Rethink Query Optimization in HTAP Databases

HAOZE SONG[*], The University of Hong Kong, Hong Kong SAR
WENCHAO ZHOU, Alibaba Group, China
FEIFEI LI, Alibaba Group, China
XIANG PENG, Alibaba Group, China
HEMING CUI, The University of Hong Kong, Hong Kong SAR

The advent of data-intensive applications has fueled the evolution of hybrid transactional and analytical processing (HTAP). To support mixed workloads, distributed HTAP databases typically maintain two data copies that are specially tailored for data freshness and performance isolation. In particular, a copy in a row-oriented format is well-suited for OLTP workloads, and a second copy in a column-oriented format is optimized for OLAP workloads. Such a hybrid design opens up a new design space for query optimization: plans can be optimized over different data formats and can be executed over isolated resources, which we term ***hybrid plans***. In this paper, we demonstrate that *hybrid plans* can largely benefit query execution (e.g., up to 11× speedups in our evaluation). However, we also found these benefits will be potentially at the cost of sacrificing data freshness or performance isolation since traditional optimizers may not precisely model and schedule the execution of hybrid plans on real-time updated HTAP databases. Therefore, we propose **Metis**, an HTAP-aware optimizer. We show, both theoretically and experimentally, that using the proposed optimizations, a system can largely benefit from hybrid plans while preserving isolated performance for OLTP and OLAP, and these optimizations are robust to the changes in workloads.

CCS Concepts: • **Information systems** → **Data access methods**; **Query optimization**; **Data layout**; • **Computer systems organization** → **Real-time system architecture**.

Additional Key Words and Phrases: Hybrid Transactional and Analytical Processing (HTAP) Databases, Adaptive Query Plan, Mixed Workloads

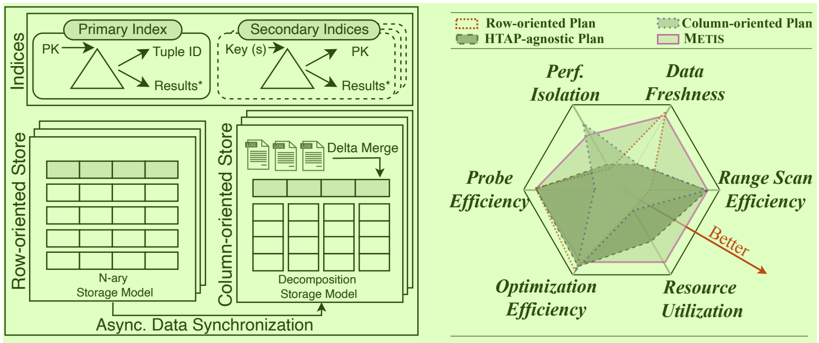
ACM Reference Format:
Haoze Song, Wenchao Zhou, Feifei Li, Xiang Peng, and Heming Cui. 2023. Rethink Query Optimization in HTAP Databases. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 256 (December 2023), 27 pages. https://doi.org/10.1145/3626750

1 INTRODUCTION

Today, data-intensive applications often utilize vast amounts of data for diverse real-time business tasks (e.g., data-driven decisions [4, 13, 17, 26]), necessitating weaving analytical and transactional processing techniques together [45]. In response, many recent academic and industrial efforts have been devoted to developing hybrid transactional and analytical processing (HTAP) systems [2, 16, 31, 33, 35, 41–43, 49, 51, 55–57, 61, 62, 68], which are expected to provide ❶ prompt analysis of

[*]Work performed during an internship at Alibaba Group.

Authors' addresses: Haoze Song, hzsong@cs.hku.hk, The University of Hong Kong, Pok Fu Lam, Hong Kong SAR; Wenchao Zhou, zwc231487@alibaba-inc.com, Alibaba Group, 969 West Wen Yi Road, Yu Hang District, Hangzhou, Zhejiang, China; Feifei Li, lifeifei@alibaba-inc.com, Alibaba Group, 969 West Wen Yi Road, Yu Hang District, Hangzhou, Zhejiang, China; Xiang Peng, pengxiang.px@alibaba-inc.com, Alibaba Group, 969 West Wen Yi Road, Yu Hang District, Hangzhou, Zhejiang, China; Heming Cui, heming@cs.hku.hk, The University of Hong Kong, Pok Fu Lam, Hong Kong SAR.



(a) Hybrid Physical Layout. (b) Performance Comparison.

Fig. 1. (a) shows an example of the hybrid physical layout in modern HTAP systems (e.g., SQL Server [28], TiDB [33]): the row-oriented tables are well-suited for updates and probes; a second copy in a column-oriented layout is optimized for range scan. Leveraging a hybrid physical layout, Metis strikes a practical balance between performance, isolation, and freshness for HTAP (see (b)).

fresh data and ❷ isolate the performance of interleaved workloads.

A practical HTAP database generally consists of an online transactional processing (OLTP) engine that supports high throughput transaction processing, and an online analytical processing (OLAP) engine supports analytics with low latency. To handle mixed workloads efficiently, a popular category of distributed HTAP databases (e.g., SQL Server [42], TiDB [33], ByteHTAP [16], PolarDB-IMCI [67], Oracle Dual [41], and AlloyDB [31]) typically employs the two engines with specialized data stores and asynchronously replicated data from one copy to the other, achieving both ❶ and ❷.

An example is shown in Figure 1a: a row-oriented store (for short, row store) that stores data in rows is optimized for operating on a single data tuple at a time and accessing many attributes, favor for OLTP; a column-oriented store (for short, column store) that stores the same attributes of different rows contiguously in columns is optimized for accessing a massive number of rows at a time with a subset of tuple attributes, favor for OLAP. To provide swift OLAP capabilities on new data, updates are asynchronously replicated from the row store into the column store while posing minimal impact on OLTP. We further formulate the system model in §2.1.

Given such a design, OLTP and OLAP workloads can be independently processed on their desirable stores, thus naively providing isolations between OLTP and OLAP in the storage layer.

Unfortunately, restricting each workload to its specialized store leaves much of the performance potential unrealized. This is because, for read-only queries, both the row and column store can significantly outperform one another based on the characteristics of system implementations and workloads [1, 28, 39] (see our experimental results in §2.2). Thus, there may be queries for which neither the column store nor the row store is optimal.

To reach the full potential of the hybrid physical layouts, several HTAP systems [28, 33] have integrated the two stores as alternative data access methods in their query optimizers to generate ***hybrid plans*** for queries. Specifically, a hybrid plan allows a single query to retrieve data from both the row and column stores simultaneously and calculate the query results based on a consistent data view.

Motivation. Nevertheless, existing approaches [28, 33] select access paths and do query optimizations simply based on queries' selectivity [39], neglecting the ***data dynamicity*** of HTAP databases. In §3, we show that blindly pursuing hybrid plans can easily make the generated plans sub-optimal and damage the two important properties: data freshness (❶) and performance isolation (❷).

1 We regard the two properties essential because high data freshness [33, 45, 50, 56] provides users with intelligent insights into the fresh data at generation speed (i.e., OLAP queries can always work on the new data generated by OLTP), which distinguishes HTAP databases from traditional Extract-Transform-Load (ETL) method [65]. Performance isolation [33, 50, 61] in HTAP refers to the ability to maintain the performance of both OLTP and OLAP workload while another workload increases, which is important for providing individual service-level agreements (SLAs).

2 Therefore, in this paper, we take the first step to systematically study the benefits, side-effects, and solutions of hybrid plans given our target HTAP system model (§2.1). Our key insight is that, to keep hybrid plans efficient, we should put *data dynamicity* into the design of the query optimizer by capturing the mutual relationship between reads (i.e., read-only queries) and writes (i.e., write transactions). In our paper, *data dynamicity* is defined as the nature in that the OLTP engine continuously executes users' write transactions and incrementally applies new data from the row store into the column store. Based on our insight, we identify three key challenges.

3 The first challenge is *how to precisely model the cost of data access paths when new writes continuously update the replicated data copy for reads?* As shown in Figure 1a, distributed HTAP databases typically support timely updates in the read-optimized column store (i.e., the data copy for reads) through a separate delta store. Delta store accumulates updates continuously and periodically merges them into the columnar storage (see Figure 1a, detailed in §2.1). This architecture makes the traditional cost model imprecise for evaluating the cost of data access paths: there is no fixed selectivity threshold for access path selection; rather, the division depends on the workload's dynamicity (i.e., the concurrency of writes).

4 To address this challenge, we propose a new delta-store-aware cost model incorporating the data dynamicity into the optimizer: Demain (**De**lta-**main** model). Demain captures the performance of select operators in both delta stores and column stores and thus can efficiently guide the access path selection.

5 The second challenge is *how to optimize data freshness and execution time together, especially when new writes are propagated asynchronously?* Generally, in an HTAP database that uses asynchronously replicated data copies, optimizing execution time can be at the cost of data freshness. This is because, due to data replication, the visibility of new writes in the column store is always delayed. Hence, even if the column store may outperform the row store (i.e., the data copy for writes) on the sequential scan, the execution must be blocked until the new writes are fully synchronized to the column store, leading to a longer response time.

6 We regard considering visibility delay in query optimizations as an important research problem because, even though multiple existing works [33, 56, 61] strive to minimize the visibility delay between the row store and column store from system scopes, depending on the deployment, the visibility delay is still pronounced (e.g., 10*s* delay in DB2 IDAA [12], 8*mins* delay in production at Google [68], 606*ms* delay in experiments at ByteDance [16]).

7 For this challenge, we propose a new visibility-aware plan selection algorithm. It firstly estimates the visibility delay between the row store and column store based on the ongoing and predicted workload characteristics. When optimizing queries, it advances the query performance by pre-executing plans on the available data, thus masking the notorious visibility delays.

8 The final challenge is *how to ensure isolated performance between reads and writes when query plans are hybrid?* A strawman approach uses a pre-defined quota for the reads in row stores (i.e., the data copy for writes). For example, TiDB limits the default access table size on its row store for the OLAP workload to at most 500 *MB* [33]. However, manual intervention cannot effectively utilize resources while reducing query latency. A configuration that works well for one workload is unlikely to work well for another.

9 We develop a new resource-aware query re-optimization approach for hybrid plans. Instead of

scheduling resources [56] or limiting resource usage [33, 42, 49], our re-optimization approach can automatically adapt to the workload shift. In our approach, when a high resource contention is detected, it re-optimizes the plans by opportunistically combining efficient sub-plans of previously-optimized plans into a good new plan, which can alleviate the resource contention without a whole plan re-optimization.

System Integration. We combine all these new optimization techniques (i.e., delta-aware cost model, visibility-aware plan selection, and resource-aware query re-optimization) into our prototype: METIS[1], an HTAP-aware plan optimizer. METIS is developed based on METISDB. We detail our storage model of METISDB in §4.1.

Overall, METIS captures the *data dynamicity* to keep hybrid plans efficient. METIS pushes the boundary of traditional optimizers' design space by adding data freshness and performance isolation as new design goals. Figure 1b compares METIS, row-, column-oriented, and HTAP-agnostic plans. Among them, METIS achieves a practical point in the design space and can speed up analytical queries without sacrificing freshness and performance isolation.

Contributions. To the best of our knowledge, this paper provides the first treatment of efficiently accommodating hybrid plans in distributed HTAP databases. Our contributions are four-fold:

- We systematically analyze hybrid plans given a popular category of HTAP databases using multiple specialized data copies.
- We develop METIS, including a new cost model (Demain), a new visibility-aware plan selection algorithm, and a new plan re-optimization approach to ensure performance isolation.
- We extensively evaluate METIS using CH-benCHmark [19], TPC-DS [23], and YCSB [24]. The evaluation results demonstrate the effectiveness of METIS: it can generate efficient hybrid plans, and the plans are robust to the change of workloads.
- Our implementation (i.e., METIS) can be a practical template for the future adoption of HTAP-aware query optimizations in those HTAP databases that have the same system model as METISDB.

The rest of the paper is organized as follows. §2 discusses the system model of our target HTAP databases and the motivation for hybrid plans. §3 details the problems of HTAP-agnostic hybrid plans. §4 provides an overview of METIS. §5 discusses our new cost model. §6 presents our visibility-aware algorithm and re-optimization approach. §7 evaluates the performance of METIS. Finally, a discussion of related works is available in §8, and §9 concludes the paper.

2 BACKGROUND AND MOTIVATION

In this section, we first provide a system model of our target HTAP databases and show the motivation for hybrid plans. For specific storage implementations of METISDB, we refer readers to §4.1. For discussions on the related works, we refer readers to §8.

2.1 Hybrid Data Format in HTAP

Following the philosophy of *"one size doesn't fit all"* [53], a popular category of distributed HTAP systems utilizes hybrid physical designs to handle complicated HTAP workloads efficiently (e.g., SQL Server [42], TiDB [33], SAP HANA ATR [43], Oracle Dual [41], VEGITO [61], Janus [7], UniStore [35], L-store [57], IBM DB2 IDAA [12], PolarDB-IMCI [67], F1 Lightning [68], and AlloyDB [31]).

We summarize the common designs of these systems (including our METISDB) and distill their common grounds into an abstract system model, which outlines the applicable scope of our high-level optimization principles (i.e., delta-store-aware cost model, visibility-aware plan selection,

[1]METIS was the Titan goddess of good counsel, planning, and wisdom, which signifies our optimizations for generating an efficient execution plan.

and resource-aware plan re-optimizations). As such, MetisDB is one of the state-of-the-art HTAP databases belonging to this category, and we provide Metis as a concrete instance for applying the HTAP-aware optimization principles to MetisDB (§4). We reuse Figure 1a as a reference and illustrate our system model as follows:

- **[Row store]** The system has a row store that contiguously stores all attributes of each data tuple. All write transactions are handled by the row store. Each write transaction will generate a new data version in the row store, and each data version is tagged with a monotonically increasing timestamp.
- **[Indices]** Indices are built over the row store to speed up writes and point lookups. When querying on an indexed column, indices can provide results in sorted order. We do not consider indices over column store since it incurs additional overhead and complexity on real-time updates [40].
- **[Column store]** The system has a column store using the decomposition storage model. It groups the same attributes of different rows together and stores them continuously. The column store supports fast sequential scans on uncompressed data.
- **[Delta store]** The system has a delta store that absorbs timestamped updates efficiently. The delta store is write-optimized and periodically merges the absorbed updates into the column store. Meanwhile, the delta store supports multi-version concurrency control. When performing a column-oriented scan, queries first retrieve fresh data from the delta store and combine them with the results from the column store to generate a fresh view [33, 42, 45].
- **[Data synchronization]** The system has a data transfer pipeline that asynchronously transfers updates from row store to column store, and the synchronization is off the critical path of transaction processing. Column stores can fall behind, and the row store will never fall into pushback.

We then explain the rationales behind the HTAP-specific but commonly-used designs and demonstrate how they are general to the distributed HTAP databases:

- **[Delta Store]** As the column store is extremely read-optimized for column-wise reads, a write-optimized delta store (for row-wise writes) is necessary to keep the column store abreast with the row store (❶). Otherwise, the column store can fall arbitrarily behind due to the gap in write efficiency, leading to bad data freshness; additionally, due to data synchronization threads (i.e., the threads for writing new data) and OLAP threads (i.e., the threads for processing queries) will write and read the delta store simultaneously, multi-version concurrency control (MVCC) is commonly used to resolve conflicts between read and write operations by maintaining multiple snapshots. To the best of our knowledge, delta stores are widely used in existing HTAP systems, e.g., the delta storage in SQL Server [42] and ByteHTAP [16], deltaTree in TiDB [33], and transaction maps in Oracle Dual [41].
- **[Asynchronous Replication]** Recall another important property of HTAP databases is performance isolation (❷). In real workloads, OLTP is usually mission-critical and sensitive to performance. To minimize the negative impact of data replication on transactions, asynchronous replication is widely adopted, which allows ongoing transactions to be committed before the data synchronization of these transactions is finished. For example, the transactional replication in SQL Server [42], Redo Logs replication in ByteHTAP [16], and Raft Learner replication in TiDB [33].

2.2 Motivation of Hybrid Plans

We now experimentally motivate hybrid plans and show practical scenarios where hybrid plans take effect. We show the potential of hybrid plans using HTAP-agnostic hybrid plans (i.e., the state-of-the-art approach adopted by existing HTAP databases [33, 41]) on MetisDB without involving real-

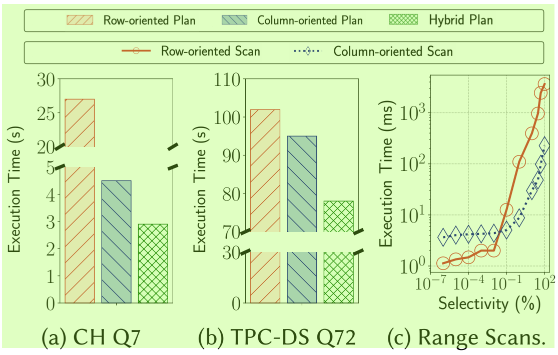


Fig. 2. Motivation of using hybrid plans in an HTAP database (see 2a and 2b). 2c shows the performance of row-oriented scans and column-oriented scans with varying selectivity.

time updates. Specifically, HTAP-agnostic hybrid plans are generated by adding column-oriented scans as an alternative access path into the cost model of row stores without considering the *data dynamicity* of HTAP databases (§7.1). When putting the plans into the HTAP context, such plans can lead to sub-optimal performance, which we study in §3.

We did the experiments on two well-studied benchmarks: CH-benCHmark [19] and TPC-DS [23]. Both of them contain multiple queries with wide variations in complexity and range of scanned data. We calculated the speedups for each query by comparing the execution time of the hybrid plan to the faster one of the row- and column-oriented plans. Detailed experiment configurations for hardware, software, and workloads are shown in §7.

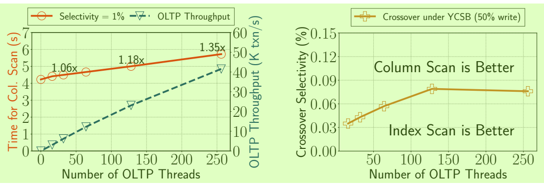
Our evaluation results show that neither the row-oriented nor column-oriented plans could be optimal for a number of given queries. In CH-benCHmark, nine queries (i.e., 40.9%), out of twenty-two, benefit from hybrid plans and achieve 1.68× speedups in geometric mean; in TPC-DS, seventy-seven queries (i.e., 77.8%), out of ninety-nine, benefit from hybrid plans and achieve 3.06× speedups; among the queries, TPC-DS Q72 achieved 11× speedups, in which indices on fact tables are used. We show two representative queries from each workload in Figure 2a and Figure 2b.

Based on our experiments, we summarize three factors that motivate the desirability of hybrid plans. The first factor is the diversity of data access patterns inside a single query. When a query joins multiple tables, the variance of data size and query selectivity[2] on each table motivates using different access paths for different tables. The performance comparison of the row- and column-oriented scan with different selectivity is shown in Figure 2c.

The second factor is the division of data schema. Star schemas [30, 54] and snowflake schemas [44, 63], as two successful data models, provide a clear division between dimension tables and fact tables, where dimension tables are most likely to join fact tables with their primary keys. Therefore, when using such schemas in HTAP databases, retrieving data from row stores (with the primary indices) for dimension tables and retrieving data from column stores for fact tables can be a competitive candidate for the optimal plan.

The third factor is the queries' requirement for physical properties (e.g., sort order). When a query requires specific properties on a portion of tables, either demand by user requests (e.g., "order by" in SQL) or demand by an inside operator (e.g., sort-merge join requires ordered data), specific stores (e.g., a B+ tree index that delivers sorted data) have the opportunities to outperform the others on these tables, leading to hybrid plans.

[2]Same as previous papers [40, 60], we say a predicate (filter) is selective (or its selectivity is low) when the result set has few qualifying tuples. A selectivity that ranges from 0% to 100% signifies the percentage of qualifying tuples in the database.



(a) Impact of OLTP on Scan. (b) Impact on Crossover Sel.

Fig. 3. Impact of Delta Store (Read Amplifications).

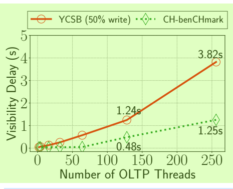


Fig. 4. Freshness Loss

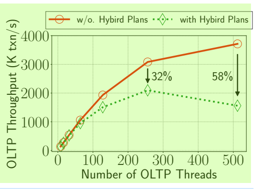


Fig. 5. Throughput Drop

3 Problems of HTAP-agnostic Hybrid Plans

In this section, we study the performance issues caused by *data dynamicity*. We consider the mutual relationship between read-only queries and write transactions when using HTAP-agnostic hybrid plans (i.e., the approach we used in §2.2). In particular, we study the impact of writes on reads in §3.1 and §3.2, and in turn, the impact of reads on writes in §3.3.

3.1 Impact of Data Synchronization

According to our system model (§2.1), when processing an HTAP workload, new data is continuously generated by write transactions on the row store and replicated to the column store through a stand-by delta store. Thus, to conduct a column-oriented scan, the OLAP engine should always combine the new data from the delta store to provide a fresh data view (e.g., using the k-way merge algorithm [59]). Thus, it costs additional overhead for retrieving data, leading to read amplification on column-oriented scans and thus influencing the decisions on access path selection.

Depending on specific policies adopted by the delta store, the severity of read amplification may differ. For instance, database administrators can set a rigid boundary for the size of delta stores and enforce delta merge operations immediately when the size of delta stores exceeds the limitations. Thus, the effect of read amplification on column-oriented scans can be bounded. However, such an approach must block writes (a.k.a write stalls [48, 69]) when the write workloads are heavy, and the speed of delta merge may not catch up with the new writes. Note that, even in such a case, a new cost model that captures the effect of delta store is still desirable.

Instead of imposing such a rigid boundary, we explore the impact of data synchronization from the optimizer's view and do not manually configure the size of our delta store. Thus, the size of our delta store can continuously grow with the new appended updates. We assume the database engines will later merge delta into the column store periodically in the background. As such, for a real-time workload, we consider OLTP write concurrency as the major factor contributing to the overhead of data synchronization.

In our experiments, we executed the YCSB [24] workload on MetisDB to fine-tune its OLTP concurrency and the percentage of read/write ratio. For OLAP, we used a plain query (i.e., the $Q2$ in §7.1). Precisely, the OLAP query scans a table in the database with a predictor to control the selectivity. We run OLTP workloads for 10 minutes to warm up. Figure 3a shows the results: the execution time for column scans increased proportionally with the concurrency of OLTP workloads.

We then study the influences on access path selection. Figure 3b shows the results. Same as in previous papers [28, 40], we define selectivity crossover as the selectivity that the row-oriented scans and column-oriented scans have an identical execution time. Thus, the column-oriented scans should be optimal for queries with higher selectivity than the crossover. The results show that as the concurrency grows, the crossover point rises to higher selectivity at the beginning and plateaus eventually when the throughput of data synchronization is close to saturation.

Takeaways. Retrieving data from delta stores causes additional overhead to column scan, which is critical to access path selection.

3.2 Impact on Data Freshness

Ensuring high data freshness (❶) is one of the most important design goals of real-time HTAP databases [45, 50, 56]. Generally, in HTAP, row stores have better data freshness than column stores as all data are generated on row stores and then propagated into column stores asynchronously. As suggested by previous papers [33, 45, 68], such a freshness loss can not be ignored (§1).

We studied the visibility delay, defined as the time delay between an update committed on the row store and when queries using column-oriented scans can read that update. We report the 99.9th visibility delay (in ten seconds) of MetisDB. The results are shown in Figure 4. Overall, the visibility delay increased (i.e., positively correlated) with the concurrency of OLTP workloads due to the overhead of processing more data.

Our observation is that, from the optimizer's perspective, queries can always take the best data freshness by either directly executing queries on the row store or blocking the query execution until all new data in the delta store becomes visible. It introduces a dilemma: optimizing the time for retrieving data by using column-oriented scans may lead to negative optimization on response latency due to the blocking time. This scenario could happen even if the cost model for access path selection is completely accurate.

Therefore, we note that there is a new opportunity for hybrid plans: an optimizer can opt to pre-execute a portion of sub-plans on the row store while optimizing the execution time for the rest of the plans on desirable data sources. By doing so, the query can start to be processed before all new data is available at the delta store, masking the blocking time (caused by the visibility delay) with the pre-scheduled execution time on the row store. We detail our visibility-aware plan selection algorithm in §6.1.

3.3 Impact on Performance Isolation

Another essential property of HTAP databases is performance isolation (❷). Without hybrid plans, operations for retrieving data in OLTP and OLAP workload are handled by separate stores. In this case, database administrators can assign hardware resources (e.g., CPU cores) to each store and execution engine with resource management tools (e.g., Cgroup [38]) or deploy stores across machines to provide isolation by nature, along with independent scalability for OLTP and OLAP. In MetisDB, we deployed the row store and column store on different machines along with execution engines (§7.1), thus providing independent CPU, memory, and disk resource.

However, when using hybrid plans, such isolation is broken. Issues are two-fold. The first is the performance drop in OLTP. Figure 5 shows the OLTP throughput loss of the HTAP databases on the CH-benChmark workload (see §7 for detailed configurations). When the workload for OLTP is light (i.e., less than 256 OLTP threads), hybrid plans have little effect on OLTP throughput. Things change when loads of row stores are close to being saturated: hybrid plans impose an OLTP throughput drop (up to 58%) due to the competition for both physical resources and logical resources (e.g., latches in internal data structures). The second issue is the inefficiency of hybrid plans. Under high contention, hybrid plans that operate row-oriented scans must wait for the schedule, causing their performance to fall short of expectations.

These two new issues are specific to HTAP databases and may not be pronounced in traditional data warehouses that only perform read-only queries since they do not target to serve OLTP workloads and do not provide performance isolation between OLTP and OLAP.

Desiderata	Roadmaps of Metis	Design Knobs	Proposed Solution
Low Execution Time	Hybrid Plans	Cost Model	Demain Model
High Data Freshness	Masking the Cost of Visibility Delay	Plan Selection Algo.	Visibility-aware Optimizations
Strong Performance Isolation	Restricting the Abuse of Hybrid Plans	Re-optimization Algo.	Proactive Re-optimization with Plan Stitch

Table 1. Desideratas, roadmaps, design knobs, and solutions of Metis for efficient query optimization in HTAP Databases.

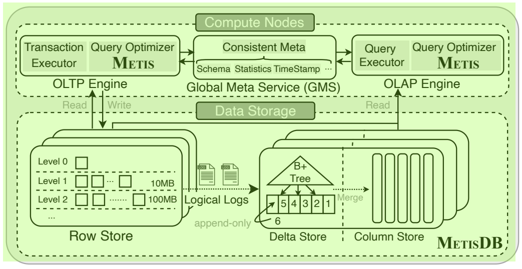


Fig. 6. An overview of METISDB.

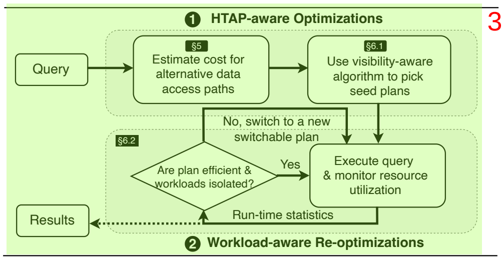

1 HTAP-aware Optimizations
Query
§5
Estimate cost for alternative data access paths
§6.1
Use visibility-aware algorithm to pick seed plans
No, switch to a new switchable plan
§6.2
Are plan efficient & workloads isolated?
Yes
Execute query & monitor resource utilization
Results
Run-time statistics
2 Workload-aware Re-optimizations

Fig. 7. METIS's optimization workflow.

4 METIS Overview

This section presents an overview of METIS, a prototype of our HTAP-aware query optimizer. METIS is developped on METISDB. To begin with, we first introduce the detailed architecture and storage model of METISDB (as a specific instance of the system model in §2.1) and then present the design of METIS. As shown in Table 1, METIS responds to each desideratum of HTAP-aware optimizations by redesigning several critical design knobs of the optimizers.

4.1 METISDB Overview

As shown in Figure 6, METISDB has a shared-nothing architecture in which compute and storage is decoupled. In the storage tier, we use a variation of RocksDB as a row store and a variation of ClickHouse as a column store. METISDB supports using in-storage range filters, and we do not consider other computation pushdowns.

The row store is partitioned into multiple shards and deployed across multiple machines. Each shard is replicated into the delta store asynchronously using logical logs. The column store is co-located with the delta store and uses the same partition policy as the row store. The new data in the delta store will be periodically merged into the column store in batch. We assume the network between the row store and the delta store is reliable. Every new update is eventually delivered and processed.

METISDB provides snapshot isolation for both OLTP transactions and OLAP queries. To do so, both row and delta store supports multi-version concurrency control (MVCC). New data tuples are streamingly inserted into the delta store with unique row IDs and the commit timestamps of the affiliated transactions.

For the compute node, we have three key components. OLTP and OLAP engines are the system's entry point and are stateless, which includes modules such as optimizer and executor. Engines are responsible for distributed data routing, concurrency control, and secondary index maintenance. Caches for OLTP and OLAP are managed independently. Besides the engines, we have a global meta service, which maintains the globally consistent information about Tables, Schema, Statistics, and other Metadata of the system. It also takes responsibility for providing global timing service. As such, each transaction and query will be started by taking a unique and monotonical timestamp to obtain a globally consistent snapshot. Additional timestamps are used for transaction commit.

Storage Model. The row store of METISDB is implemented over log-structured merge trees (for short, LSM-tree). Specifically, each row in the row store is stored as a key-value pair, using its Table ID and Row ID as a key and storing all of the attributes (columns) contiguously as a value. When performing a table scan, the storage engine retrieves all key-value pairs with the same table ID as the given table. We can build both primary and secondary indices to speed up the look-up performance on the row store. All indices are also implemented as key-value pairs. For instance, an entry of a primary index stores its primary key as a key, and the value is the corresponding row ID of the indexed row.

Data in the column store is stored by columns and in the form of arrays (i.e., vectors or chunks of columns) during the execution. It supports timely updates by a columnar delta tree that organizes an append-only delta store with a *B*+ tree index to locate updates efficiently. When performing a column scan, the storage engine retrieves data from both the delta store and the stable column chunks and then merges the results for output.

4.2 Workflow and Key Components of Metis

We develop Metis over MetisDB and incorporate it into the optimizer layer of the compute nodes. Figure 7 illustrates the workflow of Metis. After receiving query optimization requests, Metis first performs cost estimations for enumerated plans based on the cost model and cardinality estimation, in particular, using the new delta-store-aware cost model (Demain) for access path selection (§5). Then, Metis eliminates those far-from-optimal plans and uses its visibility-aware plan selection algorithm to form a set of seed plans (§6.1). Specifically, the seed plans include a row-oriented plan that has the lowest cost when only the row store is used, a column-oriented plan that has the lowest cost when only the row store is used, and an HTAP-aware hybrid plan that has the lowest cost when both the row store and the column store are considered.

To execute, Metis starts with the plan that has the minimal estimation cost in the set of seed plans and continuously monitors the query performance (e.g., by validating its statistics estimation). To ensure performance isolation, Metis monitors the resource utilization of the database. When the plan fails to fit in its performance boundary (e.g., 20% in our implementations, by default), either caused by violating performance isolation or errors in estimates for intermediate subexpressions, Metis stitches a new plan by reusing the sub-plans from the seed plans (§6.2). Since runtime statistics provide a more accurate runtime measurement, Metis can correct negative optimization either caused by resource contention or errors in cardinality estimation. Finally, the query results are returned to the clients.

Design Rationales. Metis improves the accuracy of traditional cost models by considering the mainstream architectures of distributed HTAP databases and picks visibility-aware plans by considering the data synchronization mechanism in which replicated data becomes visible in sequence (§6.1). As shown in Figure 7, different from sticking to *plan-first execute-next*, Metis interleaves plan optimization and execution. Thus, Metis enables the optimizer to defer the plan choice to run-time, conforming to the nature of the *data dynamicity* in a continuously updated HTAP database. Meanwhile, Metis can ensure performance isolation between OLTP and OLAP workloads by re-optimizing the generated "optimal" plans.

4.3 Limitations and Discussions

Metis has two limitations. First, same as other cost-model-based optimizers, Metis relies on cardinality estimation to predicate the size of the results set, which may not be accurate for complex queries [11, 27]. In our implementations, Metis inherits cardinality estimation, transformation rules, and logical optimizations (e.g., join re-ordering algorithm) from [5, 18, 33]. However, thanks to the adoption of proactive re-optimizations, Metis can mitigate the negative effects of errors by switching to a new plan in run-time.

Second, Metis models column scans as sequential scans and assumes all data are decompressed before computing. Notable data warehouses advance their OLAP performance by working directly over compressed data. It could be challenging when considering hybrid plans. In this case, we need to determine an assemblage point, design a new intermediate representation for data from the row store and column store, and model the cost. We leave these exciting explorations as our future work. However, indices should still be helpful to speed up queries, making the access path selection necessary. Additionally, in HTAP, data in the delta store cannot be compressed as new data is generated continuously. Thus, our cost model for the delta store is still valuable.

5 Demain Model

Demain provides an example of applying our optimization principles to evaluate the performance penalty of delta stores. Based on Demain, we theoretically analyze the impact of *data dynamicity* on access path selection, which is not well-studied by existing works (§8.1 and §8.2). Even though Demain may not be directly applicable to other HTAP databases since the implementations of storage may differ, the core concept of our delta-store-aware principle still holds. In the rest of this section, we first provide model preliminaries in §5.1, model the access paths in §5.2, and discuss the selection in §5.3.

5.1 Model Preliminaries

Demain models access paths in HTAP from four perspectives (see Table 2): queries, datasets, hardware, and storage. We differentiate the bandwidth for row- and column-oriented scans since the stores in MetisDB are deployed across machines and occupy individual resources. In the later reference, we also differentiate the I/O bandwidth for sequential and random access with a superscript.

Without loss of generality, our cost model targets range scans, which typically filter out data according to the given predictors. We show a sample in *SQL* below:

```
SELECT col1 FROM table WHERE col1 between ${a} and ${b}
```

When answering range queries in the row store of MetisDB (i.e., an LSM-tree-based key-value store), besides the requested data, tombstones (i.e., the metadata that records invalid instances of the deleted key) and invalid entries have to be read and discarded [58]. We omit the cost of reading the tombstones since their size contributes little to a full table scan. We model read amplifications by T (see Table 2), which is a factor that captures how much the size of the entries inserted in the LSM tree is greater than that of the tuples in the dataset (i.e., $N \cdot t_s$). As such, T can be calculated by obtaining the size of the entries in the LSM tree, which is maintained by the global Meta service of MetisDB (§4.1) at runtime.

5.2 Modeling Access Path In HTAP

Network I/O Cost. The storage-computation separation architecture may incur new overhead for retrieving data from storage nodes to computation nodes. We model the network cost by considering the transmission delay and in-network delay. For simplicity, we exclude the stack delay on the end host. As MetisDB performs in-storage data filtering, the size of transferred data is $sel \cdot N \cdot w_{res}$. Thus, we show the total cost of forwarding data results below, where the first part of the equation calculates transmission delay, and L_{net} models in-network delay.

$$Cost_{net} = \frac{sel \cdot N \cdot w_{res}}{B_{net}} + L_{net} \tag{1}$$

Row Scan. Scanning data in the underlying row stores are accessed sequentially for a given table. The cost of retrieving data on the storage node includes three parts. First, it requires moving data from disks (e.g., SSD) to memory. Second, it relies on moving data from memory to the CPU cache to perform scans. Third, it consumes CPU cycles to filter data according to the predictor. As all these steps are done in a pipeline manner (i.e., tuples are processed by memory and CPU before the entire database is moved from disk into memory), the total execution time is bounded by the maximum value of the three factors. We use the cost for retrieving data from disk as the approximate value since it costs much more than the other two factors. Thus, we have the cost for a row scan in seconds:

$$Cost_{row} = T \cdot N \cdot max\left\{\frac{ts}{DB_{row}^{seq}}, \frac{ts}{MB_{row}^{seq}}, f_p \cdot p\right\} \approx \frac{T \cdot N \cdot ts}{DB_{row}^{seq}} \tag{2}$$

Index Scan. Given an index on the accessed column, the cost of retrieving data from the index is two-fold. First, the index needs to be sequentially traversed to find a set of row IDs corresponding to the requested value range. Second, the storage engine should filter rows according to the proposed set without paying for the overhead of reading the entire key-value pairs. Similar to the row scan, we omit the cost of memory and CPUs. Thus, we have:

$$Cost_{index} = Cost_{IndexTraversal} + Cost_{DataTraversal} \tag{3}$$

$$\approx \frac{T \cdot N \cdot (w_k + w_{id})}{DB_{row}^{seq}} + \frac{T \cdot N \cdot (w_{id} + sel \cdot ts)}{DB_{row}^{rand}} \tag{4}$$

A special case is that if all needed data can be obtained from the indexed keys (a.k.a covering index), the cost of data traversal (i.e., the second part of Equation 3) can be eliminated.

Column Scan on Column Store. To perform a column scan on column store, MetisDB retrieves data from the specified single column directly instead of reading the entire tuple. Thus, using the same approximation as row scan, we have:

$$Cost_{col}^{*} = N \cdot max\left\{\frac{w_a}{DB_{col}^{seq}}, \frac{w_a}{MB_{col}^{seq}}, \frac{f_p \cdot p}{f_{vec}}\right\} \approx \frac{N \cdot w_a}{DB_{col}^{seq}} \tag{5}$$

Delta Scan. MetisDB implements $B+$ trees as a part of the append-only delta stores (§4.1). Same as the existing work [40], we model the cost on $B+$ tree in two parts. The first part is for traversing the internal structure of the $B+$ tree to find the starting point in the first leaf node corresponding to the requested value range. According to our preliminaries, the height of $B+$ is caculated as $\lceil log_b(N_d)\rceil$. Thus, MetisDB should retrieve $1 + \lceil log_b(N_d)\rceil$ tree nodes in total. One is added for the root node. As the $B+$ tree uses binary search to find the target child node, half of the keys will be read for each node, contributing to $b/2$. Thus, we have:

$$Cost_{TreeTraversal} = (1 + \lceil log_b(N_d)\rceil) \cdot \frac{b}{2} \cdot (f_p \cdot p + \frac{1}{MB_{col}^{rand}}) \tag{6}$$

The second part is for traversing leaf nodes to read the indexed row IDs and find the tuples according to the row IDs. Particularly, in the delta store, a tuple ID in a leaf node of the $B+$ tree can be either matched to a *delete* or an *insert* operation. For those *deletes*, a row ID does not essentially cost a read in the delta store since the content is not actually being used. MetisDB can merge those *deletes* into the results of the column scan by ignoring the invalidated entries with a matching tuple ID. In practice, we use lightweight statistics in the delta stores to count the number of *insert* operations, and the others are *delete* operations. For simplicity, we assume all operations are *inserts* in the following equations. We assume each tuple in the delta store matches a full tuple update, i.e., the size of the indexed tuple equals its tuple size in row format. We also assume a uniform distribution for data access and updates. Hence, the cost becomes:

$$Cost_{DataTraversal} = N_d \cdot \frac{(w_{id} + sel \cdot ts)}{DB_{col}^{rand}} \tag{7}$$

$$Cost_{delta} = Cost_{TreeTraversal} + Cost_{DataTraversal} \tag{8}$$

Query	sel	Selectivity of query q (%)
	w_{res}	Results witdh (bytes per output tuple)
Dataset	N	Number of tuples
	ts	Tuple size (bytes per tuple)
Hardware Resource	B_{net}	Network bandwidth (bytes/s)
	L_{net}	Latency of in-Network delay (s)
	DB_{row}	Disk bandwidth of read in row store (bytes/s)
	DB_{col}	Disk bandwidth of read in column store (bytes/s)
	MB_{row}	Memory bandwidth of read in row store (bytes/s)
	MB_{col}	Memory bandwidth of read in column store (bytes/s)
	p	The inverse of CPU frequency
	f_p	Factor accounting for instruction pipeline
	f_{vec}	Factor accounting for vectorized OLAP engine
Storage	T	Ratio of the size amplification in LSM tree
	w_a	Attribute width (bytes)
	w_{id}	RowID width (bytes)
	b	B+ tree fanout for delta store
	w_k	Key width of the index (bytes)
	N_d	Delta size (Unconsolidated tuples per column)

Table 2. Preliminaries and notations for Demain. We color all those HTAP-related preliminaries in grey.

Column Scan in HTAP. Putting Equation 8 and Equation 5 together, we have the overall cost of a column scan in HTAP:

$$Cost_{col} = Cost^{*}_{col} + Cost_{delta} \tag{9}$$

The total cost of retrieving data should also combine with the cost of network I/O for transmission data from storage nodes to computation nodes (see Equation 1), which can be critical for estimating the execution latency. However, the cost is not critical for access path selection since the storage node filters data according to the predictor locally, and thus the size of the transmitted results should be identical for different access paths.

5.3 Access Path Selection

Using the equations in §5.2, we first detail the comparison between row scans, column scans on column store, and index scans. From Equation 2 and Equation 5, it shows that, for a disk-based database, the major advancement of column stores is to help reduce I/O cost (i.e., slim down the cost from $T \cdot ts$ to w_a for each tuple, where ts is always $\geq w_a$ and T is always ≥ 1). This benefit is pronounced especially when the table has massive columns, and the number of requested columns is few. It becomes a bit tricky when an index exists on the conditional columns. The read performance of the row store can be enhanced since the index can help skip unnecessary tuple access but only read the entire tuples corresponding to the requested value range. Given this, traditional optimizers [6, 15, 60] decide the threshold of switching access path depending on the query selectivity (i.e., sel in Equation 4). When the query has low selectivity, the overall cost of the index scan should be lower. It should also be noted that index scans may have poorer performance than row scans when the sel is particularly large since they pay additional overhead on sequential index traversal and do data traversal randomly.

We then discuss the access path selection in Metis. Metis refines the cost of column scan in Equation 9 by incorporating the performance penalty on the delta store. Hence, the row store has more potential to outperform the column store (Equation 8). According to our model, Metis prefers a row scan when the *sel* is particularly large, and the performance penalty of N_d outpaces the benefit of slimming I/O cost $T \cdot ts$ to w_a. When *sel* is particularly small, Metis prefers an index scan. Otherwise, Metis chooses a column scan.

6 Runtime Optimizations

In this section, we present our visibility-aware plan selection algorithm and resource-aware plan re-optimization in Metis.

6.1 Visibility-aware Plan Selection

Given the revised cost model, when performing plan enumeration, Metis represents each physical plan in a directed acyclic graph (DAG), which constructs a partial order for plan execution. Metis leverages DAGs to predict plans' performance. Each vertex in the graph corresponds to a physical operator. Each edge represents a dependency between two operators due to data dependency. Specifically, there exist two types of edges: if an operator (O_i) must wait for the whole data output of another operator (O_j) before execution, we term such a dependency as a hard dependency ($O_i \longrightarrow O_j$); otherwise, if O_i can be executed in a pipeline manner, we term it as a soft dependency ($O_i \rightsquigarrow O_j$).

For example, if a hybrid plan ***P1*** retrieves data from table A in the row store using an index scan (O_{A_index}), retrieves data from table B in the column store using a column scan (O_{B_column}), and joins table A and table B using a pipelined hash join [32] by assuming table A as a build table (O_{A_build}) and table B as a probe table ($O_{B\_ probe}$), then there exist three edges between four operators: $O_{A\_index} \rightsquigarrow O_{A\_build}$, $O_{B\_column} \rightsquigarrow O_{B\_ probe}$, and $O_{A\_build} \longrightarrow O_{B\_ probe}$.

Recall our insight into the visibility-aware plan selection: scheduling pre-execution on the available data ahead instead of blocking queries until all data becomes visible can help mask the visibility delay of HTAP databases (§3.2). Therefore, in ***P1***, Metis can retrieve data from the row store and perform the first phase (i.e., building a hash table) of the hash join ahead of retrieving data from the column store, reducing the overall response latency. Compared to an alternative physical plan ***P2*** that retrieves all data from the column store and then performs hash join, ***P1*** may outperform ***P2*** in query latency even if the execution time of the index scan is a bit longer than the column scan since ***P2*** has to be blocked until the new data of table A is integrated into the column store.

Another example is shown in Figure 8. We show four candidate plans that have the same logical structure (i.e., using the optimized logical plan) but adopt different access paths for physical operators. For simplicity, we assume using hash joins for all join operators. Thus, the parts of plans in the green box are ready to be executed at the time of plan generation, and the other parts should be blocked. Note that even though Plan#4 opts to retrieve data for Table C from the row store, it has to be blocked due to data dependency in DAG.

Algorithm 1 shows the pseudocode for our visibility-aware plan selection. Based on the DAG abstraction, we first calculate pre-execution tasks for each plan by removing unavailable pending tasks in the graph (Lines 13-14). By doing so, we get a set of pre-execution tasks (i.e., *subs*). Each task in the set is a strongly connected component that contains multiple physical operators. Since there is no data dependency between the tasks in *subs*, they can be executed in parallel. Therefore, the overall performance improvement of scheduling pre-execution should be the execution time of the longest task. We combine the knowledge of visibility delay into the plan cost at Line 18, which captures the end-to-end query latency observed by users. Finally, we use the revised cost to guide the selection function (Lines 4-10) and generate a visibility-aware physical plan that has the cheapest cost on query latency.

Visibility Delay Estimation. In MetisDB, visibility delay comes from four aspects (see Figure 6): first, new writes are committed into the data transfer pipeline and then shipped from the row store to the column store; second, the transferred logical log are parsed by the delta store; third,

Algo 1: Visibility-aware Plan Selection (§6.1).

```
Para: $G_p$ ← The DAG representation of a hybrid plan $p$
Para: $\mathcal{M}$ ← The cost model that estimates cost in seconds
Para: $\alpha$ ← The visibility delay in the HTAP database
Function planSelection($DAGs$, $\alpha$) do
    $bestPlan$ ← ∅;  $Cost_{bset}$ ← 0;
    for each $G_p$ in $DAGs$ do
        $Cost_p$ ← calculateCost($G_p$, $\alpha$);
        if $bestPlan$ == ∅ ∨ $Cost_p < Cost_{best}$ :
            $bestPlan = p$;  $Cost_{best}$ ← $Cost_p$;
    return $bestPlan$;
Function calculateCost($G_p$, $\alpha$) do
    $preCost$ ← 0;  $exeCost$ ← estimateCost ($G_p$, $\mathcal{M}$) ;
    $invalidSet$ ← findAllPendingTasks($G_p$);
    $subs$ ← removePendingTasks($G_p$, $invalidSet$);
    /* removePendingTasks returns the strongly connected components after removing pending
       tasks in $G_p$. */
    foreach $sub \in subs$ do
        $Cost_{sub}$ ← estimateCost (sub, $\mathcal{M}$);
        if $preCost < Cost_{sub}$ : $preCost$ ← $Cost_{sub}$ ;
    return Cost ← $exeCost + \alpha$ - min {$preCost, \alpha$};
```

new updates are appended into the delta store and indexed by B+ tree; fourth, the updates are committed into the delta store and finally becomes visible to queries.

As such, besides the time for data shipping that is bounded by the physical limitations (i.e., in-network delay), the foremost parameter that affects the visibility delay is loads of writes: more writes mean more data to be processed and lead to more queuing time. Moreover, as the delta store of MetisDB is append-only, scattered updates from the row store are also appended into the delta store. Thus, we do not need to consider the distribution of updates.

Given this, Metis estimates visibility delay using historical data, which maps the concurrency of OLTP workloads to visibility delay. An example of the mapping is shown in Figure 4. For robustness, our Global Meta Service (§4.1) continuously collects new measurements (e.g., a data sample for each second) and calibrates the estimation at runtime. We calculate the moving average (i.e., the average number of the last 30 samples) of visibility delay for the estimation of each concurrency and solve the curve using polynomial regression.

In our practice, our approach for visibility delay estimation is effective enough for the implementation of our visibility-aware plan selection algorithm (see Figure 13). Note that Metis do not rely on a strictly precise estimation. Underestimating makes Metis prefer columns. However, even for the worst case, Metis can still consistently outperform HTAP-agnostic plans, which can be considered a special case when the estimated delay = 0. In contrast, over-estimating is rare as we use historical data to estimate and capture regular traffic. System exceptions (e.g., congestion) usually lead to underestimating. We study the performance sensitivity in Figure 15.

Future Extension. For data consistency, MetisDB guarantees snapshot isolation (§4.1), and Metis always uses the latest timestamp to execute queries, achieving the best data freshness. In the previous example (e.g., Figure 8), we assume all data in the column store is not visible at the plan-generated time and becomes visible atomically when all new data is synchronized into the delta store. This simplified abstraction matches the behavior that the database retrieves data with a global timestamp ts_q and blocks all reads until the timestamp of the delta store $ts_{del} \geq ts_q$ to guarantee all

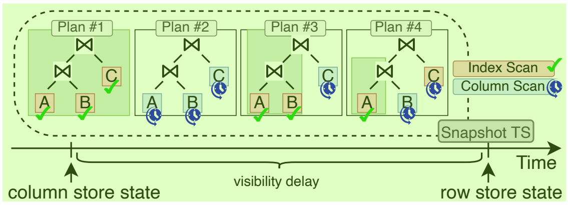


Fig. 8. An example of visibility-aware plan selection. The parts in green are ready to be executed.

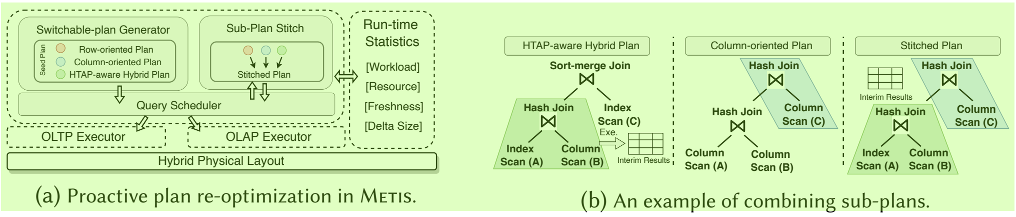


Fig. 9. Proactive plan re-optimization and sub-plan stitch.

transactions with $ts \leq ts_{del}$ are visible by the column scan.

In practice, METISDB tracks timestamps in the granularity of data chunks, where updates to a data chunk are sequenced individually. Thus, in METIS, a portion of data can be available in the delta store at the beginning, and the visibility of the delta store increases gradually. Therefore, incremental computation techniques can be combined into the pre-execution on the columns to improve the performance further. We leave these developments as our future work since it is orthogonal to our proposal.

6.2 Proactive Query Re-optimizations

To keep plans efficient and workloads isolated, METIS re-optimizes plans proactively. Figure 9a shows an overview of our approach. Intuitively, when performing query optimizations, METIS generates a set of seed plans (i.e., a row-oriented plan, a column-oriented plan, and an HTAP-aware plan) to save the cost of re-optimization. When executing queries, METIS starts with the cheapest plan and continuously re-optimizes plans according to the runtime statistics.

Seed Plans. To generate seed plans, METIS uses the revised cost model (§5) and visibility-aware plan selection algorithm (§6.1). METIS first generates a hybrid plan with the cheapest cost. Based on the same logical structure of the hybrid plan, Metis generates an optimized row-oriented plan and an optimized column-oriented plan by considering different physical operators. Therefore, all seed plans have the same logical structure but may adopt different physical operators (e.g., row scan versus column scan, nested loop join versus hash join, and stream aggregation versus hash aggregation). Generally, seed plans consist of three physical plans when there exists a hybrid plan that outperforms the row-oriented and column-oriented plans; otherwise, seed plans consist of two plans.

We do not individually optimize the logical structure for the row-oriented and column-oriented plans since the cardinality estimations that can influence the optimality of the logical plan (e.g., to decide an optimal join order) are exactly the same. For re-optimizations on the logical structures at runtime (e.g., runtime join reorder), several notable works [3, 8, 46] have been proposed, which are orthogonal to our paper and can be adopted by METIS.

Runtime Statistics. To decide whether a query plan should be re-optimized, METIS quantitatively evaluates the effects of its decisions based on runtime statistics. For efficient execution, METIS re-optimizes physical operators when the estimated cardinality is far from the real statistics (e.g., beyond 20%). For the performance isolation, METIS stitches sub-plans to alleviate resource contention. We use a threshold of the observed physical resource utilization to control the use of the physical resource (e.g., 80% CPU utilization) and use a threshold of contention footprint (i.e., the number of conflict transactions) to control the logical contention on each table.

Plan Stitch. Inspired by [25], when re-optimizing a plan, METIS stitches sub-plans from the seed plans without losing partial work in the query execution pipeline. As the seed plans in METIS have the same logical structure, stitching a new plan is essentially re-optimizing the selection of physical operators for the remaining un-executed plans. Thus, works done in the previous plan can be partially reused in the new stitched plan (see our example below).

Example. The example query in Figure 9b joins Tables A, B, and C. METIS starts the execution with the cheapest hybrid plan and generates the interim results by joining Tables A and B. Due to inaccurate cardinality or resource contention, re-optimization is triggered when retrieving data from Table C. In this case, METIS stitches a new plan with the column-oriented plan in the set of seed plans. Using seed plans, METIS avoids a re-estimation for the cost of different physical operators by carrying the previous information of the first-pass query optimization into the plan execution. A similar idea is also used in dynamic programming.

The stitched plan uses a column scan for Table C instead of an index scan for better performance. Consequently, METIS uses a hash join instead of a sort-merge join because the cost will be higher when the column store can not provide a sort order of the input data. Using this stitched plan, METIS then reuses the interim results from joining Table A and Table B to build a hash table for joining Table C. By doing so, METIS eliminates a complete re-execution.

7 EXPERIMENTS

In this section, we first study the overall performance of METIS. We then evaluate the three key techniques (i.e., ① delta-store-aware cost model, ② visibility-aware plan selection, and ③ resource-aware plan re-optimization) in sequence. Finally, we discuss the learned lessons. Our evaluation focused on the following questions:

§7.2 What is the overall performance of METIS?
§7.3 What are the benefits of using Demain in METIS?
§7.4 How does the visibility-aware plan selection algorithm help?
§7.5 How does the re-optimization behave?
§7.6 Which techniques are more beneficial to HTAP databases?

7.1 Evaluation Setups

Hardware Configurations. We ran all experiments on a cluster with seven machines. Each machine has a 2.60GHz Intel(R) Xeon(R) E5-2690 v3 CPU (i.e., 24 cores with a single NUMA node), 64GB memory with 544Gbps bandwidth, 960GB Dell DD4G0 SSD with 6Gbps bandwidth, and a 40Gbps NetXtreme NIC.

System Deployments. We used three row stores that partition the database horizontally and three column stores that use the same partition policy as row stores. Each of the stores is deployed on individual machines co-located with the execution engines. We set up a client program along with the global meta service on a stand-by machine to send client requests and maintain globally consistent meta. We emulated 3*ms* in-network delays among machines using Linux tc [34], which is in line with the latency inside a data center [9].

Workloads. To emulate diverse application scenarios and analyze the performance of METIS, we used three well-studied workloads.

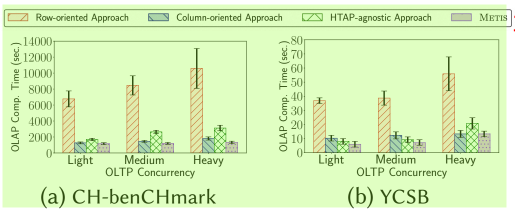


Fig. 10. Analytical Queries Completion Time.

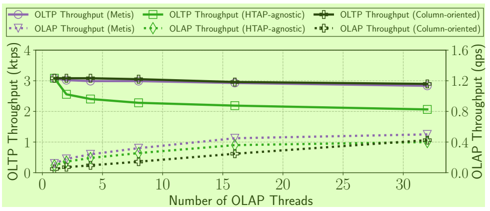


Fig. 11. Impact of OLAP on OLTP performance. The left y-axis is for OLTP, and the right y-axis is for OLAP.

CH-benCHmark. We used CH-benCHmark (for short, CH), a notable workload in HTAP scenarios, to evaluate the performance of Metis under hybrid workloads. It integrates an OLAP workload (i.e., TPC-H [21]) into an OLTP workload (i.e., TPC-C [22]) with a unified data schema. It contains five types of transactions and twenty-two types of analytical queries. Similar to TPC-C, CH organizes data by warehouses. We used 100 data warehouses in our experiments.

TPC-DS. We adopted TPC-DS [23] with $sf = 100$ for analyzing hybrid plans in §2.2. As TPC-DS is a pure OLAP workload without any write transactions, we did not use it in evaluating Metis.

YCSB. To fine-tune the workloads, we developed a micro-benchmark using the APIs of YCSB [20]. The micro-benchmark includes two tables (A and B). Each table has 100 million rows with a 64-bit primary key attribute and ten data attributes. Primary indices are built on each table. In addition to the transactions that perform ten random reads or writes, we create an analytical query (**Q1**) that joins the two tables and controls the amount of data accessed by each table with range predictors; a query (**Q2**) that queries Table A with a range on the primary key (§3.1).

Baselines. The primary competitor of Metis is using HTAP-agnostic plans, which are used by multiple existing HTAP databases [33, 41, 67]. Specifically, the HTAP-agnostic approach reused the existing cost model from the row store and added column scans as an alternative data access path without considering *data dynamicity* (§3). For example, PolarDB-IMCI [67] inherited the cost model from *MySQL* and forwarded queries to the column store only when the cost exceeds a pre-defined threshold.

In our experiments, we emulated the HTAP-agnostic approach using the same codebase of Metis while disabling all HTAP-aware optimizations (i.e., without ①, ②, and ③). In addition, we also study the performance of only using row- and column-oriented plans for OLAP queries, where the optimization space is restricted to row or column stores. It should be noted that the baseline using only row- and column-oriented plans may not be realistic in a real deployment, as they rely on user hints to distinguish OLTP and OLAP queries, limiting the functionality of HTAP. On the contrary, Metis or the HTAP-agnostic approach can distinguish them automatically according to the cost model (even if the model may not be correct).

7.2 Overall Performance of Metis.

To study the performance of Metis (i.e., using ① + ② + ③) under HTAP workloads, we created three distinct scenarios for CH and YCSB, where the OLTP is light (~20% peak throughput), medium (~50% peak throughput), and high (~80% peak throughput).

Specifically, we used 32 OLTP threads for the light concurrency, achieving 732.54 transactions per second (for short, *tps*) for CH and 8180.42 *tps* for YCSB; we used 128 OLTP threads for the medium concurrency, achieving 1930.98 *tps* for CH and 23108.06 *tps* for YCSB; we used 256 OLTP threads for the high concurrency, reaching 3082.64 *tps* for CH and 41556.60 *tps* for YCSB.

Figure 10 reports the completion time of analytical queries in ten rounds. For CH, OLAP clients issued 220 TPC-H-like queries iteratively. For YCSB, OLAP clients issued 10 YCSB $Q1$ with 0.01%

selectivity in Table A and 1% selectivity in Table B. We used 24 threads for intra-query parallelism.

The results demonstrated that OLTP concurrency could affect the performance of all candidate query plans. This is because, as the OLTP concurrency increased, more transactions consumed more resources in the row store and thus affected the performance of row and index scans. Meanwhile, data synchronization could also lead to read amplification and influence the performance of column scans. However, compared to the row and index scans, the performance degradation on the column scans is much smaller since it avoids direct resource conflicts with OLTP. Compared to HTAP-agnostic plans, the performance degradation of Metis was relatively small. This was because Metis could select the access path for sub-queries more accurately (to be illustrated in §7.3).

For the YCSB workload, we observed that, under light and medium OLTP concurrency, both the approach using HTAP-agnostic plans and Metis could generate the optimal plans. We checked the physical plan by the ANALYZE statements in *SQL*. The plan retrieved data from Table A using index scans and from Table B using column scans. Then, the plan joined Table A with Table B using Hash Join. Metis performed slightly better because our visibility-aware plan selection algorithm could schedule the execution on the row store ahead (i.e., all data was available in the delta store).

Under high OLTP concurrency, Metis shifted to the column plan to alleviate resource contention on the row store. At the same time, the HTAP-agnostic approach still used the hybrid plan, leading to sub-optimal performance (i.e., 1.56× latency, Figure 10b).

Performance Isolation. Maintaining strong performance isolation between OLTP and OLAP is an important design goal of Metis. We study the property under the CH workload. We first set up 256 OLTP threads to saturate the OLTP throughput and then increased the OLAP workloads by adding OLAP threads.

As shown in Figure 11, for Metis, the OLAP throughput increased with the number of OLAP threads and eventually plateaued; the OLTP throughput was basically preserved throughout the experiment. On the contrary, the HTAP-agnostic approach incurred a much more severe OLTP throughout degradation compared to Metis, in line with the evaluation results in §3.3. This is because Metis will proactively re-optimize in-efficient plans in the face of resource contention. After re-optimizations, Metis uses the column-oriented plans or sub-plans for the analytical queries. Thus, Metsis can achieve the same performance isolation property (i.e., less than 8% OLTP throughout degradation) as the approach without using hybrid plans (e.g., the column-oriented approach in Figure 11).

Takeaway: Metis can efficiently leverage hybrid plans to speed up analytical queries in different HTAP workloads while preserving strong performance isolation between OLTP and OLAP.

7.3 Benefits of Using Demain

In this experiment, we disabled ② and ③ and studied the performance benefits of Demain (①) under the default settings of the CH-benCHmark workload. For OLTP, we ran 256 threads to saturate the throughput, leading to 3082.64 *tps*. For OLAP, we executed analytical queries in sequence.

Table 3 shows the latency of analytical queries. For the HTAP-agnostic approach and Metis (Demain-only), we marked the generated row plans with $\mathcal{R}$, column plans with C, and hybrid plans with $\mathcal{H}$. Overall, Metis performed better than the other three competitors. Compared to the HTAP-agnostic approach, Metis's realistic competitor, Metis achieved 1.72× speedups in geometric mean.

For specific queries, both the approach using HTAP-agnostic plans and Metis can benefit from hybrid plans (e.g., $Q4$). This is because, for these queries, neither the row nor the column store can be superior for retrieving data. However, the approach using HTAP-agnostic plans can lead to poor performance due to error-prone cost estimation (as discussed in §3). For example, $Q3$. Hence, the latency of the HTAP-agnostic plans was even 1.52× slower than the column-oriented plans. We highlight the sub-optimal cases in grey (i.e., the cases where HTAP-agnostic plans performed worse

	Row-oriented	Col.-oriented	HTAP-agnostic	METIS (Demain Only)
Q1	69.06s	7.37s	7.37s ($\mathcal{C}$)	7.37s ($\mathcal{C}$)
Q2	26.98s	2.42s	2.42s ($\mathcal{C}$)	2.42s ($\mathcal{C}$)
Q3	48.71s	4.52s	48.71s ($\mathcal{R}$)	4.52s ($\mathcal{C}$)
Q4	5.68s	7.02s	3.51s ($\mathcal{H}$)	3.51s ($\mathcal{H}$)
Q5	10.27s	11.32s	7.78s ($\mathcal{H}$)	7.78s ($\mathcal{H}$)
Q6	9.33s	1.00s	1.00s ($\mathcal{C}$)	1.00s ($\mathcal{C}$)
Q7	29.55s	4.59s	2.89s ($\mathcal{H}$)	2.89s ($\mathcal{H}$)
Q8	46.18s	8.42s	8.42s ($\mathcal{C}$)	8.42s ($\mathcal{C}$)
Q9	158.92s	28.97s	183.81s ($\mathcal{H}$)	28.97s ($\mathcal{C}$)
Q10	3.70s	4.68s	4.99s ($\mathcal{H}$)	3.70s ($\mathcal{C}$)
Q11	14.77s	2.78s	2.78s ($\mathcal{C}$)	2.78s ($\mathcal{C}$)
Q12	26.27s	2.56s	42.19s ($\mathcal{H}$)	2.56s ($\mathcal{C}$)
Q13	60.74s	5.73s	5.73s ($\mathcal{C}$)	5.73s ($\mathcal{C}$)
Q14	12.73s	1.40s	1.40s ($\mathcal{C}$)	1.40s ($\mathcal{C}$)
Q15	150.10s	2.63s	2.63s ($\mathcal{C}$)	2.63s ($\mathcal{C}$)
Q16	27.05s	1.25s	1.25s ($\mathcal{C}$)	1.25s ($\mathcal{C}$)
Q17	92.02s	12.02s	12.02s ($\mathcal{C}$)	12.02s ($\mathcal{C}$)
Q18	8.67s	14.11s	8.67s ($\mathcal{R}$)	8.67s ($\mathcal{R}$)
Q19	35.55s	2.82s	28.38s ($\mathcal{H}$)	2.82s ($\mathcal{C}$)
Q20	41.69s	6.38s	6.38s ($\mathcal{C}$)	6.38s ($\mathcal{C}$)
Q21	OOM	18.93s	9.42s ($\mathcal{H}$)	9.42s ($\mathcal{H}$)
Q22	3.57s	1.21s	9.69s ($\mathcal{H}$)	1.07s ($\mathcal{H}$)
G-Mean	24.87s	4.53s	6.91s	4.03s

Table 3. A comparison of using different optimization approaches. We mark the generated row plans with $\mathcal{R}$, column plans with $\mathcal{C}$, and hybrid plans with $\mathcal{H}$ in the last two columns. For the HTAP-agnostic approach, we highlight the negative optimization cases in grey, which are corrected in METIS by Demain.

than the best of row- and column-oriented plans). Similar negative effects were also validated in [28]. Furthermore, we observed that, on specific queries (e.g., $Q9$, $Q10$, $Q12$, and $Q22$), the performance of the hybrid plans generated by the HTAP-agnostic approach might be even poorer than both row- and column-oriented plans.

Demain corrects these sub-optimal plans by improving the accuracy of cost estimation. For example, in $Q9$, the HTAP-agnostic plan retrieves data from the row store for the table `order` and the rest of the data from the column store. On the contrary, according to our new cost model, METIS additionally retrieves data from the row store for the table `nation` and `supplier`.

In addition to the three baselines we presented, one may think about building an optimizer that selects the best of row- and column-oriented plans. However, such an approach should only be feasible with a unified cost model that can precisely predicate the cost of the two plans. When creating a unified cost model, the problem of the HTAP-agnostic approach exists, i.e., such an optimizer can choose sub-optimal plans due to the error-prone cost estimation.

Accuracy Analysis of Demain. We then individually analyzed the accuracy of Demain by comparing the predicted crossover selectivity against the measured crossover selectivity for access path selection. Recall the definition of crossover in §3.1. The crossover is the selectivity when the row and column scans have an identical execution time. We run YCSB for ten minutes with different OLTP concurrency to warm up. We used YCSB $Q2$ and fine-tuned the selectivity of the predictor to find the measured crossover. The predicted crossover was calculated by solving the equation when the index scan has the exact cost as the column scan in §5. As shown in Figure 12, the error rate was up to 12.28% and within the standard deviation of the measured crossover selectivity.

Takeaway: Demain, as a key component of METIS, can accurately predicate the crossover selectivity

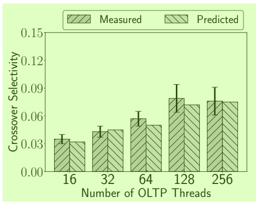


Fig. 12. Demain Accuracy.

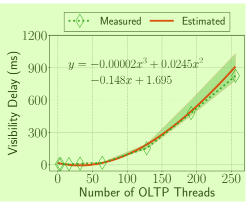
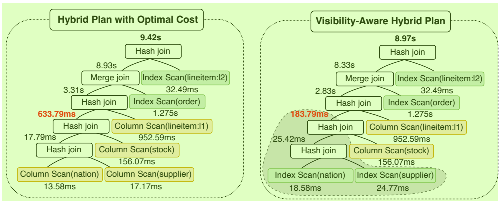


Fig. 13. Estimate Accuracy.



(a) Naive hybrid plan. (b) Visibility-aware plan

Fig. 14. An example of using pre-execution.

	*Q*4	*Q*5	*Q*7	*Q*21	*Q*22	G-mean
Visibility-Agnostic	3.51s	7.78s	2.89s	9.42s	1.07s	3.80s
Visibility-Aware	3.07s	7.10s	2.53s	8.97s	0.87s	3.36s

Table 4. Benefits of visibility-aware plan selection algorithm.

between index and column scans and thus guide the access path selection in MetisDB. Using Demain, Metis corrected sub-optimal plans of the HTAP-agnostic approach and achieved the best performance among the competitors.

7.4 Benefits of Visibility-Aware Plan Selection

We then study the preformance of Metis when our visibility-aware plan selection algorithm (②) was enabled. As the algorithm uses a cost model as an input, we also used Demain in this experiment (i.e., using ① + ②). We used the same set-ups of CH-benCHmark as those in §7.2. Table 4 shows the results. For clarity, we only present the queries that can benefit from pre-execution. The latency of the other queries remained the same as Metis (Demain-only) in Table 3. Among the five queries, our algorithm achieved 1.13× speedups in geometric mean compared to the visibility-agnostic approach.

Case Study. To provide an in-depth analysis of the algorithm, we study the physical plan of *Q*21 with the accumulated execution time for each operator. Figure 13a shows the plan with "*optimal* " cost, which was generated by disabling the algorithm (i.e., using Demain-only). Correspondingly, Figure 13b shows the visibility-aware plan.

When retrieving data from Table `nation` and `supplier`, column scan can outperform row scan slightly (i.e., 13.58*ms* versus 18.58*ms* and 17.17*ms* versus 24.77*ms*). However, due to the visibility delay (i.e., ~800*ms* in our experiment), data in the column store was unavailable until all new data was synchronized from the row store. Therefore, the plan with the "*optimal* " cost had to be blocked and not scheduled until new data became visible in the column store. On the contrary, our visibility-aware plans could be scheduled ahead of the full data synchronization as new data was available in the row store. By doing so, a portion of the plan (i.e., the grey part in Figure 13b) could be executed ahead of time to mask the visibility delay. Then, the amortized cost (183.79*ms*) of the sub-plan was cheaper than the sub-plan of the "*optimal* " plan (633.79*ms*).

As a result, though the visibility-aware plan could pay more cost to retrieve data, its latency was shorter than the plan with the "*optimal* " cost (i.e., 8.97*s* compared to 9.42*s*).

Accuracy of Visibility Delay Estimation. We studied the accuracy of our estimation in Figure 13. For each concurrency, the measured visibility delay was calculated as the average of delays during each second over 5 minutes. The estimation curve was solved by fitting the average of samples (except the outliners) using a cubic polynomial. Overall, our estimation is roughly accurate, and the error rate was up to 24.8% of the observed samples.

Sensitiviy Study. As shown in Figure 15, we conducted a sensitivity analysis to determine how the estimation errors affect the performance of analytical queries. We studied the five CH queries

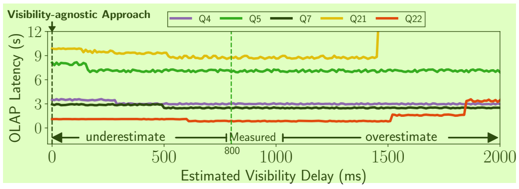


Fig. 15. Sensitiviy Study of Visibility Delay.

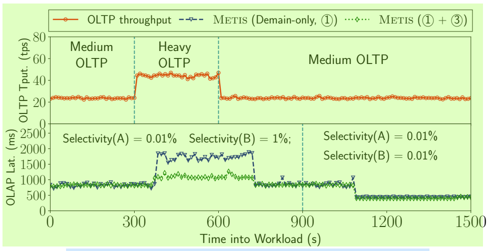


Fig. 16. Impact of Shifted Workload.

from Table 4. We set the estimated visibility delay from 0*ms* to 2000*ms* in the step of 10*ms*. As pointed out in §6.1, the visibility-agnostic approach was a special case when estimated visibility delay = 0*ms*. Overall, underestimation makes queries prefer column scans, and the performance of queries can fall back to the performance of the visibility-agnostic approach. Over-estimation makes queries prefer row scan, and the performance of queries can fall back to the performance of the row-oriented approach when the estimation tends to infinity. However, such situations can rarely happen in a real use case (§6.1).

Takeaway: For specific queries, our algorithm has the potential to further enhance the query performance by pre-executing the non-blocking sub-plans. Estimation errors of visibility delay may impact the quality of the query plans but can be mitigated in practice.

7.5 The Case for Re-optimizations

The benefits of resource-aware query re-optimization (③) are two-fold: first, it maintains the performance isolation between OLTP and OLAP (evaluated in Figure 11); second, it keeps the generated plan efficient using accurate runtime statistics (e.g., the case for YCSB under high OLTP concurrency, §7.2).

In this experiment, we demonstrate how our re-optimization approach helps METIS achieve the two benefits by examining its OLTP throughput and OLAP latency over time. As the re-optimizations are complementary to the approach of improving the cost model for realizing the two benefits (e.g., re-optimization captures runtime resource competition between OLTP and OLAP that Demain does not cover), we also studied the performance of using Demain-only for the baseline. For clarity, we disabled ②.

Figure 16 shows the OLTP throughput and OLAP (i.e., YCSB $Q1$) latency. To make the OLAP latency stable, in this experiment, we used a single OLAP thread to send $Q1$ iteratively, thus resulting in roughly the same OLTP throughout for the two competitors.

At the beginning of our experiment (i.e., 0~300*s*), we used 128 OLTP threads for medium OLTP throughout. Using Demain, both the two competitors could generate the optimal hybrid plan. Recall that the plan prefers a hybrid plan that retrieves data from Table A using index scans and Table B using column scans (same as §7.2).

As we shifted the number of OLTP threads from 128 to 256 at 300*s*, the OLTP increased correspondingly, which consumes ~ 92% of CPU cycles on the row store. In such a case, METIS (① + ②) avoided retrieving data from the row store favor for both query efficiency and performance isolation between workloads. It re-optimized the hybrid plan using seed plans (§6.2) and thus retrieved data from both Tables A and B using column scans. Differently, METIS (Demain-only) still used the same hybrid plan as in the first phase because Demain did not capture the resource conflicts on CPUs and regarded disk I/O as the major parameter.

At 900*s*, we changed the selectivity on Table B. Both the two variations of METIS generated a row-oriented plan and joined the two tables using Sort-merge Join instead of Hash Join.

Takeaway: The re-optimization can help METIS adapt to the shifted workload smoothly while keeping performance isolated between workloads and providing reasonable OLAP performance.

7.6 Lessons Learned

So far, we have studied the performance of METIS and individually evaluated the benefits of the three key optimizations (①, ②, and ③). We now summarize the lessons we learned from this study:

First, improving the cost model for access path selection is the key to achieving the potential of hybrid plans. Sub-optimal plans can lead to negative optimization (e.g., up to 16.4× performance degradation, see Table 3), overshadowing all the potential benefits.

Second, visibility-aware plan selection can further enhance the performance of hybrid plans by pre-execution. However, the potential is physically bounded by the visibility delay between the row and column store. The improvements of specific queries depend on the accuracy of the cost model and delay estimation.

Third, re-optimizations can mitigate the cost model's inaccuracy and handle resource conflicts that are out-of-model. In this way, re-optimizations are the key to performance isolation and can not be eliminated even if the inputs of the cost model (e.g., cardinality estimation) are perfectly accurate.

8 RELATED WORK

8.1 Query Optimization in HTAP Databases

Independent Cost Model. Several HTAP databases [12, 16, 51] process query optimizations for OLTP and OLAP independently, using a routing-based approach, and are not designed for hybrid data access. After receiving *SQL* requests, they rely on embedded user-level hints or a middleware layer to differentiate OLTP and OLAP queries. Then, they execute queries on the desirable engines and stores. This approach is easy to implement; however, it is at the cost of performance and functionality since the prior knowledge of queries can be challenging to acquire and may be inaccurate.

Unified Cost Model. Another approach is supplementing column stores as an additional data access path. For instance, F1 Lightning [68] generates logical plans using its F1 optimizer (i.e., an optimizer designed for OLTP) and considers lightning-only indexes and views during physical planning. SQL Server [28] analyzes and recommends column stores by its Database Engine Tuning Advisor (DTA) when suitable for a given workload. TiDB [33] extends query optimizer to explore physical plans accessing both row and column stores. Oracle Dual [41] supplements column indices to its optimizer as an alternate execution method for high-speed table scans. Compared to METIS, all of them either do not support *hybrid plans* (e.g., F1) or generate *HTAP-agnostic plans* (e.g., TiDB).

8.2 Access Path Selection in Modern Databases

Access path selection is one of the most fundamental optimizations in databases for retrieving data from tables. Recent studies focus on the analysis of large-scale column databases, in-memory row databases, and hybrid physical designs in HTAP databases.

Kester et al. [40] analyze the problem of access path selection for in-memory analytical databases by comparing probes on $B+$ trees to shared scans on column stores. Dziedzic et al. [28] present the analysis of access path selection for a commercial-strength database, considering secondary $B+$ trees on top of column-store indexes. Abadi et al. [1] provide an experimental study to quantify the significant differences between column store and row-oriented $B+$ trees. Unlike existing works, our paper discusses the access path selection over the specially-tailored HTAP databases (i.e., using delta-main architectures).

8.3 Using Pre-execution for Optimization

Speculative Execution allows a processor to perform a series of tasks before it is prompted to [29, 52]. It is well-studied in operating systems. If it turns out the work was not needed after all, the results are ignored. Differently, METIS never trashes pre-execution results.

Streaming Processing (e.g., Kafka [64] and Flink [14]) process queries incrementally. When certain input data streams are blocked, they pre-execute the queries on available data. METIS shares a similar design. However, differently, METIS selects a consistent data view before execution and considers different data formats for query optimizations.

8.4 Proactive Query Re-optimizations

Several influential works [10, 25, 27, 36, 37, 47, 66, 70] inspire our resource-aware re-optimization approach. We discuss some of them below.

Proactive Re-optimizations. Babu et al. [10] estimate statistics computed as bounding boxes and generate a switchable seed plan for runtime re-optimization. Ding et al. [25] harness valuable information of efficient sub-plans collected from other previously executed plans and stitch these sub-plans at runtime.

Resource-aware Query Plan. Viswanathan et al. [66] integrate resource planning within a query planner using a cost-based model in Hive and Spark. Li et al. [47] propose a resource-aware deep-learning model that can predict the execution time of plans and thus combine the knowledge of available resources into query planning.

Differently, METIS is additionally designed to keep performance isolation between workloads, which is HTAP-specific.

9 CONCLUSION

In this paper, we systematically analyze the potential and challenges of hybrid plans in a distributed HTAP database. We propose METIS, an HTAP-aware hybrid optimizer. METIS uses a revised cost model for access path selection, selects plans with a visibility-aware algorithm, and re-optimizes queries proactively. Our experiments show the efficiency and adaptivity of METIS.

Acknowledgments

We thank all anonymous reviewers for their valuable comments. The work is supported in part by the Alibaba Group, the HKU-SCF FinTech Academy R&D Funding Scheme in 2021 and 2022, HK RIF (R7030-22), HK ITF (GHP/169/20SZ), and HK GRF (17208223).

References

[1] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data.* 967–980.

[2] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. 2022. *Proteus: Autonomous Adaptive Storage for Mixed Workloads.* Technical Report. Technical Report. University of Waterloo. https://cs. uwaterloo. ca ….

[3] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. 2012. Reoptimizing Data Parallel Computing.. In *NSDI*, Vol. 12. 281–294.

[4] Nitin Agrawal and Ashish Vulimiri. 2017. Low-latency analytics on colossal data streams with summarystore. In *Proceedings of the 26th Symposium on Operating Systems Principles.* 647–664.

[5] Rafi Ahmed, Allison Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zait, and Thierry Cruanes. 2006. Cost-based query transformation in Oracle. In *VLDB*, Vol. 6. 1026–1036.

[6] Gennady Antoshenkov. 1993. Dynamic query optimization in Rdb/VMS. In *Proceedings of IEEE 9th International Conference on Data Engineering.* IEEE, 538–547.

[7] Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2017. Janus: A hybrid scalable multi-representation cloud datastore. *IEEE Transactions on Knowledge and Data Engineering* 30, 4 (2017), 689–702.

[8] Ron Avnur and Joseph M Hellerstein. 2000. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data.* 261–272.

[9] AWS. 2023. AWS Latency Monitoring. https://www.cloudping.co/grid.

[10] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 107–118.

[11] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. 2015. Smooth scan: Statistics-oblivious access paths. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 315–326.

[12] Dennis Butterstein, Daniel Martin, Knut Stolze, Felix Beier, Jia Zhong, and Lingyun Wang. 2020. Replication at the Speed of Change: A Fast, Scalable Replication Solution for near Real-Time HTAP Processing. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3245–3257. https://doi.org/10.14778/3415478.3415548

[13] Shaosheng Cao, XinXing Yang, Cen Chen, Jun Zhou, Xiaolong Li, and Yuan Qi. 2019. TitAnt: Online Real-Time Transaction Fraud Detection in Ant Financial. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2082–2093. https://doi.org/10.14778/3352063.3352126

[14] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[15] Surajit Chaudhuri. 2009. Query optimizers: time to rethink the contract?. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 961–968.

[16] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, et al. 2022. ByteHTAP: bytedance's HTAP system with high data freshness and strong data consistency. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3411–3424.

[17] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. 2021. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 210–227.

[18] Inc. ClickHouse. 2023. ClickHouse — open source distributed column-oriented DBMS. https://github.com/ClickHouse/ClickHouse/tree/22.6.

[19] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. 2011. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*. 1–6.

[20] Brian Cooper. 2010. Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB.

[21] The Transaction Processing Council. 1992. TPC-H. http://www.tpc.org/tpch/.

[22] The Transaction Processing Council. 2014. TPC-C. http://www.tpc.org/tpcc/.

[23] The Transaction Processing Council. 2015. TPC-DS. http://www.tpc.org/tpcds/.

[24] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. 2014. YCSB+ T: Benchmarking web-scale transactional databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE, 223–230.

[25] Bailu Ding, Sudipto Das, Wentao Wu, Surajit Chaudhuri, and Vivek Narasayya. 2018. Plan stitch: harnessing the best of many plans. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1123–1136.

[26] Science Direct. 2004. Real-Time Pricing. https://www.sciencedirect.com/topics/engineering/real-time-pricing.

[27] Anshuman Dutt and Jayant R Haritsa. 2014. Plan bouquets: query processing without selectivity estimation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1039–1050.

[28] Adam Dziedzic, Jingjing Wang, Sudipto Das, Bolin Ding, Vivek R Narasayya, and Manoj Syamala. 2018. Columnstore and B+ tree-Are Hybrid Physical Designs Important?. In *Proceedings of the 2018 International Conference on Management of Data*. 177–190.

[29] Kaushik Ghosh. 1995. *Speculative execution in real-time systems.* Ph. D. Dissertation. Citeseer.

[30] Matteo Golfarelli and Stefano Rizzi. 2017. From Star Schemas to Big Data: 20 Years of Data Warehouse Research. *A comprehensive guide through the Italian database research over the last 25 years* (2017), 93–107.

[31] Google. 2022. AlloyDB for PostgreSQL under the hood: Columnar engine. https://cloud.google.com/blog/products/databases/alloydb-for-postgresql-columnar-engine.

[32] Hui-I Hsiao, Ming-Syan Chen, and Philip S Yu. 1994. On parallel execution of multiple pipelined hash joins. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*. 185–196.

[33] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[34] Bert Hubert. 2023. tc(8), Linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html.

[35] SnowFlake Inc. 2023. Unistore: A modern approach to working with transactional and analytical data together in a single platform. https://www.snowflake.com/workloads/unistore/.

[36] Alekh Jindal, Lalitha Viswanathan, and Konstantinos Karanasos. 2019. Query and Resource Optimizations: A Case for Breaking the Wall in Big Data Systems. *arXiv preprint arXiv:1906.06590* (2019).

[37] Navin Kabra and David J DeWitt. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 106–117.

[38] The kernel development community. 2023. Control Groups. https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.

html.

[39] Michael S Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access path selection in main-memory optimized data systems: Should I scan or should I probe?. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 715–730.

[40] Michael S Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access path selection in main-memory optimized data systems: Should I scan or should I probe?. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 715–730.

[41] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. 2015. Oracle database in-memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1253–1258.

[42] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time analytical processing with SQL server. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1740–1751.

[43] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, and Wook-Shin Han. 2017. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1598–1609.

[44] Mark Levene and George Loizou. 2003. Why is the snowflake schema a good data warehouse design? *Information Systems* 28, 3 (2003), 225–240.

[45] Guoliang Li and Chao Zhang. 2022. HTAP Databases: What is New and What is Next. In *Proceedings of the 2022 International Conference on Management of Data*. 2483–2488.

[46] Quanzhong Li, Minglong Shao, Volker Markl, Kevin Beyer, Latha Colby, and Guy Lohman. 2006. Adaptively reordering joins during query execution. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 26–35.

[47] Yan Li, Liwei Wang, Sheng Wang, Yuan Sun, and Zhiyong Peng. 2022. A Resource-Aware Deep Cost Model for Big Data Query Processing. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 885–897.

[48] Chen Luo and Michael J Carey. 2019. On performance stability in LSM-based storage systems. *Proceedings of the VLDB Endowment* 13, 4 (2019).

[49] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, et al. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 2530–2542.

[50] Elena Milkai, Yannis Chronis, Kevin P Gaffney, Zhihan Guo, Jignesh M Patel, and Xiangyao Yu. 2022. How Good is My HTAP System?. In *Proceedings of the 2022 International Conference on Management of Data*. 1810–1824.

[51] MySQL. 2022. MySQL Heatwave. https://dev.mysql.com/doc/heatwave/en/heatwave-introduction.html.

[52] Edmund B Nightingale, Peter M Chen, and Jason Flinn. 2005. Speculative execution in a distributed file system. *ACM SIGOPS operating systems review* 39, 5 (2005), 191–205.

[53] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid transactional/analytical processing: A survey. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1771–1775.

[54] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Performance Evaluation and Benchmarking: First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers 1*. Springer, 237–252.

[55] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. 2013. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1080–1091.

[56] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through elastic resource scheduling. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2043–2054.

[57] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2016. L-store: A real-time OLTP and OLAP system. *arXiv preprint arXiv:1601.04084* (2016).

[58] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A tunable delete-aware LSM engine. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 893–908.

[59] Hemant Saxena, Lukasz Golab, Stratos Idreos, and Ihab F Ilyas. 2021. Real-time LSM-trees for HTAP workloads. *arXiv preprint arXiv:2101.06801* (2021).

[60] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. 23–34.

[61] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 219–238.

[62] Inc. SingleStore. 2023. SingleStore: Real-Time Distributed SQL. https://www.singlestore.com/.

[63] T Spenser and T Loukas. 1999. From Star to Snowflake to ERD: Comparing Data Warehouse Design Approaches. *Enterprise Systems Journal* 14 (1999), 62–69.

[64] Khin Me Me Thein. 2014. Apache kafka: Next generation distributed messaging system. *International Journal of Scientific Engineering and Technology Research* 3, 47 (2014), 9478–9483.

[65] Panos Vassiliadis. 2009. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining (IJDWM)* 5, 3 (2009), 1–27.

[66] Lalitha Viswanathan, Alekh Jindal, and Konstantinos Karanasos. 2018. Query and resource optimization: Bridging the gap. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1384–1387.

[67] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, Yujie Wang, Baokai Chen, Chang Cai, Yubin Ruan, Xiaoyi Weng, Shibin Chen, Liang Yin, Chengzhong Yang, Xin Cai, Hongyan Xing, Nanlong Yu, Xiaofei Chen, Dapeng Huang, and Jianling Sun. 2023. PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. *Proc. ACM Manag. Data* 1, 2, Article 199 (jun 2023), 25 pages. https://doi.org/10.1145/3589785

[68] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, et al. 2020. F1 Lightning: HTAP as a Service. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3313–3325.

[69] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, gLiu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: reducing write stalls and write amplification in LSM-tree based KV stores with a matrix container in NVM. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 17–31.

[70] Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. 2015. Robust query optimization methods with respect to estimation errors: A survey. *ACM Sigmod Record* 44, 3 (2015), 25–36.

Received April 2023; revised July 2023; accepted August 2023