the tuple:

$$<Id, Job\_Id, <D_t>, R_t>, \quad (3)$$

where Id is a task unique identifier, Job_Id is the identifier of the parent job, D_t is a set of the task IDs that must complete before task t can start, R_t is a set of resources the task accesses, where each resource is a tuple:

$$, R=\{<table, mode, P>\} \quad (4)$$

where table is a name of delta table, mode is the access type (e.g. MERGE, DELETE, UPDATE, READ), P is a set of partition key-value pairs defining the specific partition being accessed. An empty set P denotes a full table operation.

The workload's dependencies form a Directed Acyclic Graph (DAG) $G_D=(V_T,E_D)$ where a direct edge $(t_i, t_j) \in E_D$ exists if $t_i \in D_{\{t_j\}}$

$$(t_i, t_j) \in E_D \Leftrightarrow t_i \in D(t_j) \quad (5)$$

A transactional conflict between two tasks ($t_p \in T$, $id_q \in T$) (from any jobs) occurs if they attempt to modify the same resource simultaneously. We formally define the conflict function as:

$$C(t_p, t_q) = \begin{cases} 1, if \ (t_p.table = t_q.table) \\ \wedge \left(\{t_p.mode, t_q.mode\} \cap (MERGE, DELETE, UPDATE) \neq \emptyset\right) \\ \wedge \ (t_p.P = \emptyset \vee t_q.P = \emptyset \vee t_p.P \cap t_q.P \neq \emptyset) \\ 0, otherwise \end{cases} \quad (6)$$

This definition specifies that a conflict exists if two tasks access the same table, at least one is a write-like operation, and either one operates on the full table or their partition specifications overlap. Based on this, we construct the Task Conflict Graph as:

$$G_T=(V_T,E_T),\ (t_p,t_q)\ \in E_T \Leftrightarrow\ C(t_p,t_q)=1 \tag{7}$$

In equation (7) an edge $(t_p,t_q) \in E_T$ exists if $C(t_p,t_q)=1$

3.2. Scheduling Algorithm to prevent concurrent transaction conflict in delta tables

The objective of scheduling algorithm is to assign each task t to a discrete execution batch (time slot), denoted by the function χ(t), such that all dependencies are respected and no two conflicting tasks are assigned to the same batch. This is subject to two constraints: tasks' dependencies are respected (8) and conflicts are avoided (9):

$$\forall(t_i,t_j) \in E_D, \chi(t_j)<\chi(t_i) \tag{8}$$

$$\forall(t_i,t_j) \in E_T, \chi(t_i) \neq \chi(t_j) \tag{9}$$

The algorithm operates iteratively. At each iteration k, it identifies the set of "runnable" tasks R_k - those whose predecessors have all completed. Let C_k be the set of all tasks completed up to iteration k. The set of runnable tasks is then:

$$R_k=\{t \in V_T\{C\square_k|D_t \subseteq C_k\} \tag{10}$$

The algorithm, detailed in Algorithm 1, proceeds by building a conflict subgraph G_T^k induced by the runnable tasks R_k. It then applies a greedy graph coloring algorithm to G_T^k, which assigns a "color" c to each runnable task such that no two conflicting tasks share the same color. All tasks assigned the same color form a conflict-free execution batch $S_{\{k,c\}}$ that can be run in parallel. The union of all these color-based batches forms the set of tasks s_k to be scheduled in the current macro-step. This process repeats until all tasks have been scheduled.

Algorithm 1. Conflict free scheduler

Input: A set of tasks V_T, dependency edges E_D, conflict edges E_T.

Output: A partitioned schedule $S = \{S_0, S_1, ..., S_N\}$

Initialization: $C_0 \longleftarrow \emptyset$, $k \longleftarrow 0$

```
repeat
1. Identify runnable tasks $R_k \longleftarrow \{t \in V_T \setminus C_k \mid D_t \subseteq C_k\}$
2. Build conflict subgraph $G_T^k \longleftarrow G_T \cap (R_k \times R_k)$
3. Assign colors to resolve conflicts $\chi(t) \longleftarrow$ GreedyColoring($G_T^k$)
4. $S_k \longleftarrow \emptyset$
5. for each color c in $\chi_k$ do

6. $S_{\{k,c\}} \longleftarrow \{t \in R_k \mid \chi(t) = c\}$
```

```
7. $S_k \longleftarrow S_k \cup S_{\{k,c\}}$
8. end for
9. Update completed tasks $C_{\{k+1\}} \longleftarrow C_k \cup S_k$
10. $k \longleftarrow k + 1$ until $C_k = V_T$
```

3.3. Computational complexity of the scheduling algorithms to prevent a concurrent transaction conflict in delta tables

Let $n = |V_T|$ be the total number of tasks and $|E_D|$ be the number of dependency edges. The initial construction of the full conflict graph G_T requires a pairwise comparison of all tasks, resulting in a complexity of $O(n^2)$. The main loop runs at most n times. In each iteration k:

Identifying runnable tasks R_k can be done efficiently in $O(n + |E_D|)$. Building the induced subgraph G_T^k and applying a greedy coloring algorithm has a complexity of $O(|R_k| + |E_T^K|)$. In the worst case, $|R_k|$ can be up to n, giving a complexity of $O(n + |E_T|)$.

Therefore, the dominant step is the initial all-pairs conflict discovery, leading to an overall worst-case complexity of $O(n^2)$ for generating the complete schedule. This quadratic complexity is computationally feasible for typical data engineering workloads where n is less than hundred, making the algorithm practical for real-world implementation.

4. Experiment preparation

To validate the correctness and practical utility of the proposed meta-scheduling framework, a test scenario was designed to test the scheduler's ability to detect and resolve a delete update concurrent transaction conflict that occurs within a single, multi-tasks job.

The initial state of the workflow is defined in a job with the following Directed Acyclic Graph (DAG) of dependencies (Figure 4):

- Two parallel ingestion tasks (Ingest_air_sensor_data, Ingest_meteo_sensor_data) load raw data from IoT sensors of different types.
- These are followed by parallel verification tasks (Verify_air_sensor_data, Verify_meteo_sensor_data), which validate data according to schema.
- The verified data is then combined in a central (Aggregate_to_Enrich) task. This task writes its output to a primary data mart table for further enrichment with geo special objects.
- After the aggregation step, two tasks are designed to run in parallel: 1) Clean_up_raw_data: task performs a DELETE operation on the raw data tables that have now been processed (deletes records from a staging table, staging.raw_events, where status = 'processed'). 2) Update_Status: task reads from the same staging.raw_events table to gather metrics before updating a final status table.

•

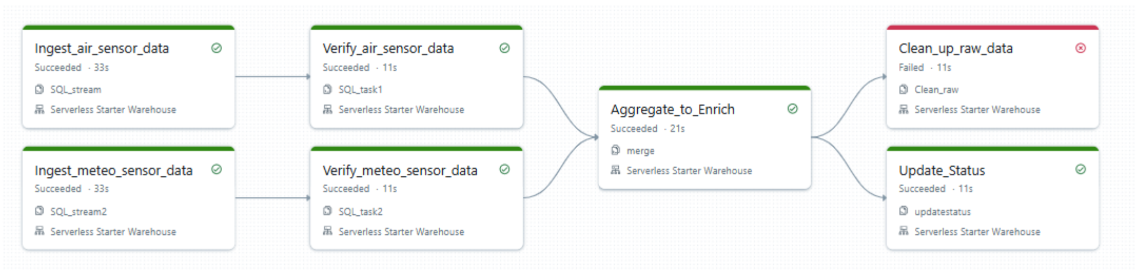


Figure 4: Extract-Transform-Load pipeline with a concurrent transaction conflict

Under the standard scheduler, both Clean_up_raw_data and Update_Status would be triggered simultaneously after Aggregate_to_Enrich completes. This creates a concurrent transaction conflict: If the DELETE operation from Clean_up_raw_data modifies the table while Update_Status is attempting to UPDATE it, a ConcurrentDeleteReadException is raised, causing the Update_Status task to fail (Figure 5).
To evaluate the computational performance and scalability of the proposed meta-scheduling algorithm, a test scenario was designed with the objectives to measure the algorithm's execution time as a function of both the total number of tasks in a workload (n) and the degree of conflicts (d).

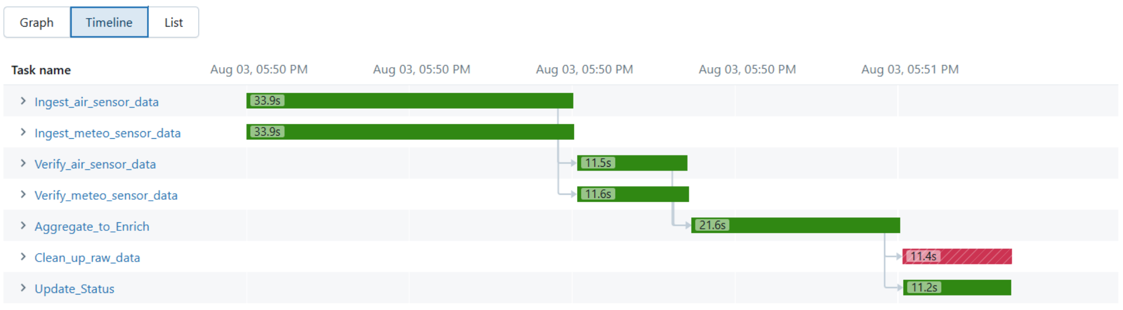


Figure 5: Concurrent Delete Exception caused by two transactions are modifying the same delta table

A synthetic workload with controlled characteristics: 1) a total number of tasks that represent the number of vertices in the conflict graph was varied across the range [50, 100, 200, ... ,1000] to simulate workloads of increasing scale. 2) Conflict Density (d) represents the probability that any two tasks in the workload have a resource conflict with. Three distinct levels of conflict density were defined for this study: Small Density: A low probability of conflict (approx. 14-20% conflicting tasks), representing a largely independent set of tasks where high parallelism is expected. Medium Density: An intermediate level of conflicts (approx. 20-28%), representing a more complex and interconnected workload. High Density: A high probability of conflict (approx. 33%), representing a "worst-case" scenario where many tasks compete for the same resources, necessitating significant serialization.

The performance evaluation was conducted executing the procedure:

```
For each value of n in the specified range:
    For each conflict density level d:
        1.Generate a synthetic workload of n tasks with the given conflict density d.
        2.Record a start timestamp.
        3.Execute the complete meta-scheduling algorithm 1.
        4.Record an end timestamp.
        5.Calculate the total Scheduler Execution Time as the difference between the end and start timestamps.
    endFor
endFor
```

Based on our theoretical analysis, we formulated two key hypotheses to be validated by this experiment:

- Hypothesis 1 (Scalability with n): the execution time of the scheduler will grow at a rate consistent with a quadratic O(n^2) complexity as the number of tasks n increases. This is due to the all-pairs comparison required for initial conflict discovery.
- Hypothesis 2 (Sensitivity to d): while execution time is primarily a function of n, it will also be influenced by the conflict density d, reflecting the varying computational cost of the graph coloring algorithm on graphs of different structures and densities.

The experiment will be executed with delta lake tables in DataBricks platform. Algorithm 1 is implemented with PySpark libraries.

5. Results and discussions

The job configuration on Figure 6 represents the output of meta-scheduling algorithm after it has analyzed the submitted job; identified that Clean_up_raw_data (a DELETE operation) and Update_Status (an Update operation) both target the same table, staging.raw_events and the same partition's value. The scheduler modifies the dependency graph - adds a conflict edge between the Clean_up_raw_data and Update_Status nodes.

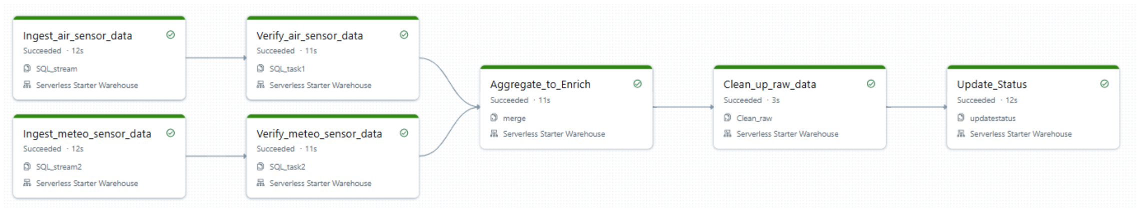
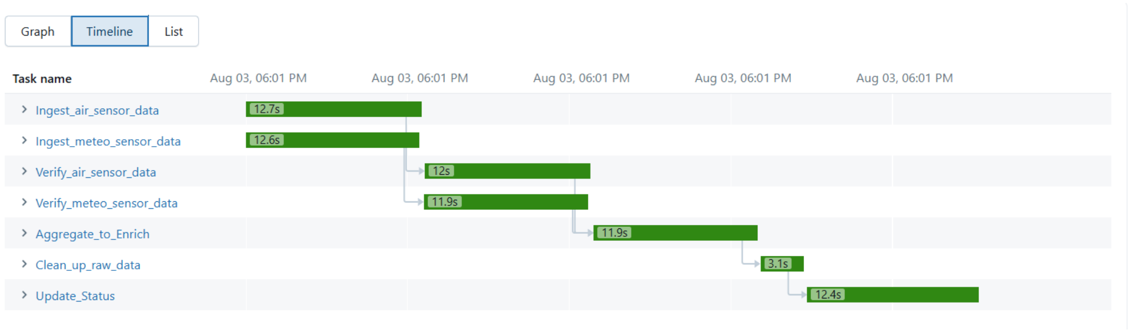


Figure 6: Extract-Transform-Load pipeline rescheduled to prevent write-read transaction conflict



Task name | Aug 03, 06:01 PM | Aug 03, 06:01 PM | Aug 03, 06:01 PM | Aug 03, 06:01 PM | Aug 03, 06:01 PM

Ingest_air_sensor_data 12.7s
Ingest_meteo_sensor_data 12.6s
Verify_air_sensor_data 12s
Verify_meteo_sensor_data 11.9s
Aggregate_to_Enrich 11.9s
Clean_up_raw_data 3.1s
Update_Status 12.4s

Figure 7: Extract-Transform-Load pipeline successfully completed

On figure 8 (a) is captured number of tasks with conflict for given level of conflict density and total number of tasks ranging from 50 to 1,000; on figure 8 (b) is illustrated the execution time it took for the algorithm to construct the conflict graph, to apply graph coloring, and generate the final conflict-free schedule.

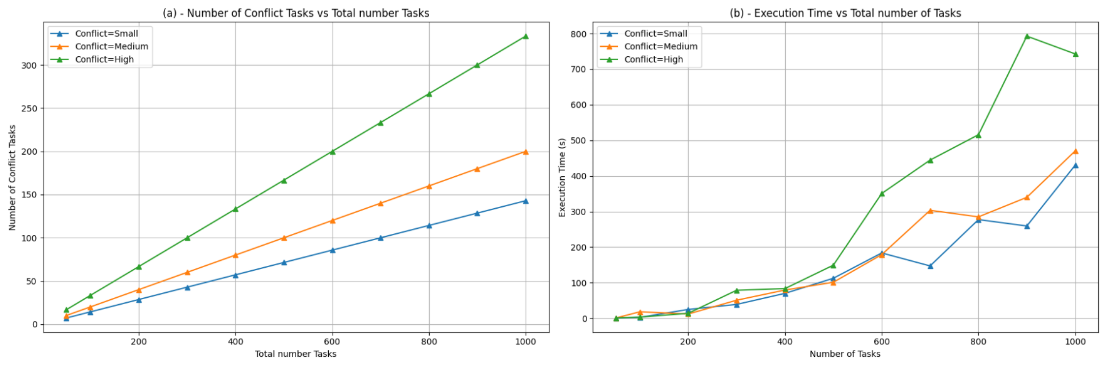


Figure 8: (a) - number of tasks with conflict for given level of conflict density and total number of tasks; (b) - the execution time of scheduling algorithm

On Figure 8 (b) is visible that as the total number of tasks (n) increases, the scheduler's execution time grows at a non-linear rate, which is consistent with the O(n^2) complexity derived from the all-pairs conflict discovery phase. For example, under a small conflict density, increasing the workload from 100 tasks to 1,000 tasks (a 10x increase) resulted in the execution time growing from approximately 3.45 seconds to 742.72 seconds (a ~215x increase), clearly demonstrating a quadratic-like trend. This confirms that the dominant computational cost of the algorithm is the initial construction of the conflict graph, as predicted by our complexity analysis.

In addition, for a fixed number of tasks, conflict density influences the execution time (Figure 9). The most visible is when the total number of tasks is 100 (n=100) and execution time is 3.45s, 18.48s, 2.75s for conflict density is small, medium, high correspondingly. 18.48ms is not expected according to the rule: Time (Small) < Time (Medium) < Time (High), so a proposed algorithm has a performance anomaly. Those outliers in collected performance statistics are identified according to IQR Method and Z-score formular:

- Execution time is 792.7s for total number of tasks is 900 and conflict density is small.
- Execution time is 742.7s for total number of tasks is 1000 and conflict density is small.

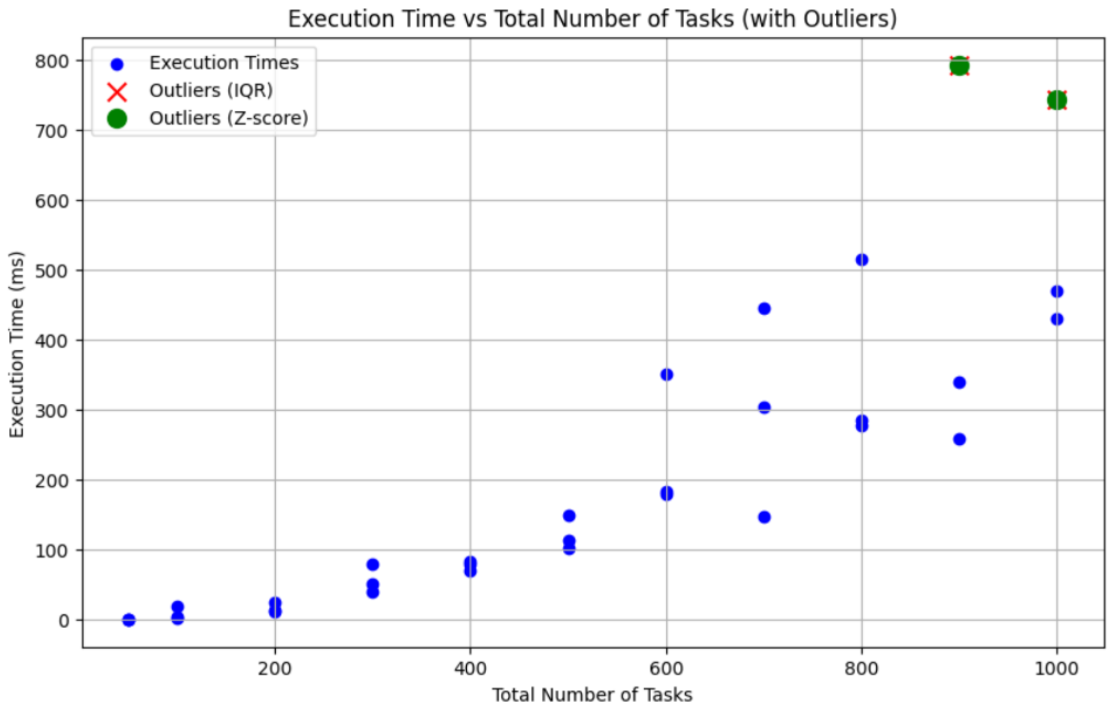


Figure 9: Scheduling algorithm's performance anomaly

The reason for outliers is the characteristics of the underlying greedy graph coloring algorithm, which is sensitive to the graph's structure. For the total number of tasks is more than 800 the algorithm explores many more color options for each vertex this leads to more cache misses and less predictable branching in the algorithm's execution path, resulting in longer runtimes. As including more than 800 tasks running simultaneously is a rare scenario, for engineering pipeline we conclude that the proposed algorithm demonstrates a predictable O(n^2) scaling, confirming its feasibility for real-world application.

6. Conclusion

This research set out to develop and evaluate a proactive, partition-aware meta-scheduling framework for dynamic preventing transactional conflicts in modern data lakehouse environments. A proposed scheduling framework constructs a partition-aware conflict graph based on the discovered resource access patterns. It triggers graph coloring algorithm under conditions that indicate a material change only. The primary objectives were to design a computationally feasible algorithm and to validate its performance scalability.

The experimental results presented in this paper have successfully verified our initial hypotheses. Firstly, the scheduler's execution time was shown to scale quadratically with the number of tasks, consistent with the theoretical $O(n^2)$ complexity of the conflict discovery phase (Hypothesis 1). Secondly, the results confirmed that the algorithm's performance is sensitive to the structural complexity of the conflict graph, which is influenced by the conflict density (Hypothesis 2). These findings validate the proposed algorithm as a practical and effective solution.

The current research limitation is that a proposed meta-scheduler creates a valid, conflict-free schedule but does not consider other factors like cost, deadlines, or job priority. The "greedy" coloring algorithm produces a valid schedule, but not necessarily the optimal one from a business perspective.

Further work focus will be to incorporate multi-objective optimization into the execution planner. This involves extending the conflict graph into a weighted, directed acyclic graph where vertices are weighted by job priority, estimated compute cost, and expected runtime.

The scheduling algorithm would then move beyond simple graph coloring to employ more advanced techniques from operations research, such as critical path analysis or list scheduling algorithms, to generate a schedule that not only avoids conflicts but also aims to minimize total workflow cost.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] F. Althoey, A. Waqar, SH. Alsulamy, AM. Khan, A. Alshehri, Il Falqi, M. Abuhussain, “Influence of IoT implementation on Resource management in construction,” *Heliyon*, vol. 10, no. 15, August 2024. doi: 10.1016/j.heliyon.2024.e32193.

[2] A. Khan, K. Alrasheed, A. Waqar, H. Almujibah, O. Benjeddou, “Internet of things (IoT) for safety and efficiency in construction building site operations,” *Scientific reports*, vol 14, no.2, 2024. doi: 10.1038/s41598-024-78931-0.

[3] D. Yu, Q. Tao, Q. Liu, Y. Jin, Y. Sun, P. Fu, “Lifecycle management of urban renewal enabled by Internet of Things: Development, application, and challenges,” *Results in Engineering*, 2025. doi: 10.1016/j.rineng.2025.105706.

[4] W. Zonghui, K. Veniaminovna, V. Vladimirovna, K. Ivan, H. Isleem, “Sustainability in construction economics as a barrier to cloud computing adoption in small-scale Building projects,” *Scientific Reports*, vol. 15, no.1, 2025. doi:10.1038/s41598-025-93973-8

[5] C. Rucco, A. Longo, M. Saad, “Efficient Data Ingestion in Cloud-based architecture: a Data Engineering Design Pattern Proposal”, 2025. *arXiv preprint arXiv:2503.16079.*

[6] I. Hassan, “Storage structures in the era of big data: from data warehouse to lakehouse,” *Journal of Theoretical and Applied Information Technology*, vol. *102, no.* 6, pp. 2428-2441, March 2024.

[7] K. Gade, “Data Lakehouses: Combining the Best of Data Lakes and Data Warehouses”. *Journal of Computational Innovation*,” vol. *2,* no.1, 2022.

[8] Isolation levels and write conflicts on Databricks. Available at: https://docs.databricks.com/aws/en/optimizations/isolation-level.

[9] PJ Liu, CP Li, H Chen, "Enhancing storage efficiency and performance: A survey of data partitioning techniques." Journal of Computer Science and Technology 39, no. 2, 2024 , pp. 346-368.

[10] Concurrency control - Delta Lake Documentation. Available at: https://docs.delta.io/latest/concurrency-control.html.

[11] A Cheng, A Kabcenell, J Chan, X Shi, P Bailis, N. Crooks, and I. Stoica, "Towards optimal transaction scheduling."Proceedings of the VLDB Endowment 17, no. 11 (2024): 2694-2707. doi:10.14778/3681954.3681956.

[12] H Pan, S Cai, TTA Dinh, Y Wu, YM Chee, G Chen, BC Ooi, “CCaaLF: Concurrency Control as a Learnable Function”. *arXiv preprint arXiv:2503.10036*, 2025.

[13] S Chen, C Shen, C Wu, “Intelligent Transaction Scheduling to Enhance Concurrency in High-Contention Workloads”. *Applied Sciences*, vol. 15, no. 11, 2025, 6341; doi:10.3390/app15116341.

[14] S. De An efficient technique of resource scheduling in cloud using graph coloring algorithm. *Global Transitions Proceedings*, vol. 3, no. 1, pp. 169-176, 2022.
doi: 10.1016/j.gltp.2022.03.005.