Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads

Joy Arulraj
Carnegie Mellon University
jarulraj@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Prashanth Menon
Carnegie Mellon University
pmenon@cs.cmu.edu

ABSTRACT

Data-intensive applications seek to obtain trill insights in real-time by analyzing a combination of historical data sets alongside recently collected data. This means that to support such hybrid workloads, database management systems (DBMSs) need to handle both fast ACID transactions and complex analytical queries on the same database. But the current trend is to use specialized systems that are optimized for only one of these workloads, and thus require an organization to maintain separate copies of the database. This adds additional cost to deploying a database application in terms of both storage and administration overhead.

To overcome this barrier, we present a hybrid DBMS architecture that efficiently supports varied workloads on the same database. Our approach differs from previous methods in that we use a single execution engine that is oblivious to the storage layout of data without sacrificing the performance benefits of the specialized systems. This obviates the need to maintain separate copies of the database in multiple independent systems. We also present a technique to continuously evolve the database's physical storage layout by analyzing the queries' access patterns and choosing the optimal layout for different segments of data within the same table. To evaluate this work, we implemented our architecture in an in-memory DBMS. Our results show that our approach delivers up to 3× higher throughput compared to static storage layouts across different workloads. We also demonstrate that our continuous adaptation mechanism allows the DBMS to achieve a near-optimal layout for an arbitrary workload without requiring any manual tuning.

1. INTRODUCTION

Organizations need to quickly transform freshly obtained data into critical insights. Such workloads, colloquially known as hybrid transaction-analytical processing (HTAP), seek to extrapolate insights and knowledge by analyzing a combination of historical data sets with real-time data [40, 48]. Data has immense value as soon as it is created, but that value diminishes over time. For many application domains, such as high-frequency trading and Internet advertising, it is imperative that this analysis uses the newest data for the results to have the most impact. Many organizations implement HTAP pipelines using separate DBMSs. The most common practice is to use one DBMS for transactions and another for analytical queries. With this model, the new information from transactions goes into an on-line transactional processing (OLTP) DBMS. Then in the background, the system uses an extract-transform-load utility to migrate data from the OLTP DBMS to a data warehouse for on-line analytical processing (OLAP).

There are several problems inherent in such a bifurcated environment. Foremost is that the time it takes to propagate changes between the separate systems is often measured in minutes or even hours. This data transfer inhibits an application's ability to act on data immediately when it is entered in the database. Second, the administrative overhead of deploying and maintaining two different DBMSs is non-trivial as personnel is estimated to be almost 50% of the total ownership cost of a large-scale database system [45]. It also requires the application developer to write a query for multiple systems if they want to combine data from different databases.

A better approach is to use a single HTAP DBMS that can support the high throughput and low latency demands of modern OLTP workloads, while also allowing for complex, longer running OLAP queries to operate on both hot (transactional) and cold (historical) data. What makes these newer HTAP systems different from legacy general-purpose DBMSs is that they incorporate many of the advancements over the last decade from the specialized OLTP and OLAP systems. The key challenge with HTAP DBMSs, however, is executing OLAP workloads that access old data along with new data in the database while simultaneously executing transactions that update the database. This is a difficult problem and existing HTAP DBMS resort to using separate query processing and storage engines for data that is stored in different layouts. That is, they employ a separate OLTP execution engine for row-oriented data that is better for transactions and a separate OLAP execution engine for column-oriented data that is better for analytical queries. They then have to use a synchronization method (e.g., two-phase commit) to combine the results from the two different parts of the system [5, 29, 47]. Cobbling systems together in this manner increases the complexity of the DBMS and degrades performance due to the additional overhead needed to maintain the state of the database across different runtimes. This in turn limits the types of questions that can be asked about data as soon as it enters the database, which is the main selling point of a HTAP DBMS.

To overcome this problem, we present a method to bridge the architectural gap between the OLTP and OLAP systems using a unified architecture. Our approach is to store tables using hybrid layouts based on how the DBMS expects tuples will be accessed in the future. This means that a table's "hot" tuples are stored in a format that is optimized for OLTP operations, while the other "cold" tuples in that same table are stored in a format that is more

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2915231

amenable to OLAP queries. We then propose a logical abstraction over this data that allows the DBMS to execute query plans that span these different layouts without using separate engines and with minimal overhead. Our other contribution is a novel on-line reorganization technique that continuously enhances each table's physical design in response to an evolving query workload. This enables the DBMS to migrate the database to the near-optimal storage layout for an arbitrary application without requiring a human administrator to configure the DBMS for the particular application and in a transactionally safe manner.

To evaluate our approach, we implemented our storage and execution architecture in the Peloton HTAP DBMS [1]. We compare our methods with other state-of-the-art storage models and show that they enable the DBMS to achieve up to 3× higher throughput across different hybrid workloads. We also demonstrate that our reorganization method allows the DBMS to continuously modify the layout of tables without manual tuning and with minimal overhead.

The remainder of this paper is organized as follows. We first discuss the performance impact of a storage model, and the benefits of a flexible storage model for HTAP workloads in Section 2. Next, in Section 3, we describe the design of our system architecture based on this model. We then present the concurrency control mechanism that enables the DBMS to support hybrid workloads in Section 4, followed by our on-line layout reorganization technique in Section 5. We then present our experimental evaluation in Section 6. We conclude with a discussion of related work in Section 7.

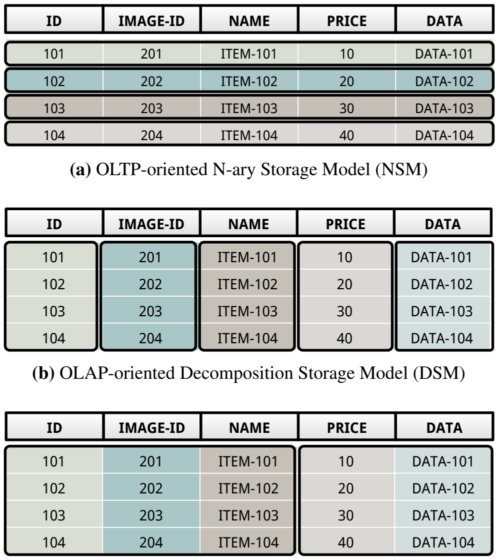
2. MOTIVATION

We begin with an overview of the impact of the table's storage layout on the performance of the DBMS for different types of workloads. We then make the case for why a "flexible" storage model is the best choice for HTAP workloads.

2.1 Storage Models

There are essentially two ways that DBMSs store data: (1) the n-ary storage model and (2) decomposition storage model. These models prescribe whether the DBMS stores the data in a tuple-centric or in an attribute-centric manner. Note that the choice of the storage model is independent of whether the primary storage location of the database is on disk or in-memory.

All DBMSs that are based on the architecture of the original DBMSs from the 1970s (i.e., IBM System R, INGRES) employ the n-ary storage model (NSM). With this approach, the DBMS stores all of the attributes for a single tuple contiguously. In the example shown in Figure 1a, all the attributes belonging to the first tuple (`ID #101`) are stored one after another, followed by all the attributes of the second tuple (`ID #102`). NSM works well for OLTP workloads because the queries in transactions tend to operate only on an individual entity in the database at a time (e.g., a single customer record), and thus they need to access most (if not all) of the attributes for that entity. It is also ideal for insert-heavy workloads, because the DBMS can add tuples to the table using a single write.



ID	IMAGE-ID	NAME	PRICE	DATA
101	201	ITEM-101	10	DATA-101
102	202	ITEM-102	20	DATA-102
103	203	ITEM-103	30	DATA-103
104	204	ITEM-104	40	DATA-104

(a) OLTP-oriented N-ary Storage Model (NSM)

ID	IMAGE-ID	NAME	PRICE	DATA
101	201	ITEM-101	10	DATA-101
102	202	ITEM-102	20	DATA-102
103	203	ITEM-103	30	DATA-103
104	204	ITEM-104	40	DATA-104

(b) OLAP-oriented Decomposition Storage Model (DSM)

ID	IMAGE-ID	NAME	PRICE	DATA
101	201	ITEM-101	10	DATA-101
102	202	ITEM-102	20	DATA-102
103	203	ITEM-103	30	DATA-103
104	204	ITEM-104	40	DATA-104

(c) HTAP-oriented Flexible Storage Model (FSM)

Figure 1: Storage Models – Different table storage layouts work well for OLTP, OLAP, and HTAP workloads. The different colored regions indicate the data that the DBMS stores contiguously.

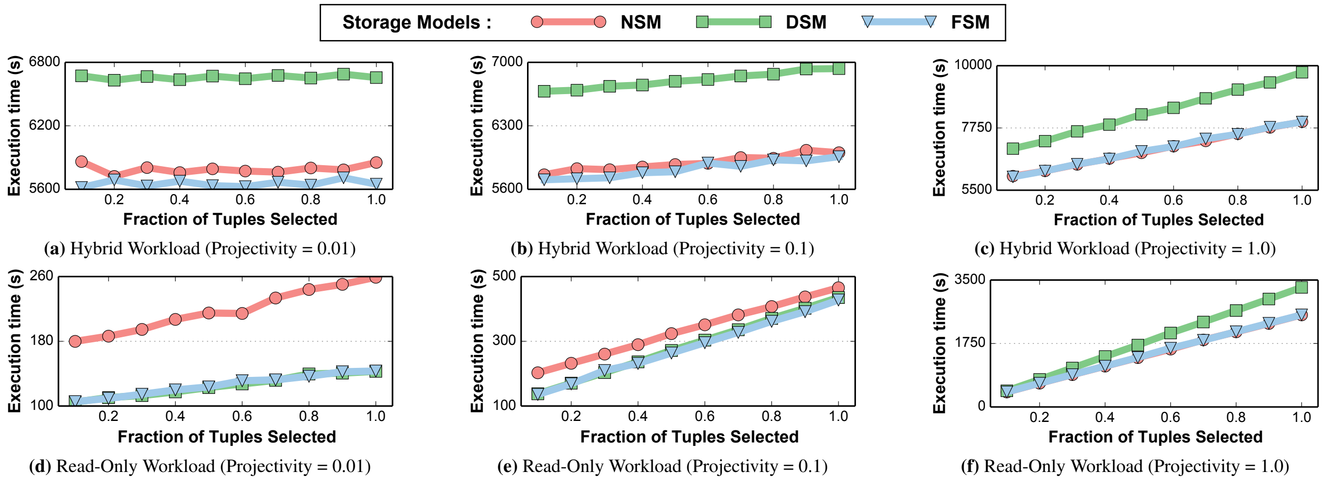
NSM is not a good choice, however, for analytical queries in OLAP workloads. This is because these queries tend to access multiple entities in a table at the same time. These queries also typically only access a subset of the attributes for each entity. For example, a query might only analyze the location attribute for all of the customer records within a particular geographic region. But NSM-based DBMSs can only read tables tuple-at-a-time, and therefore they process the data in a tuple-centric manner within their query operators [21]. Prior research has shown that this execution strategy has lower CPU efficiency due to high interpretation overhead [15]. NSM-based DBMSs squander I/O and memory bandwidth when executing OLAP queries because they unnecessarily access and process attributes that are not even needed for the final result.

An alternate approach, known as the decomposition storage model (DSM), stores data attribute-at-a-time [7, 17]. That is, the DBMS stores the tuples' values for a single attribute in a table contiguously. This is the storage model employed in modern OLAP DBMSs, including Vertica [49] and MonetDB [15]. Figure 1b shows an example of the storage layout of a table using DSM: the DBMS allocates space for all the values belonging to the first attribute (`ID`), followed by those that belong to the second attribute (`IMAGE-ID`). DSM works well for OLAP workloads because the DBMS only retrieves the values belonging to those attributes that are needed by the query. They also process the data attribute-at-a-time, thereby improving the CPU efficiency due to lower interpretation overhead and skipping unnecessary attributes [6, 7].

But just like how NSM systems are inefficient for read-only OLAP workloads, DSM systems are not ideal for write-heavy OLTP workloads. This is because these workloads are comprised of queries that insert and update tuples into the table. Unlike in NSM, this is an expensive operation in DSM, as the storage manager needs to copy over the tuples' attributes to separate storage locations. Hence, the NSM and DSM architectures pose a problem for a DBMS supporting HTAP workloads that include transactions that update the state of the database while also executing complex analytical queries on this data. We contend that it is better for the DBMS to use a flexible storage model that provides the benefits of both the NSM and DSM for their respective workloads, while avoiding the problems that they encounter with the other workloads.

2.2 The Case for a Flexible Storage Model

We refer to a storage model that generalizes the NSM and DSM as the *flexible* storage model (FSM). It supports a wide variety of hybrid storage layouts where the DBMS co-locates the attributes that are frequently accessed together. We defer a detailed description of the FSM to Section 3. In the example shown in Figure 1c, all the values belonging to the first three attributes (`ID, IMAGE-ID, NAME`) are stored contiguously, followed by those belonging to the last two attributes (`PRICE, DATA`).



(a) Hybrid Workload (Projectivity = 0.01)

(b) Hybrid Workload (Projectivity = 0.1)

(c) Hybrid Workload (Projectivity = 1.0)

(d) Read-Only Workload (Projectivity = 0.01)

(e) Read-Only Workload (Projectivity = 0.1)

(f) Read-Only Workload (Projectivity = 1.0)

Figure 2: Performance Impact of the Storage Models – Time taken by the execution engine to run (1) a hybrid, and (2) a read-only workload on top of the NSM, DSM, and FSM storage managers.

FSM-based DBMSs can exploit a fundamental property of HTAP workloads in modern database applications. In these workloads, the access patterns of transactions are such that tuples are more likely to be updated in an OLTP transaction when they are first added to the database. Then, over time they become *colder* and thus are less likely to be updated again. For instance, more than half of the content that Facebook users access and interact with are shared by their friends in the past two days, and then there is a rapid decline in content popularity over the following days [13]. For these workloads, it is advantageous to store the hot data in a tuple-centric layout (NSM) since it is likely to be modified during this period. But then once a particular data item passes some threshold, the DBMS can reorganize the cold data to an attribute-centric layout (DSM) that works well for analytical queries [47]. A FSM-based DBMS can support these HTAP workloads efficiently by storing different parts of the same table under different hybrid storage layouts.

To better understand these issues, we now compare the performance impact of these different storage models in a motivating experiment. Consider a database with a single table $\mathcal{R}(a_0, a_1, \ldots, a_{500})$ that has 10m tuples. Each attribute a_k is a random integer value in the range $[-100, 100]$. The application for this database contains the following two queries:

Q_1: `INSERT INTO` R `VALUES` $(a_0, a_1, \ldots, a_{500})$
Q_2: `SELECT` $a_1, a_2, \ldots, a_k$ `FROM` R `WHERE` $a_0 < \delta$

The transactional query Q_1 inserts a tuple into the table R, while the analytical query Q_2 projects a subset of attributes a_1,a_2, …,a_k from all the tuples in the table R that satisfy the predicate $a_0 < \delta$. Note that different values for k and δ affect its projectivity and selectivity, respectively. We consider two workloads: (1) a *hybrid* workload of 1000 scan (Q_2) queries followed by 100m insert (Q_1) queries, and (2) a *read-only* workload of 1000 scan (Q_2) queries.

We measure the total time that an in-memory DBMS takes to execute these two workloads when its storage manager uses the NSM, DSM, and FSM storage layouts. For the FSM layout, the DBMS co-locates the attributes accessed in Q_2 together in the following layout: $\{\{a_0\}, \{a_1, \ldots, a_k\}, \{a_{k+1}, \ldots, a_{500}\}\}$. We consider three scenarios where we increase the projectivity of the scan query (Q_2) progressively from 1% to 100%. For each of these projectivity settings, we also vary the selectivity of the query from 10% to 100%. We note that the time taken to execute a scan query is higher than the time taken to execute an insert query. We defer the description of our experimental setup to Section 6.

Figures 2a to 2c present the results for the hybrid workload. We observe that NSM and FSM storage managers outperform the DSM storage manager by up to 1.3× across all scenarios. This is because DSM needs to split a tuple's attributes on every insert and store them in separate memory locations. FSM works well on this workload because it adopts wider vertical partitions similar to the NSM, and therefore executes insert queries faster than DSM. On the scan queries, FSM outperforms NSM because it stores the predicate attribute (a_0) and the projected attributes ($a_1, \ldots, a_k$) separate from the other attributes in the table. During both predicate evaluation and subsequent projection, the FSM storage manager retrieves only the attributes required for query processing.

The results for the read-only workload are shown in Figures 2d to 2f. We see that on this workload the DSM and FSM layouts outperform the NSM on low projectivity analytical queries. They execute the workload up to 1.8× faster than NSM due to better utilization of memory bandwidth. This experiment highlights the performance impact of the storage model on HTAP workloads. It is clear from this that neither a NSM nor a DSM storage layout is a good solution for HTAP workloads. A better approach is to use a FSM DBMS that can process data stored in a wide variety of hybrid layouts, including the canonical tuple-at-a-time and the attribute-at-a-time layouts. Given this, we now present our FSM-oriented DBMS architecture.

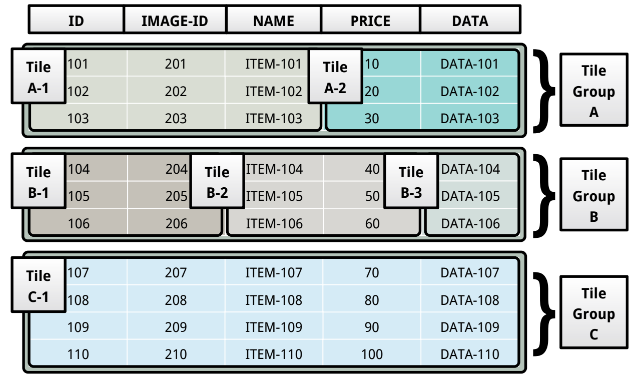
3. TILE-BASED ARCHITECTURE

We now describe a manifestation of the FSM approach that uses a storage abstraction based on *tiles*. A tile is akin to a vertical/horizontal segment of table. We begin with a description of a storage architecture based on *physical tiles*. We then describe how to hide the layout of these physical tiles from the DBMS's query processing components through *logical tiles*. We will present our concurrency control protocol for this architecture in Section 4 and the dynamic layout reorganization method in Section 5.

3.1 Physical Tile

The fundamental physical storage unit of the DBMS is a *tile tuple*. Informally, a tile tuple is a subset of attribute values that belong to a tuple. A set of tile tuples form a physical tile. We refer to a collection of physical tiles as a *tile group*. The storage manager physically stores a table as a set of tile groups. All the physical tiles belonging to a tile group contain the same number of tile tuples.

Consider the table layout shown in Figure 3. The table comprises



ID	IMAGE-ID	NAME	PRICE	DATA
101	201	ITEM-101	10	DATA-101
102	202	ITEM-102	20	DATA-102
103	203	ITEM-103	30	DATA-103
104	204	ITEM-104	40	DATA-104
105	205	ITEM-105	50	DATA-105
106	206	ITEM-106	60	DATA-106
107	207	ITEM-107	70	DATA-107
108	208	ITEM-108	80	DATA-108
109	209	ITEM-109	90	DATA-109
110	210	ITEM-110	100	DATA-110

Figure 3: Physical Tile – An example storage layout of a table composed of physical tiles. This table comprises of three tile groups (A, B, C).

of three tile groups (A, B, C), that each contain a disparate number of physical tiles. The tile group A consists of two tiles (A-1, A-2). The tile A-1 contains the first three attributes (ID, IMAGE-ID, NAME) of the table, while the tile A-2 contains the remaining two attributes (PRICE, DATA). These two tiles form the tile group A.

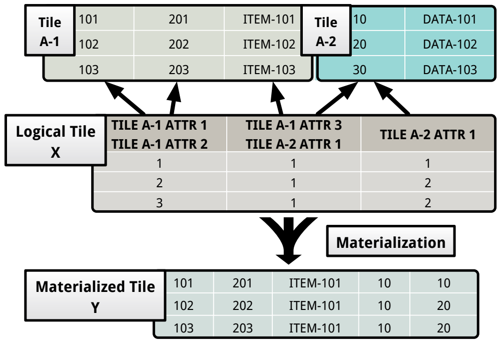
A tuple can be stored in an FSM database using different layouts over time. For instance, the default approach is for the DBMS to store all the new tuples in a tuple-centric layout, and then as they become colder, it reorganizes them into a layout with narrower OLAP-friendly vertical partitions. This is done by copying over the tuples to a tile group with the new layout, and swapping the original tile group in the table with the newly constructed one. As we describe in Section 5, this reorganization occurs in the background and in a transactionally safe manner to avoid false negatives/positives.

We note that the NSM and DSM layouts are special cases of the FSM layout when using tiles. If each tile group consists of one tile that contains all the attributes of the table, then it is the same as the NSM's tuple-centric layout. On the other hand, if each tile consists of exactly one attribute, then it is equivalent to the DSM's attribute-centric layout. Besides supporting flexible vertical partitioning, this tile-based architecture also supports horizontal partitioning of a table. This design choice allows the DBMS to configure the number of tuples stored in a tile group, such that the tiles fit within the CPU cache. For two tables with different schemas, the DBMS can therefore choose to store different number of tuples per tile group.

3.2 Logical Tile

An architecture based on physical tiles enables the DBMS to organize data in any possible layout that works well for the HTAP workload. The problem with storing data in different layouts, however, is that it is difficult to efficiently execute queries over this data. This is because the design of the DBMS's query processing components is not optimized for a particular data layout. One solution is to transform each tuple into a standard format, independent of the layout of its tile group when a query reads the data. But this requires extra processing overhead for each query, and thus the DBMS loses the benefits of the OLTP- or OLAP-optimized storage layouts.

Another approach is for the DBMS to use multiple execution engines for query processing that are optimized for the different hybrid layouts. But this requires expensive merging of the results produced by the operators in the different execution engines [47]. In this case, the DBMS needs to employ an additional synchronization method (beyond its internal concurrency control protocol) to enforce ACID guarantees. Beyond these two issues, just having to maintain multiple execution paths in the DBMS's source code that are specialized for the different layouts is notoriously difficult [50].



Tile A-1

101	201	ITEM-101
102	202	ITEM-102
103	203	ITEM-103

Tile A-2

10	DATA-101
20	DATA-102
30	DATA-103

Logical Tile X

TILE A-1 ATTR 1 TILE A-1 ATTR 2	TILE A-1 ATTR 3 TILE A-2 ATTR 1	TILE A-2 ATTR 1
1	1	1
2	1	2
3	1	2

Materialized Tile Y

101	201	ITEM-101	10	10
102	202	ITEM-101	10	20
103	203	ITEM-101	10	20

Figure 4: Logical Tile – An example of a logical tile representing data spread across a couple of physical tiles (A-1, A-2).

To overcome this problem, we propose an abstraction layer in our architecture based on logical tiles. A *logical tile* succinctly represents values spread across a collection of physical tiles from one or more tables. The DBMS uses this abstraction to hide the specifics of the layout of the table from its execution engine, without sacrificing the performance benefits of a workload-optimized storage layout. We now illustrate the abstraction with an example.

For the example in Figure 4, the logical tile X points to data in the two physical tiles A-1 and A-2. Each *column* of a logical tile contains a list of offsets corresponding to the tuples in the underlying physical tiles. A column in a logical tile can represent the data stored in one or more attributes spread across a collection of physical tiles. The DBMS stores this mapping in the logical tile's metadata region. It records this information only once for a logical tile's column. For instance, the first column in X maps to the first and second attributes of A-1. When the DBMS materializes X, it uses the first column to construct the first two attributes of the materialized tile Y.

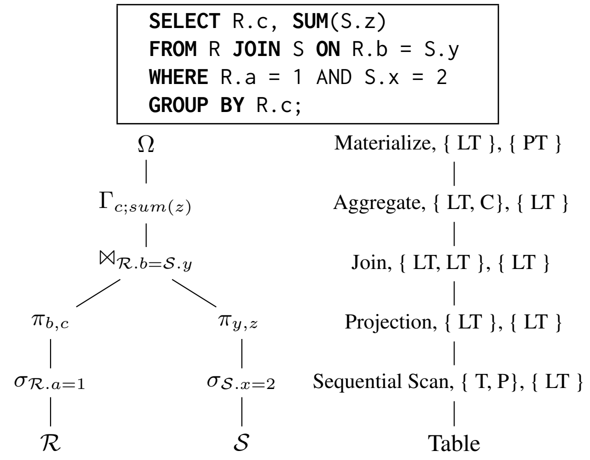
The value stored in the first column of the first row of X represents the values present in the first two attributes of the first tuple of A-1. During materialization, the DBMS transforms it into {101, 201}. Similarly, the value stored in the the second column of the first row of X maps to the third attribute of A-1 and the first attribute of A-2. On materialization, it becomes {ITEM-101, 10}.

For all the attributes that a column in a logical tile maps to, the DBMS uses the same list of tuple offsets during materialization. It is possible for two columns in a logical tile (with unique offset lists) to map to the same attribute in a physical tile. This is exemplified by the second and the third columns of X. To simplify the abstraction, we restrict logical tiles from referring to other logical tiles, meaning that every logical tile column maps to an attribute in a physical tile.

We contend that this logical tile abstraction provides expressive power even after this simplification. We next demonstrate how it enables a DBMS to process data organized in different layouts. During query execution, the DBMS can dynamically choose to materialize a logical tile, that is an intermediate query result, into a physical tile [6]. In this case, the operator constructs a *passthrough* logical tile with only one column that directly maps to the attributes in the materialized physical tile, and propagates that logical tile upwards to its parent in the plan tree.

3.3 Logical Tile Algebra

In order for the DBMS to benefit from storing data in physical tiles, it must be able to execute queries efficiently while still remaining agnostic to the underlying layouts of tuples. We refer to this property of abstracting away the physical layout of the data from the DBMS's query processing components as *layout transparency*. This reduces the coupling between the DBMS's storage manager and



```
SELECT R.c, SUM(S.z)
FROM R JOIN S ON R.b = S.y
WHERE R.a = 1 AND S.x = 2
GROUP BY R.c;
```

Ω
$\Gamma_{c;sum(z)}$
$\bowtie_{\mathcal{R}.b=\mathcal{S}.y}$
$\pi_{b,c}$
$\pi_{y,z}$
$\sigma_{\mathcal{R}.a=1}$
$\sigma_{\mathcal{S}.x=2}$
$\mathcal{R}$
$\mathcal{S}$
Materialize, { LT }, { PT }
Aggregate, { LT, C}, { LT }
Join, { LT, LT }, { LT }
Projection, { LT }, { LT }
Sequential Scan, { T, P}, { LT }
Table

Figure 5: Sample SQL query and the associated plan tree for illustrating the operators of the logical tile algebra. We describe the name, inputs, and output of each operator in the tree. We denote the logical tile by LT, the physical tile by PT, the table by T, the attributes by C, and the predicate by P.

execution engine. We now present an algebra defined over logical tiles that allows the DBMS to achieve layout transparency.

Consider the SQL query that accesses the tables $\mathcal{R}(a, b, c)$ and $\mathcal{S}(x, y, z)$ and its corresponding plan tree shown in Figure 5. It selects some tuples from $\mathcal{R}$ based on a, and some tuples from $\mathcal{S}$ based on x. It then joins the resulting tuples based on b and y. Finally, it groups the resulting tuples by c, and for each group, it computes the sum over z. Using this query as an example, we now describe the operator semantics of our logical tile algebra.[1]

Bridge Operators: Most of the logical tile algebra operators produce and consume logical tiles, which means that they are oblivious to the storage layout of the underlying physical tiles. The only operators that interact with the storage manager are the ones at the bottom and the top of the tree. We refer to these operators as *bridge operators* because they connect logical tiles and physical tiles.

The bridge operators include the table access methods, such as the *sequential scan* and the *index scan* operators. The sequential scan operator generates a logical tile for every tile group in the table. Each logical tile only contains one column that is a list of offsets corresponding to all the tuples in the tile group satisfying the predicate. In Figure 5, the sequential scan (σ) operator associated with $\mathcal{R}$ emits logical tiles that represent the tuples that satisfy the predicate a=1. The *index scan* operator identifies the tuples matching the predicate using the index, and then constructs one or more logical tiles that contain the matching tuples.

The DBMS uses the *materialize* operator to transform a logical tile to a physical tile. It also uses this operator to perform early-materialization [6]. In Figure 5, the aggregate operator (Γ) constructs a physical tile with the aggregate tuples, and then wraps around it with a passthrough logical tile. In order to send the query result to the client, the DBMS runs the materialize operator (Ω) for transforming the passthrough logical tile returned by the aggregate operator to a physical tile. In this case, the materialize operator does not need to construct a new physical tile. Instead, it directly returns the physical tile underlying the passthrough logical tile.

Metadata Operators: The metadata of a logical tile includes information about the underlying physical tiles and a bitmap that represents the rows that must be examined by the operator processing the logical tile. This category of operators only modifies the metadata of the logical tile and not the data that it represents. The *projection* operator modifies the list of attributes in the schema of the input logical tile to remove the attributes that are not needed in the upper-levels of the query plan or in its final result. For the query in Figure 5, the projection operator (π) on top of the sequential scan operator (σ) associated with $\mathcal{R}$ outputs logical tiles that contains the attributes b and c. Another metadata operator is *selection*; this operator modifies the metadata of the input logical tile to mark any row corresponding to a tuple that does not satisfy the predicate as not being part of the logical tile.

Mutators: These operators modify the data stored in the table. The *insert* operator takes in a logical tile, and appends the associated tuples to the specified table. In this case, the operator first reconstructs the tuples represented by the logical tile, and then adds them into the table. It can also directly take tuples from the client and append them to the table.

The *delete* operator takes in a logical tile, and removes the tuples present in the underlying table. It uses the tuple offsets in the first column of the logical tile to identify the locations of the tuples that should be deleted, and then proceeds with their removal. It also supports a truncate mode to quickly erase all the tuples in the table. The *update* operator first removes the tuples present in the logical tile similar to the delete operator. It then constructs the newer version of the tuples by copying over their older version, and performing the requested modifications. Finally, it appends the newer version of the tuples into the table. As we discuss in Section 4, the mutators also control the visibility of tuples for transactions.

Pipeline Breakers: The last category of operators consume the logical tiles produced by their children in the plan tree. They block the execution of the upper-level operators while they wait for their children's output. This is necessary because these operators need to have all of the logical tiles from their children before they can produce their own output. This essentially breaks the streamlined flow of logical tiles between operators during the query execution [37].

The *join* operator takes in a pair of logical tiles, and then evaluates the join predicate over them. It first constructs an output logical tile, whose schema is obtained by concatenating the schemas of the two input logical tiles. When it iterates over each pair of tuples, if it finds a pair satisfying the predicate, then it concatenates them and appends them to the output logical tile. For the query shown in Figure 5, the join operator ($\bowtie$) examines every pair of logical tiles emitted by the projection (π) operators, and then produces a concatenated logical tile containing every pair of logical tile tuples that satisfy the join predicate $\mathcal{R}.b = \mathcal{S}.y$.

Set operators, such as *union* and *intersect*, are also pipeline-breakers. While going over the logical tiles produced by their children, they keep track of the tuples observed. Finally, they emit the logical tiles after marking the tuples that should be skipped by the set operation as not being part of the associated logical tiles.

Similarly, aggregation operators (e.g., *count*, *sum*), examine all the logical tiles from their child to construct the aggregate tuples. Unlike the set operators, these operators build new physical tiles to store the aggregate tuples. For instance, in Figure 5, the aggregate operator (Γ) constructs a physical tile that contain the sum over the attribute z for every group of tuples with a unique value for the attribute c. It then constructs a set of passthrough logical tiles, and propagates them upwards in the plan tree one logical tile at a time.

3.4 Discussion

We now discuss the benefits of the logical tile abstraction for an HTAP DBMS. These include the following:

Layout Transparency: The operators need not be specialized for

[1] A formal algebraic definition of all the operators in our logical tile algebra is presented in Appendix A.

all possible storage layouts because the logical tile algebra hides this information from them. This design obviates the need for expensive merge operations over results produced using different execution engines. It also reduces the DBMS's code complexity, thereby improving maintainability and testability.

Vectorized Processing: The iterator paradigm in many query-processing systems is implemented using executors that process data one tuple at a time [21]. This approach suffers from a high interpretation overhead and prevents key compiler optimizations such as loop pipelining [15]. In contrast, the operators in a tile-based DBMS process data logical tile at a time. As shown in prior research on columnar DBMSs, vectorized processing improves the CPU efficiency by reducing the interpretation overhead [7].

Flexible Materialization: Since neither early materialization nor late materialization is a universally good strategy [6], a tile-based DBMS can dynamically choose to materialize at any operator in the plan tree during the query execution. It can then propagate the passthrough logical tiles upwards in the plan tree. This flexible materialization strategy works well for HTAP workloads.

Caching Behavior: The DBMS can optimize several dimensions in how it creates tile groups, such as the number of tuples per tile group, to ensure that the tiles fit well within the cache hierarchy. Furthermore, the logical tiles' succinct representation enables the DBMS to more easily manage complex intermediate query execution results within the cache.

For the query shown in Figure 5, processing one logical tile at a time reduces the number of cache misses and function calls. The DBMS also copies less data and chases fewer pointers because it only materializes the logical tile in the aggregation operator. Lastly, our logical-tile algebra bridges the theoretical gap between row-stores and column-stores within a single DBMS architecture.

4. CONCURRENCY CONTROL

HTAP workloads are comprised of short-duration transactions that are executed alongside long-running analytical queries. The DBMS must ensure that these OLAP queries do not see the effects of transactions that start after they begin and the readers should not block on writers [41]. It is for this reason that most HTAP DBMSs employ multi-version concurrency control (MVCC) [34, 38, 46].

We now discuss how to use MVCC with our tile-based architecture. We adopt a similar approach taken in previous systems where the DBMS records the versioning information directly alongside the tuple data [30, 46]. When a new transaction is started, the DBMS assigns it a unique *transaction identifier* from a monotonically increasing global counter. When a transaction is ready to commit, the DBMS assigns it a unique *commit timestamp* that it obtains by incrementing the timestamp of the last committed transaction. Each transaction maintains a metadata context that includes: (1) the timestamp of last committed transaction that should be visible to this transaction, and (2) *references* to the set of tuple versions that the transaction either inserts or deletes during its lifetime. A reference contains only the location of the tuple and not its actual data. In our tile-based architecture, it includes the tile group identifier and the offset of the tuple within that tile group. Every tile group contains the following versioning metadata for each tuple:

- **TxnId:** A placeholder for the identifier of the transaction that currently holds a latch on the tuple.
- **BeginCTS:** The commit timestamp from which the tuple becomes visible.
- **EndCTS:** The commit timestamp after which the tuple ceases to be visible.

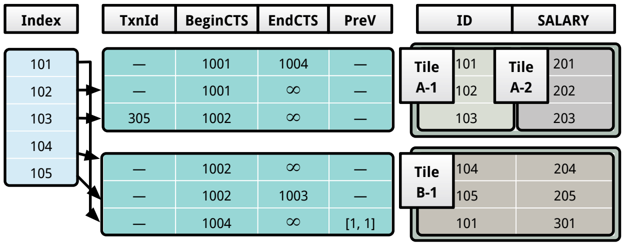


Figure 6: Concurrency Control – Versioning information that the DBMS records in the tile groups for MVCC.

- **PreV:** Reference to the previous version, if any, of the tuple.

Figure 6 shows an example of this versioning metadata. The DBMS stores this information separate from the physical tiles, so that it can handle all the physical tiles in a tile group in the same manner. The system uses the previous version field to traverse the version chain and access the earlier versions, if any, of that tuple. This version chain can span across multiple tile groups, and thus versions may be stored under different physical layouts in memory.

4.1 Protocol

We now discuss the operators in the execution engine that work with the storage manager to ensure transactional isolation. By default, the DBMS provides snapshot isolation.

Mutators: The *insert* operator starts by requesting an empty tuple slot from the DBMS. It then claims that slot by storing its transaction identifier in that tuple's `TxnId` field with an atomic compare-and-swap operation. The `BeginCTS` field is initially set to infinity so that the tuple is not visible to other concurrent transactions. When the transaction commits, the DBMS sets the tuple's `BeginCTS` field to the transaction's commit timestamp, and resets its `TxnId` field. In Figure 6, the first two tuples in the first tile group are inserted by the transaction with commit timestamp `1001`. The DBMS appends all the new tuples in a tile group with the default NSM layout.

The *delete* operator first acquires a latch on the target tuple by storing its transaction identifier in the tuple's `TxnId` field. This prevents another transaction from concurrently deleting the same tuple. At the time of committing the transaction, the DBMS sets the tuple's `EndCTS` to the transaction's commit timestamp, thereby ceasing its existence. In Figure 6, the tuple with `ID 105` is deleted by the transaction with commit timestamp `1003`.

The *update* operator begins with marking the older version of the tuple as invisible. It then constructs the newer tuple version by copying over the older one, and performing the requested modifications. Finally, it appends the newer tuple version into the table. In Figure 6, the transaction with commit timestamp `1004` updates the `SALARY` of the tuple with `ID 101`. Note that the `PreV` field of the third tuple in the second tile group refers to the older version of tuple in the first tile group. At this point in time, the transaction with identifier `305` holds a latch on the tuple with `ID 103`.

Bridge Operators: The only other category of operators that are concerned about the visibility of tuples are the table access methods. The *sequential scan* and *index scan* operators emit one or more logical tiles for the matching tuples. Before predicate evaluation, they first determine if the tuple is visible to their transaction, by verifying whether the transaction's last visible commit timestamp falls within the the `BeginCTS` and `EndCTS` fields of the tuple. The tuple versions that are inserted by the transaction are also visible to these operators. Due to this design, all the other operators of the logical tile algebra are decoupled from the tuple visibility problem.

At any point in time, the DBMS can roll back an uncommitted transaction by resetting the TxnId field of the tuples that it has latched, and releasing any unused tuple slots. Over time, the older versions of a tuple become invisible to any current or future transactions. The DBMS periodically garbage collects these invisible tuple versions in an asynchronous incremental manner. The garbage collection process not only helps in recovering the storage space, but also refreshes the statistics maintained by the query planner.

4.2 Indexes

A tile-based DBMS can use any order-preserving in-memory index (e.g., B+tree) for primary and secondary indexes. While the key stored in the index comprises of a set of tuple attributes, the value is a logical location of the latest version of a tuple. We do not store raw tuple pointers because then the DBMS would have to update them when it reorganizes the layout of the tile groups.

In Figure 6, the index entry for the tuple with ID 101 refers to the third tuple in the second tile group. An operator might encounter a version of the tuple that is not visible to its current transaction. When this occurs, it uses the PreV field to traverse the version chain to find the newest version of the tuple that is visible to the transaction. In this way, the DBMS does not need to redundantly maintain the versioning information in the table as well as in the indexes [30].

4.3 Recovery

Our DBMS architecture includes a recovery module that is responsible for the logging and recovery mechanisms. It employs a variant of the canonical ARIES recovery protocol that is adapted for in-memory DBMSs [12, 32, 35]. During regular transaction execution, the DBMS records the changes performed by the transaction in the write-ahead log, before committing the transaction. It periodically takes snapshots that are stored on the filesystem to bound the recovery time after a crash.

At the start of recovery, the DBMS first loads the latest snapshot. It then replays the log to ensure that the changes made by the transactions committed since the snapshot are present in the database. Changes made by uncommitted transactions at the time of failure are not propagated to the database. As we do not record the physical changes to the indexes in this log, the DBMS rebuilds all of the tables' indexes during recovery to ensure that they are consistent with the database [32]. We leave a detailed discussion of Peloton's recovery module as future work.

5. LAYOUT REORGANIZATION

All of the above optimizations in the design of our FSM-based DBMS are moot unless the system is able to efficiently and effectively reorganize the layout of the database to adapt with the changes in the HTAP workload. The crux of our approach is to track the recent query workload at runtime, and then periodically compute a workload-optimized storage layout for each table in the background. The DBMS then reorganizes the table to match the computed layout.

There are two approaches to perform data reorganization in a DBMS. The first is to combine query processing with data reorganization. In this case, the DBMS copies existing data to new tile groups with a query-optimized layout before processing the data [11]. This approach can be prohibitively expensive for some queries due to the additional I/O overhead.

An alternative is to decouple this reorganization from query execution and use a separate background process to reorganize the data in an incremental manner one tile group at a time. Over time, this process optimizes the storage layout for the workload and amortizes the cost across multiple queries. It is because this approach is incremental [25] that we adopt this second strategy. We now describe the information that the DBMS collects to guide this process.

Algorithm 1 Vertical Partitioning Algorithm

```
Require: recent queries $Q$, table $T$, number of representative queries $k$
  function UPDATE-LAYOUT($Q, T, k$)
      # Stage I : Clustering algorithm
      for all queries $q$ appearing in $Q$ do
          for all representative queries $r_j$ associated with $T$ do
              if $r_j$ is closest to $q$ then
                  $r_j \leftarrow r_j$ + w × (q - $r_j$)
              end if
          end for
      end for
      # Stage II : Greedy algorithm
      Generate layout for $T$ using $r$
  end function
```

5.1 On-line Query Monitoring

The DBMS uses a lightweight monitor that tracks the attributes that are accessed by each query. The goal is to determine which attributes should be stored in the same physical tile in the new tile group layout. The monitor collects information about the attributes present in the query's SELECT clause, as well as those that are present in the WHERE clause [11, 36]. It distinguishes between these two sets of attributes because by co-locating only the attributes in the WHERE clause, rather than those in both clauses together, it needs to retrieve less data for predicate evaluation. It then stores this information as a time series graph for each individual table.

To reduce the monitoring overhead, the monitor only gathers statistics from a random subset of queries. In this case, it needs to ensure that the storage layout is not biased towards the more frequently observed transactions. To improve the overall throughput on HTAP workloads, it needs to optimize the layout for both the transactions and the data intensive analytical queries. The DBMS achieves this by recording the query plan cost computed by the optimizer [3]. It uses this plan cost information to derive a storage layout that also benefits the analytical queries. We next describe how the DBMS uses these query statistics to compute a workload-optimized vertical partitioning for each table.

5.2 Partitioning Algorithm

Computing the optimal storage layout for a workload based on its query statistics requires exponential space and time in the number of attributes. Consider a query workload Q with m queries, and a table with n attributes. For every query $q \in Q$, the naïve partitioning algorithm needs to examine 2^n groups of attributes. It then analyses all the possible attribute partitions of the table. The number of such partitions is referred to as a *Bell number*. Based on the asymptotic limit of the Bell number, the time complexity of this algorithm is $\mathcal{O}(e^{nln(n)} + mn2^n)$ and its space complexity is $\Theta(2^n)$ [19, 23].

As this is infeasible in practice, we adopt a more resource-efficient partitioning algorithm that converges towards the optimal storage layout over time. Our approach leverages the query statistics to compute the layout in two stages. As shown in Algorithm 1, it first employs a clustering algorithm to determine which set of attributes should be stored together in the same physical tile. It then uses a greedy algorithm that uses the output of the clustering algorithm to generate a workload-aware tile group storage layout.

For each table T in the database, the DBMS maintains statistics about the recent queries Q that accessed it. For each $q \in Q$, the DBMS extracts its metadata, such as the attributes it accessed. If the DBMS is able to identify the "important" queries, then it can optimize the storage layout of the table for these queries. It performs this identification using an on-line k-means clustering algorithm and dynamically updates the layout based on incoming queries [27].

For each query q, the clustering algorithm observes the referenced attributes, and assigns it to the j^{th} cluster, whose mean representative query r_j is the most similar to q. The distance metric between two queries is defined as the number of attributes which are accessed by one and exactly one of the two queries divided by the number of attributes in T. Queries that access a lot of common attributes of T, therefore, tend to belong to the same cluster. After assigning the query to a cluster, it updates r_j to reflect the inclusion of the query q. The algorithm prioritizes each query based on its plan cost. If it treats all queries equally, the OLTP queries that tend to outnumber the long-running analytical queries will force the algorithm to quickly converge to a tuple-centric layout. The low projectivity analytical queries do not benefit from the tuple-centric layout. By prioritizing the queries based on their plan cost, the queries with higher I/O cost have a stronger influence on the layout of the table.

An attractive feature of the on-line clustering algorithm is that, over time, the means of the clusters drift towards the recent samples. To better understand this aspect, we now discuss the algorithm more formally. We define c_j to be the mean representative query of the j^{th} cluster, and c_0 to represent that mean's initial value. Each cluster's mean is a vector of numbers whose length equals the number of the attributes in T. If the algorithm adds s query samples to the cluster over time, and prioritizes the older samples with a weight w at the time of addition, then the current mean of the cluster is given by:

$$c_j = (1-w)^s c_0 + w \sum_{i=1}^{s} (1-w)^{s-i} Q_i$$

The weight w determines the rate with which the older query samples are forgotten by the algorithm. The time complexity of each iteration of this algorithm is $\mathcal{O}(mnk)$ with a space complexity of $\mathcal{O}(n(m+k))$. Therefore, it is more efficient than a naïve partitioning algorithm. By using the clustering algorithm, the DBMS maintains the top k representative queries associated with each table in the database.

The next stage in Algorithm 1 is to use a greedy algorithm to derive a partitioning layout for the table using these representative queries. This algorithm iterates over the representative queries in the descending order based on the weight of their associated clusters. For each cluster, the algorithm groups the attributes accessed by that cluster's representative query together into a tile. It continues this process until it assigns each attribute in the table to some tile. In this manner, the DBMS periodically computes a layout for the table using the recent query statistics. We next describe how it reorganizes the table to the new layout.

5.3 Data Layout Reorganization

We use an incremental approach for data layout reorganization. For a given tile group, the DBMS first copies over the data to the new layout, and then atomically swaps in the newly constructed tile group into the table. Any concurrent delete or update operation only modifies the versioning metadata that is stored separate from the physical tiles. The newly constructed tile group refers to the versioning metadata of the older tile group. The storage space consumed by the physical tiles in the older tile group is reclaimed by the DBMS only when they are no longer referenced by any logical tiles. Due to its incremental nature, the overhead of data layout reorganization is amortized over multiple queries.

The reorganization process does not target hot tile groups that are still being heavily accessed by OLTP transactions. Rather, it transforms the cold data to new layouts. This approach works well with the MVCC protocol described in Section 4. The updated versions of the tuples start off in a tuple-centric layout, and then are gradually transformed to an OLAP-optimized layout. Hence, the reorganization process does not impact latency-sensitive transactions while benefiting the OLAP queries.

Some application could have periodic cycles in their workload that oscillate between different sets of queries. In this scenario, it would be better if the DBMS does not quickly reorganize tables to a new layout because then the workload would shift back after an expensive reorganization. This is particularly important in exploratory workloads with ad-hoc OLAP queries. To avoid this problem, the DBMS prioritizes the older query samples in the clustering algorithm with a larger weight, thereby dampening the adaption mechanism. On the other hand, if the workload shift is not ephemeral, then it proceeds with the reorganization. We examine strategies for controlling data reorganization in Section 6.5.

6. EXPERIMENTAL EVALUATION

We now present an analysis of the tile-based FSM architecture. We implemented our execution engine and FSM storage manager in the Peloton DBMS [1]. Peloton is a multi-threaded, in-memory DBMS that is designed for HTAP workloads. Its execution engine supports all the common relational operators, which are implemented using the logical tile abstraction. We also integrated the runtime components for on-line query monitoring and data reorganization described in Section 5.

We deployed Peloton for these experiments on a dual-socket Intel Xeon E5-4620 server running Ubuntu 14.04 (64-bit). Each socket contains eight 2.6 GHz cores. It has 128 GB of DRAM and its L3 cache size is 20 MB.

For each experiment, we execute the workload five times and report the average execution time. All transactions execute with the default snapshot isolation level. We disable the DBMS's garbage collection and logging components to ensure that our measurements are only for the storage and query processing components.

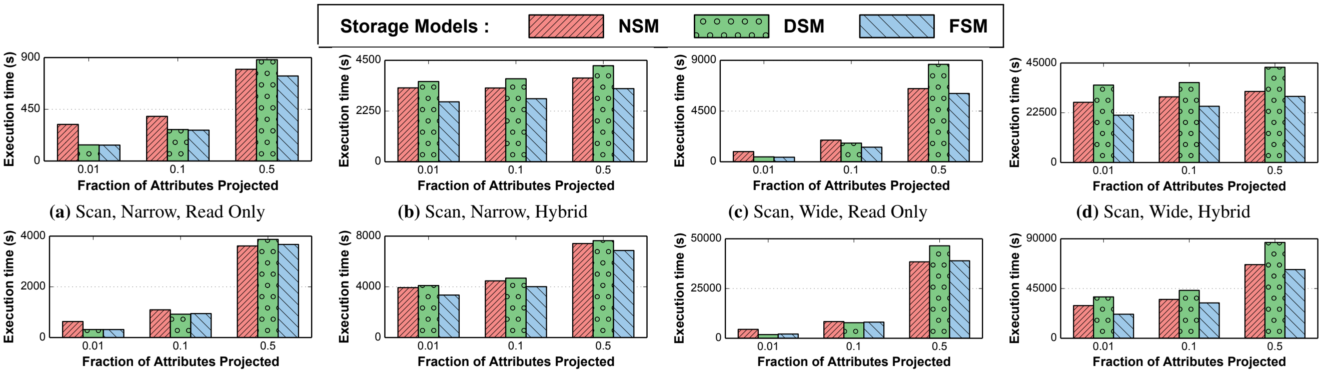
In this section, we begin with an analysis of the impact of query projectivity and selectivity settings on the performance of different storage models. We then demonstrate that a FSM DBMS can converge to an optimal layout for an arbitrary workload without any manual tuning. We then examine the impact of the table's horizontal fragmentation on the performance of the DBMS. We next perform a sensitivity analysis on the parameters of the data reorganization process. Lastly, we compare some of our design choices against another state-of-the-art adaptive storage manager [11].

6.1 ADAPT Benchmark

We first describe the benchmarks we use in our evaluation. Since there currently is a dearth of HTAP workloads for testing, we developed our own that we call the ADAPT benchmark that is composed of queries that are common in enterprise workloads [41]. This benchmark is inspired by the one that Alagiannis et al. developed for evaluating the H_2O adaptive storage manager [11].

The ADAPT database has two tables – a narrow table and a wide table. Each table contains tuples with a primary key (a_0) and p integer attributes ($a_1, \ldots, a_p$), each 4 bytes in size. The narrow table has p = 50 attributes, and the wide table has p = 500 attributes. The size of the tuples in these two tables is approximately 200 B and 2 KB, respectively. In all of our experiments, we first load 10m tuples into each table in the database.

The benchmark's workload consists of (Q_1) an *insert* query that adds a single tuple into the table, (Q_2) a *scan* query that projects a subset of attributes of the tuples that satisfy a predicate, (Q_3) an *aggregate* query that computes the maximum value for a set of attributes over the selected tuples, (Q_4) an *arithmetic* query that sums up a subset of attributes of the selected tuples, and (Q_5) a *join* query that combines the tuples from two tables based on a predicate defined over the attributes in the tables. The corresponding SQL for these queries is as follows:



(**a**) Scan, Narrow, Read Only (**b**) Scan, Narrow, Hybrid (**c**) Scan, Wide, Read Only (**d**) Scan, Wide, Hybrid

(**e**) Aggregate, Narrow, Read Only (**f**) Aggregate, Narrow, Hybrid (**g**) Aggregate, Wide, Read Only (**h**) Aggregate, Wide, Hybrid

Figure 7: Projectivity Measurements – The impact of the storage layout on the query processing time under different projectivity settings. The execution engine runs the workload with different underlying storage managers on both the narrow and the wide table.

Q_1: **INSERT INTO** R **VALUES** $(a_0, a_1, \ldots, a_p)$
Q_2: **SELECT** a_1,a_2,...,a_k **FROM** R **WHERE** $a_0 < \delta$
Q_3: **SELECT** MAX(a_1),...,MAX(a_k) **FROM** R **WHERE** $a_0 < \delta$
Q_4: **SELECT** $a_1 + a_2 + \ldots + a_k$ **FROM** R **WHERE** $a_0 < \delta$
Q_5: **SELECT** X.a_1,...,X.a_k,Y.a_1,...,Y.a_k
FROM R **AS** X, R **AS** Y **WHERE** X.a_i < Y.a_j

Note that different values for k and δ alter the projectivity and the selectivity of the queries, respectively. We use different workloads comprised of these query types to evaluate the impact of the storage models on the performance of the DBMS.

6.2 Performance Impact of Storage Models

We begin with an analysis of the storage models when executing the scan (Q_1), aggregate (Q_2), and insert (Q_3) queries under different projectivity and selectivity settings. We consider two workloads in this experiment: (1) a *read-only* workload with one scan or aggregate query, and (2) a *hybrid* workload comprised of one scan or aggregate query followed by 1m insert queries. For each workload, we first load the database and execute the queries five times till the DBMS reorganizes the tables' layout. This is the ideal scenario for the FSM storage manager. We then execute the workload again using different storage managers and measure their completion time.

Figures 7a and 7b show the results for these two workloads with the scan query under different projectivity settings. For these experiments, we configure the scan query to select all the tuples. We observe that when the projectivity of the scan query is low, the DSM and FSM storage managers execute the read-only workload 2.3× faster than their NSM counterpart. This is because they make better use of memory bandwidth by only fetching the required attributes. As the query's projectivity increases, the performance gap decreases; when half of the attributes are included in the query's output, the DSM storage manager is 21% slower than the other storage managers. We attribute this to the increase in the tuple reconstruction cost. On the hybrid workload, under low projectivity settings, the FSM storage manager outperforms both NSM and DSM by 24% and 33%, respectively. This is because it processes the scan query faster than NSM and the insert queries faster than DSM.

The results in Figures 7c and 7d show that the differences between the storage managers are larger for the wide table. When half of the attributes are projected, DSM is 43% slower than FSM on the read-only workload in Figure 7c, but it is only 19% slower than FSM for the narrow table on the same workload and projectivity setting in Figure 7a. This is because the tuple reconstruction cost increases with wider tuples. This experiment shows that the benefits of FSM are more pronounced for the wide table.

These trends also occur for the aggregate query workloads shown in Figures 7e to 7h. We configure this query to compute the maximum value over all the tuples for the attributes of interest. The magnitude of the performance gaps between the different storage managers is smaller under the low projectivity settings. FSM executes the read-only workload upto 1.9× faster than NSM. This is because the execution engine needs to materialize the logical tiles consisting of the aggregate tuples for all the storage managers.

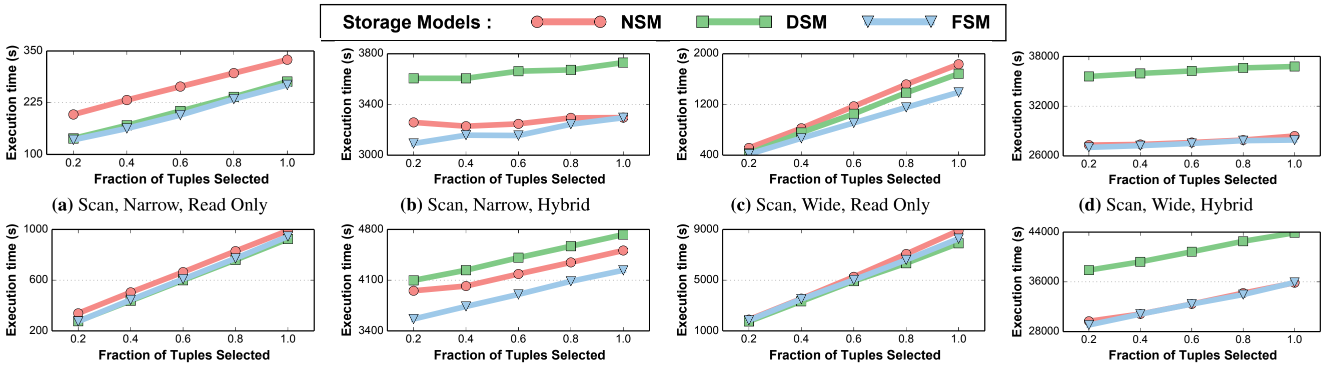
Figure 8 shows the results for the read-only and hybrid workloads under different selectivity settings. For these experiments, we fix the projectivity of the scan and aggregate queries to 0.1 (i.e., the query projects five attributes for the narrow table, and 50 attributes for the wide table). We vary the selectivity of these queries from 10–100%. Across all these settings, the execution time with FSM is either better or comparable to the other storage managers. This highlights the benefits of hybrid storage layouts for HTAP workloads.

We see that DSM outperforms NSM on the read-only workload, but then this order is flipped for the hybrid workload. This is because DSM executes the scan query faster than NSM on both the tables (Figures 8a and 8c). But NSM is faster than DSM on the insert queries (Figures 8d and 8h). FSM outperforms NSM on the read-only workload and DSM on the hybrid workload. This illustrates that the FSM storage manager can adapt to any HTAP workload unlike the DSM and NSM storage managers. In case of the workloads that contain the aggregate query, as shown in Figures 8e to 8h, the difference between the storage managers shrinks. We attribute this to the low projectivity setting of the aggregate query, which corroborates the findings in Figure 7.

6.3 Workload-Aware Adaptation

In the previous experiment, we examined a hybrid storage layout *after* the DBMS optimizes the layout for the workload. In the real world, this is not feasible due to the dynamic nature of workloads. Thus, we now examine the ability of FSM to adapt the storage layout at runtime, and compare it against static NSM and DSM layouts.

In this experiment, we execute a sequence of queries with changing properties. To mimic the temporal locality in HTAP workloads, and to clearly delineate the impact of the DBMS's storage layout on the different queries, the sequence is divided into *segments* of 25 queries that each correspond to a particular query type. This means that the DBMS executes the same type of query (with different input parameters) in one segment, and then switches to another query type in the next segment. We measure the execution time of each query in the sequence on the wide table for the NSM, DSM, and FSM storage managers. We configured the Peloton's reorganization process to speed up the adaptation process for the sake of demonstration.



(a) Scan, Narrow, Read Only **(b)** Scan, Narrow, Hybrid **(c)** Scan, Wide, Read Only **(d)** Scan, Wide, Hybrid

(e) Aggregate, Narrow, Read Only **(f)** Aggregate, Narrow, Hybrid **(g)** Aggregate, Wide, Read Only **(h)** Aggregate, Wide, Hybrid

Figure 8: Selectivity Measurements – The impact of the storage layout on the query processing time under different selectivity settings. The execution engine runs the workload with different underlying storage managers on both the narrow and the wide table.

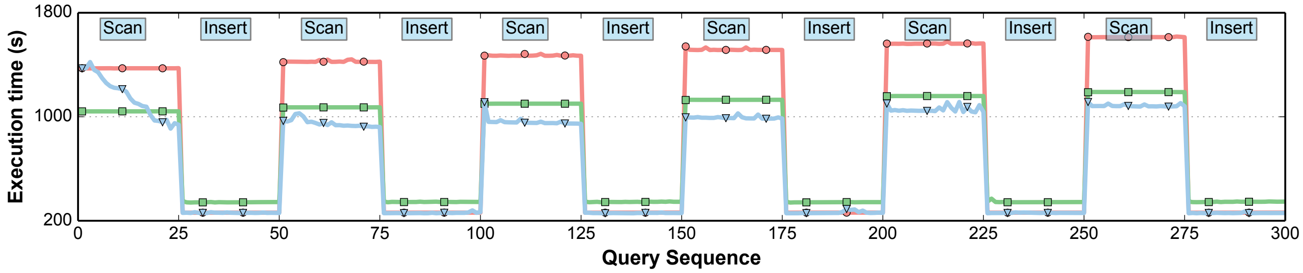


Figure 9: Workload-Aware Adaptation – The impact of tile group layout adaption on the query processing performance in an evolving workload mixture from the ADAPT benchmark. This experiment also examines the behavior of different storage managers while serving different query types in the workload.

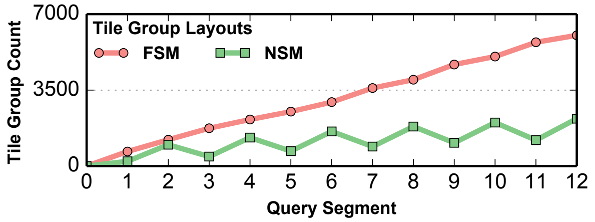


Figure 10: Layout Distribution – The variation in the distribution of tile group layouts of the table over time due to the data reorganization process.

The key observation from the time series graph in Figure 9 is that the FSM storage manager converges over time to a layout that works well for the particular segment. For instance, consider its behavior on the first query segment – a low-projectivity scan query (Q_2). When the tables are loaded into the database, FSM stores all the tuples in the default NSM layout because it has not yet seen any queries. The query execution time is, therefore, comparable to NSM. Over the next few queries, however, it starts reorganizing the data to the hybrid layout $\{\{a_0\}, \{a_1, \ldots, a_k\}, \{a_{k+1}, \ldots, a_{500}\}\}$, which is ideal for Q_2. After this happens, the query execution time drops and matches that of the DSM-based storage manager.

Next, when the workload shifts to the insert query segment (Q_1), we observe that both NSM and FSM fare better than DSM. This is because they perform fewer writes to different memory locations. After the insert queries, there is another shift in the workload back to the scan queries. FSM immediately outperforms NSM because it already reorganized most of the tile groups initially loaded into the table to an OLAP-optimized layout during the first scan query segment. The DBMS takes more time to process the scan queries in the third segment compared to those in the first segment because the table grows in size due to the interleaving insert query segment.

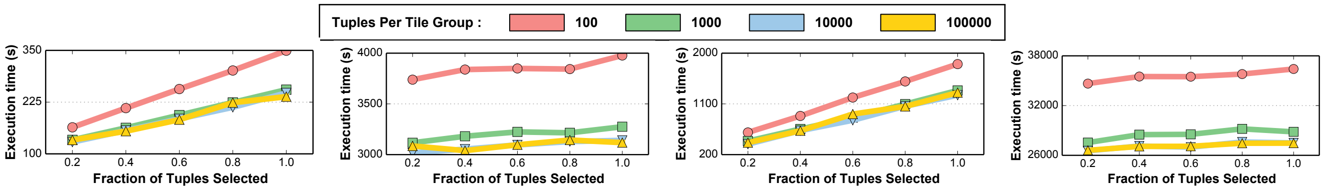
To better understand how the FSM is performing data reorganization, we analyze the distribution of the different tile group storage layouts over time. At the end of every query segment, we compute the number of tile groups per layout. The results shown in Figure 10 indicate that there are only two different layouts – the FSM layout mentioned above and the NSM layout. After every insert query segment, there is an increase in the number of tile groups with NSM layout. This is because the newly added tuples are stored in that layout. Over time, the number of tile groups with the FSM layout increases, as it works well for the scan query segment. This explains the better execution times on those segments in Figure 9.

6.4 Horizontal Fragmentation

We next measure the impact of horizontal fragmentation on the DBMS's performance. The goal is to compare query processing through logical tiles versus the canonical tuple-at-a-time iterator model [21]. We vary the number of tuples that the FSM storage manager stores in every tile group between 10–10000. We then measure the time it takes to execute the workloads comprising of scan queries (Q_2) from Section 6.2.

Figure 11a presents the results for the read-only workload on the narrow table. We observe that the execution time drops by 24% as we increase the number of tuples per tile group from 10 to 1000. We attribute this to the reduction in interpretation overhead. On the hybrid workload shown in Figure 11b, the performance gap reduces to 17%. We attribute this to the writes performed by the insert queries (Q_1) that are not affected by the horizontal fragmentation.

The results for the wide table are shown in Figures 11c and 11d. On the read-only workload, we see that the coarse-grained fragmentation setting outperforms the fine-grained one by 38%. This is because the logical tiles succinctly represent the data stored in these wide tables, and therefore the benefits of vectorized processing are more pronounced. We observe that the performance difference



(a) Scan, Narrow, Read Only **(b)** Scan, Narrow, Hybrid **(c)** Scan, Wide, Read Only **(d)** Scan, Wide, Hybrid

Figure 11: Horizontal Fragmentation – The impact of horizontal fragmentation on the DBMS's performance. We execute the read-only and hybrid workloads comprising of scan queries on the tables in the ADAPT benchmark under different fragmentation settings.

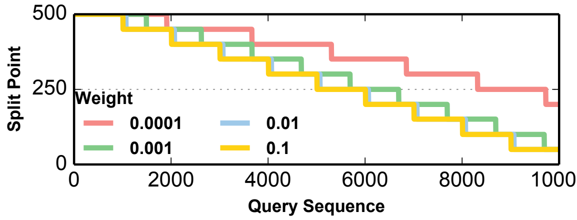


Figure 12: Weight Sensitivity Analysis – The impact of the weight w parameter on the rate at which the clustering algorithm of the data reorganization process adapts with the shifts in the HTAP workload.

tapers off when we increase the number of tuples per tile group from 1000 to 10000. This is likely because the logical tiles no longer fit in the CPU caches beyond this saturation point.

6.5 Reorganization Sensitivity Analysis

We now perform a sensitivity analysis of the parameters of the reorganization process that optimizes the storage layout of the database for the workload. We are interested in how the weight w for older query samples affects the clustering algorithm's behavior. The workload contains a sequence of scan queries (Q_2) on the wide table. The sequence is divided into *segments* of 1000 queries. We gradually reduce the projectivity of the queries from 100% to 10% across the different segments.

As the DBMS executes the workload, it updates the table's layout to the form $\{\{a_0\}, \{a_1, \ldots, a_k\}, \{a_{k+1}, \ldots, a_{500}\}\}$, where k varies with the queries' projectivity in the current segment. We refer to this k as the *split point* in the layout, and we expect that k should decrease from 500 to 50. The rate at which k changes depends on the weight w for older query samples.

Figure 12 shows the fluctuation in the split point as the storage manager processes queries under different w settings ranging from 0.001 to 0.1. When w=0.0001, we observe that the rate at which the algorithm updates the split point is low. This is because the older query samples have a stronger impact on the partitioning layout. On the other hand, when w=0.1, the algorithm closely follows the shift in the workload. In this case, the storage manager reorganizes the data aggressively, and is therefore more susceptible to ephemeral workload shifts. We strike a balance at w=0.001. Under this setting, the storage manager adapts quickly enough for HTAP workloads, but is also not easily swayed over by ephemeral workload shifts.

6.6 Data Reorganization Strategies

Lastly, we compare the key design choices in our system with the H_2O adaptive storage manager [11]. As we described in Section 5, there are two approaches to on-line data reorganization. The first is to combine query processing with reorganization (H_2O), while the other is to reorganize data in an incremental manner (our approach in Peloton). We refer to these two approaches as *immediate* and *incremental* data reorganization strategies. We compare these two strategies using a workload comprising of scan (Q_2) and arithmetic (Q_4) queries on the wide table. We use a sequence of four segments each containing 25 queries. We alternate between these two queries in each query segment.

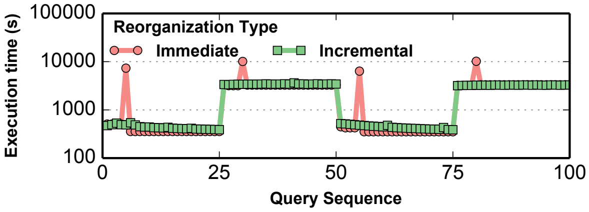


Figure 13: Data Reorganization Strategies – Comparison of the incremental and the immediate data reorganization strategies on a HTAP workload.

Figure 13 presents the time FSM takes to execute each query while adopting the different data reorganization strategies. We observe that when it adopts the former approach, there are spikes in the query latency. This is because when the partitioning algorithm derives a new layout after observing new queries, the storage manager transforms all the tile groups to the new layout within the critical path of query. Although this benefits the subsequent queries in the segment, that query incurs the entire data reorganization penalty. On the other hand, we do not observe such spikes when using the incremental strategy. This is because the reorganization cost is amortized across multiple queries. In this case, the query execution time gradually drops over time due to data reorganization. Thus, we contend that this approach is better for latency-sensitive applications.

Another important design choice is whether the storage manager can maintain multiple copies of the same piece of data with different layouts. H_2O chooses to create copies of the data with new layouts as part of the query execution. This improves the execution time of subsequent queries of the same type. But, it necessitates the construction of layout-specific access operators using code generation techniques. Our storage manager, however, only maintains one copy of the data with a particular layout at any given point in time. We adopt this approach to avoid the overhead of synchronizing the copies on write-intensive workloads.

7. RELATED WORK

We now discuss the previous research on optimizing DBMS storage and execution models for specific workloads, especially in the context of HTAP applications.

Storage Models: Several storage models have been proposed for optimizing the DBMS performance on different workloads. The ubiquitous NSM works well for OLTP workloads [17]. For OLAP workloads, the DBMS's I/O efficiency can be improved by adopting DSM, as it only fetches those columns that are required for query processing [24]. This approach also enhances the caching behavior by increasing the data locality across multiple tuples [14], but incurs high tuple reconstruction costs for OLTP queries. Hybrid NSM/DSM schemes address this trade-off by co-locating attributes accessed together in a query within the same partition [18], figuring out the optimal vertical and horizontal partitioning for a given workload [8], and employing different storage models in different replicas of a table [42].

Ailamaki et al. introduced the PAX model, where all of the data for a single tuple are stored together in a disk page similar to NSM, but it clusters the values of a particular column together for multiple tuples within that page [9, 10]. This helps improve the cache performance and bandwidth utilization by eliminating unnecessary memory references, and thereby reduce the overhead of OLAP queries with low projectivity.

Off-line Physical Design Tuning: Data Morphing generalizes the PAX storage model by decomposing the records into arbitrary groups of attributes [23]. It determines the optimal partitioning for a given workload using a hill-climbing algorithm evaluated over query workload statistics. Zukowski and Boncz show that for complex queries, different storage layouts work well for different parts of the same query plan [51]. MonetDB/X100 improves CPU efficiency by using vectorized primitives for query processing [15]. IBM's Blink horizontally partitions data to improve compression through fixed-length dictionary encoding, which enables efficient hash-based aggregation techniques and SIMD evaluation of predicates [43].

All of this work examines the performance impact of storage models on OLTP and OLAP workloads. They assume, however, that the workload is known a priori, and thus are unable to adapt to dynamic HTAP workloads. In contrast, our approach tunes the storage layout in tandem with the workload to support these applications.

On-line Physical Design Tuning: Database cracking performs index construction and maintenance as a part of regular query processing [25]. It dynamically sorts the tuples in a column-store based on the query workload, and minimizes the tuple reconstruction cost using auxiliary data structures [26]. This approach, however, does not handle co-located attributes. Bruno and Chaudhuri present an online algorithm for selecting indexes [16]. During query execution, DBMS estimates the improvement provided by a candidate index on the given workload and space constraints, and then automatically creates that index if it considers it to be beneficial. Autostore is an on-line self-tuning data store that uses a greedy algorithm for database partitioning [28].

Similar to our approach, this line of work addresses the problem of optimizing the physical design of the database during regular query processing. But all of them focus on auxiliary data structures (i.e., indexes), or they tightly couple the design of the DBMS's execution engine with that of its storage manager.

Hybrid DBMS Architectures: Since the beginning of the 21st century, there have been several DBMSs and add-ons developed for HTAP workloads. One of the first approaches, known as fractured mirrors, is where the DBMS maintains both separate NSM and DSM physical representations of the database simultaneously [42]. This approach has been recently implemented in Oracle's columnar add-on [39]. IBM's BLU is a similar columnar storage add-on for DB2 that uses dictionary compression [44]. Although these systems achieve better ad-hoc OLAP query performance than a pure row store DBMS, the cost of synchronizing the mirrors is high.

The HYRISE DBMS automatically partitions the tables into variable-length vertical segments based on how the attributes of each table are co-accessed by the queries [22]. HYRISE provides better cache utilization than PAX when scanning both narrow and wide projections. Based on the work from HYRISE, SAP developed the HANA in-memory DBMS that uses a split-storage model [31, 47]. Tuples start out in a NSM layout, and then migrate to a compressed DSM storage manager.

Another DBMS that supports dual NSM/DSM storage like HANA is MemSQL [4]. Like HANA, these storage layouts are managed by separate runtime components [5]. But MemSQL is different because the different layouts are not transparent to the application. That is, the administrator manually imports data into the DBMS as a disk-resident, read-only table stored in columnar format and then modifies their application to query those tables.

HyPer is an in-memory hybrid DBMS that stores the entire database in either a NSM or DSM layout (i.e., it cannot use hybrid layouts for tables) [29]. To prevent longer running OLAP queries from interfering with regular transactions, HyPer periodically creates copy-on-write snapshots by forking the DBMS's process and executes those queries on separate CPUs in the forked process. OLTP transactions are executed by the original process, and all OLAP queries are executed on the snapshots. All of the above DBMSs adopt static hybrid storage layouts. That is, they can only optimize the storage layout of the tables for a static workload.

Adaptive Stores: OctopusDB maintains multiple copies of a database stored in different layouts [20]. It uses a logical log as its primary storage structure and then creates secondary physical representations from the log entries. It does not address some problems with this architecture. Foremost is that the DBMS's query planner is unable to generate query plans that could span different representations. The system also incurs high synchronization costs.

H_2O is a hybrid system that dynamically adapts the storage layout with the evolving HTAP workload [11]. It maintains the same data in different storage layouts to improve the performance of read-only workloads through multiple execution engines. It combines data reorganization with query processing. Although Peloton shares many of the goals of the H_2O system, it uses a single execution engine to process data stored in different storage layouts. It performs data reorganization in the background to reduce the impact on query latency. Peloton stores a given tile group in only one storage layout, as it needs to support write-intensive workloads, where the overhead of synchronizing the copies of a tile group is high. It allows tile groups belonging to the same table to have different storage layouts.

8. CONCLUSION

This paper presented a DBMS architecture based on tiles to bridge the architectural gap between the OLTP and OLAP systems. The FSM storage manager stores tables using hybrid storage layouts based on how it expects the tuples to be accessed in the future. It organizes the hotter tile groups of a table in a format that is optimized for OLTP operations, while the colder ones are stored in a format that is more amenable to OLAP queries. We proposed a logical tile abstraction that allows the DBMS to execute query plans over data with these different layouts without using separate execution engines and with minimal overhead. Our on-line reorganization technique continuously enhances each table's physical design in response to an evolving query workload. This enables the DBMS to optimize the layout of a database for an arbitrary application without requiring any manual tuning. Our evaluation showed that the FSM-based DBMS delivers up to 3× higher throughput compared to static storage layouts across different HTAP workloads.

ACKNOWLEDGMENTS

This work was supported (in part) by the Intel Science and Technology Center for Big Data and the U.S. National Science Foundation (CCF-1438955). Thanks to Greg Thain for the paper title idea.

For questions or comments about this paper, please call the CMU Database Hotline at +1-844-88-CMUDB.

References

[1] Peloton Database Management System. http://pelotondb.org.

[2] Linux perf framework. https://perf.wiki.kernel.org/index.php/Main_Page.

[3] PostgreSQL Query Plan Cost. http://www.postgresql.org/docs/9.5/static/using-explain.html.

[4] MemSQL. http://www.memsql.com, 2015.

[5] MemSQL – Columnstore. http://docs.memsql.com/4.0/concepts/columnstore/, 2015.

[6] D. Abadi, D. Myers, D. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. In *ICDE*, 2007.

[7] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD*, 2008.

[8] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, 2004.

[9] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[10] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, 2002.

[11] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A hands-free adaptive store. In *SIGMOD*, 2014.

[12] J. Arulraj, A. Pavlo, and S. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*, 2015.

[13] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, and F. Inc. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, 2010.

[14] P. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[15] P. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[16] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, 2007.

[17] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *SIGMOD*, 1985.

[18] D. Cornell and P. Yu. An effective approach to vertical partitioning for physical design of relational databases. In *IEEE TSE*, 1990.

[19] N. de Bruijn. *Asymptotic Methods in Analysis*. Dover, 1981.

[20] J. Dittrich and A. Jindal. Towards a one size fits all database architecture. In *CIDR*, pages 195–198, 2011.

[21] G. Graefe. Volcano – an extensible and parallel query evaluation system. In *IEEE TKDE*, volume 6, pages 120–135, Feb. 1994.

[22] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: a main memory hybrid storage engine. In *VLDB*, pages 105–116, 2010.

[23] R. A. Hankins and J. M. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *VLDB*, 2003.

[24] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, pages 487–498, 2006.

[25] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.

[26] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.

[27] A. K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 2010.

[28] A. Jindal and J. Dittrich. Relax and let the database do the partitioning online. In *Enabling Real-Time Business Intelligence*, Lecture Notes in Business Information Processing, 2012.

[29] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[30] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. In *VLDB*, 2011.

[31] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka, H. Plattner, J. Krüger, and M. Grund. High-performance transaction processing in SAP HANA. *IEEE Data Eng. Bull.*, 36(2):28–33, 2013.

[32] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *ICDE*, 2014.

[33] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *VLDB Journal*, 2000.

[34] MemSQL. How MemSQL Works. http://docs.memsql.com/4.1/intro/.

[35] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1):94–162, 1992. ISSN 0362-5915.

[36] S. B. Navathe and M. Ra. Vertical partitioning for database design: A graphical algorithm. In *SIGMOD*, 1989.

[37] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[38] T. Neumann, T. Mühlbauer, and A. Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*, 2015.

[39] Oracle. Oracle database in-memory option to accelerate analytics, data warehousing, reporting and OLTP. http://www.oracle.com/us/corporate/press/2020717, 2013.

[40] M. Pezzini, D. Feinberg, N. Rayner, and R. Edjlali. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. https://www.gartner.com/doc/2657815/, 2014.

[41] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD*, 2009.

[42] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 430–441, 2002.

[43] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *ICDE*, 2008.

[44] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. In *VLDB*, volume 6, pages 1080–1091, 2013.

[45] A. Rosenberg. Improving query performance in data warehouses. *Business Intelligence Journal*, 11, Jan. 2006.

[46] D. Schwalb, M. Faust, J. Wust, M. Grund, and H. Plattner. Efficient transaction processing for Hyrise in mixed workload environments. In *IMDM*, 2014.

[47] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: The end of a column store myth. In *SIGMOD*, pages 731–742, 2012.

[48] V. Sikka, F. Farber, A. Goel, and W. Lehner. SAP HANA: The evolution from a modern main-memory data platform to an enterprise application platform. In *VLDB*, 2013.

[49] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. VLDB, pages 553–564, 2005.

[50] A. H. Watson, T. J. Mccabe, and D. R. Wallace. Structured testing: A software testing methodology using the cyclomatic complexity metric. In *U.S. Department of Commerce/National Institute of Standards and Technology*, 1996.

[51] M. Zukowski and P. A. Boncz. Vectorwise: Beyond column stores. In *IEEE Data Engineering Bulletin*, 2012.

APPENDIX

A. LOGICAL TILE ALGEBRA

We formally define the semantics of the operators in logical tile algebra. For each operator, we describe the type of the input arguments and output, as well as an algebraic definition of the transformation from the input arguments to the output. We denote a multi-set of tuples that can contain duplicates by $\langle \ldots \rangle$, and a set of tuples where all the elements are unique by $\{\ldots\}$. $|S|$ represents the size of a multi-set or set S. Given this notation, we describe the operators based on the logical tile algebra shown in Table 1.

Category	Operator	Input Arguments	Output
Bridge Operators	SEQUENTIAL SCAN	T X, predicate P	$\{$ LT $\} \equiv \{$ LT w $\mid$ w $\equiv \langle\ x \mid x \in X \wedge P(x)\ \rangle\ \}$
	INDEX SCAN	I X, predicate P	$\{$ LT $\} \equiv \{$ LT w $\mid$ w $\equiv \langle\ x \mid x \in X \wedge P(x)\ \rangle\ \}$
	MATERIALIZE	LT X	PT Y
Metadata Operators	SELECTION	LT X, predicate P	LT $\equiv \langle\ x \mid x \in X \wedge P(x)\ \rangle$
	PROJECTION	LT X, attributes C	LT $\equiv X' \mid$ schema$(X') = C$
Mutators	INSERT	T X, LT Y	T $\equiv$ X $\Rightarrow X \vee Y_p$
	DELETE	T X, LT Y	T $\equiv$ X $\Rightarrow X \setminus Y_p$
	UPDATE	T X, attributes C, expr E	T $\equiv$ X $\Rightarrow (X \setminus Y_p) \vee Z_p$
Pipeline Breakers	JOIN	LT X, LT Y, predicate P	LT $\equiv \langle\ x \mid\mid y \mid x \in X \wedge y \in Y \wedge P(x, y)\ \rangle$
	UNION	LT X, LT Y	LT $\equiv \{\ z \mid z \in X \vee z \in Y\ \}$
	INTERSECTION	LT X, LT Y	LT $\equiv \{\ z \mid z \in X \wedge z \in Y\ \}$
	DIFFERENCE	LT X, LT Y	LT $\equiv \{\ z \mid z \in X \wedge z \notin Y\ \}$
	UNIQUE	LT X	LT $\equiv \{\ z \mid z \in X\ \}$
	COUNT	LT X	int $\equiv \mid X \mid$
	SUM	LT X, attributes C	LT $\equiv [\ \sum_{x \in X} x_c\ \forall\ c \in C\ ]$
	MAX	LT X, attributes C	LT $\equiv [\ \max_{x \in X} x_c\ \forall c \in C\ ]$

Table 1: Logical Tile Algebra – Algebraic definitions for the operators of the logical tile algebra. For each operator, we describe its category, input arguments, output, and the algebraic transformation. We denote the logical tile by **LT**, the physical tile by **PT**, the table by **T**, and the index by **I**.

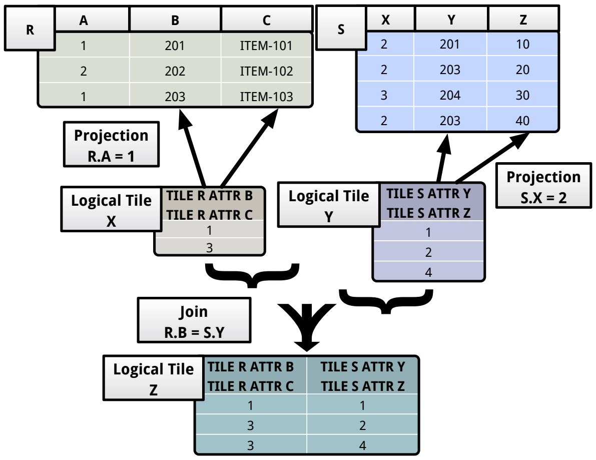


Figure 14: Logical Tile Operators – The runtime behavior of the sequential scan and join operators of the logical tile algebra for the sample query.

Bridge Operators: The bridge operators include the table access methods, such as the sequential scan and the index scan operators. The *sequential scan* operator generates a logical tile for every tile group in the table X that contains any tuple x satisfying the predicate $P(x)$. The output logical tile only contains one column, that is a list of offsets corresponding to the matching tuples in the tile group. Figure 14 shows the runtime behavior of the sequential scan operator in the plan tree presented in Figure 5. Here, the operator emits a logical tile X that represents the tuples in the physical tile R that satisfy the predicate a=1. The *index scan* operator identifies the tuples matching the predicate P using the index X, and then constructs one or more logical tiles representing those tuples. Each logical tile can only represent the matching tuples that are present within the same tile group.

The *materialize operator* transforms the logical tile X to a physical tile Y. The execution engine invokes this operator either before sending the result tuples to the client, or when it needs to perform early-materialization. In the latter case, it constructs a passthrough logical tile to wrap around the materialized tile Y, and passes it upwards in the plan tree.

Metadata Operators: The *projection* operator takes in a logical tile X, only modifies its metadata, and emits it as the output. It removes the attributes that need to be projected away from the metadata. The columns corresponding to these attributes are still present in the logical tile. The *selection* operator marks the tuples that do not satisfy the predicate P as invisible in the metadata of X. The rows corresponding to these tuples are still present in X. Due to these optimizations, these operators only need to alter the metadata of the logical tile.

Mutators: These operators directly modify the data stored in the table, and we therefore represent the transformation by $\Rightarrow$. The *insert* operator takes in a logical tile Y, and appends the associated tuples Y_p to the specified table X. The *delete* operator removes the tuples Y_p represented by Y from the table X. The *update* operator first marks the tuples Y_p represented by Y in the table X as dead. It then evaluates the expressions E associated with the attributes C over Y_p to construct newer versions of those tuples Z_p. Finally, it appends Z_p in the table. The mutators work with the transactional storage manager to control the lifetime of tuples.

Pipeline Breakers: These operators break the streamlined flow of logical tiles between them during query execution [37]. The *join* operator takes in logical tiles X and Y, and evaluates the join predicate $P(x, y)$ over them. It constructs an output logical tile by concatenating the schemas of X and Y. As iterates over every pair of tuples, if it finds a one that satisfies P, it concatenates them and appends them to the output logical tile. In Figure 14, we note that the output logical tile Z is obtained by concatenating the matching tuple pairs from the input logical tiles X and Y.

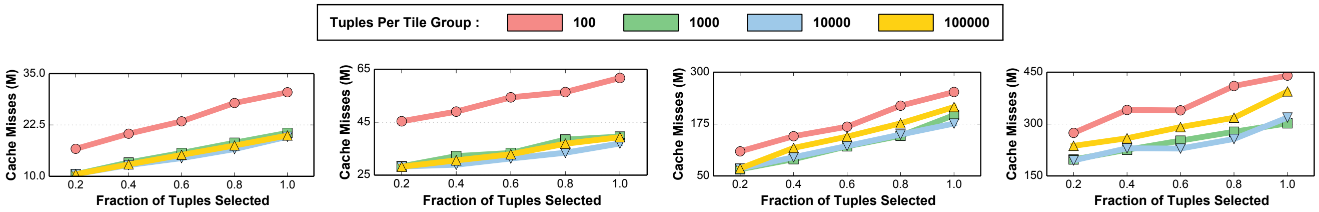
The *set* operators are also pipeline breakers. For instance, the *union* operator keeps track of the tuples that it observes, while going over all the input logical tiles from its children. Finally, it returns the logical tiles after marking the duplicate tuples as invisible.

The *aggregate* operators, such as the *count* and *sum* operators, examine all the logical tiles from the child operator to construct the aggregate tuples. Unlike the set operators, the aggregate operators build new physical tiles to store the aggregate tuples. Finally, they construct a set of passthrough logical tiles to wrap around these newly constructed physical tiles as depicted by [. . .], and propagate them upwards in the plan tree one logical-tile at a time.

We note that the logical tile abstraction has no limitations with respect to its expressiveness. Logical tiles are a "shorthand" notation for describing a set of tuples stored in one or more relations. Any relational operator can therefore be mapped to logical tile algebra.

B. JOIN QUERIES

In this section, we examine the performance impact of storage

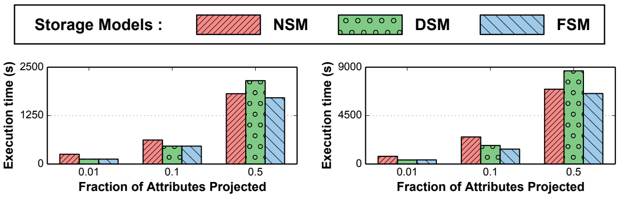


(a) Narrow, Read Only **(b)** Narrow, Hybrid **(c)** Wide, Read Only **(d)** Wide, Hybrid

Figure 15: Caching Behavior – Evaluation of the impact of horizontal fragmentation on caching behavior. We execute the read-only and hybrid workloads on the tables in the ADAPT benchmark under different horizontal fragmentation settings.

Number of Logical Tiles	Materialization Time(s)
1	17.7
10	18.3
100	25.7
1000	82.3
10000	640.4

Table 2: Indirection Overhead – Time taken to materialize 10000 tuples depending on the number of logical tiles representing them.



(a) Narrow Table **(b)** Wide Table

Figure 16: Join Measurements – The impact of the storage layout on the time taken to execute join queries under different projectivity settings. The execution engine runs the workload with different underlying storage managers on both the narrow and the wide table.

models on join queries. Our layout-agnostic join algorithms operate on logical tiles and compute the join using only the join keys. This approach is similar to cache-conscious join algorithms, like radix join [33], that evaluate the join by accessing only the attributes involved in the join. As such, the data layout does not significantly affect the join predicate evaluation [11]. However, the data layout does impact the post-join tuple reconstruction, wherein the storage manager needs to retrieve the projected attributes. The overall behavior of the join operator, while processing data stored under different layouts, is similar to that of the projection operator.

We demonstrate this using the self join query Q_5. We load the table R with 10m tuples. We then measure the time it takes to execute the join query Q_5 on the table. We vary the projectivity of the join query from 1% to 100%. We pick two random attributes a_i and a_j to construct the join predicate.

Figure 16a presents the results for the join query on the narrow table. We observe that when the projectivity of the join query is high, the NSM and FSM storage managers execute the join query 1.3× faster than their DSM counterpart. This is because these layouts reduce the cost of retrieving the projected attributes during tuple reconstruction. The results for the wide table are shown in Figure 16b. In this case, the NSM and FSM storage managers outperform the DSM storage manager by 1.5×. These results are similar to those observed in the projectivity experiment in Section 6.2.

C. CACHING BEHAVIOR

We next examine the impact of horizontal fragmentation on the DBMS's cache locality. In this experiment, we evaluate the reduction in the number of CPU cache misses due to the logical-tile-at-a-time processing model compared to the tuple-at-a-time iterator model [21]. We vary the number of tuples that the FSM storage manager stores in every tile group between 10–10000. We then count the number of cache misses incurred while executing the workloads from Section 6.2 under different horizontal fragmentation settings. We use event-based sampling in the *perf* framework [2] to track the caching behavior. We start this profiling after loading the database.

Figure 15a presents the results for the read-only workload on the narrow table. We observe that the number of cache misses drops by 1.5× when we increase the number of tuples stored in a tile group from 10 to 1000. We attribute this to the reduction in interpretation overhead. A similar drop in the number of cache misses is seen with the hybrid workload results shown in Figure 15b.

The results for the wide table are shown in Figures 15c and 15d. On the read-only workload, we see that the number of cache misses under the coarse-grained fragmentation setting is 1.7× smaller than that under the fine-grained setting. This shows that the benefits of the succinct representation of data by logical tiles are more pronounced for wide tables. We observe that the number of cache misses is higher when we increase the number of tuples per tile group from 1000 to 10000. This is because the logical tiles no longer fit in the CPU caches beyond this saturation point.

D. INDIRECTION OVERHEAD

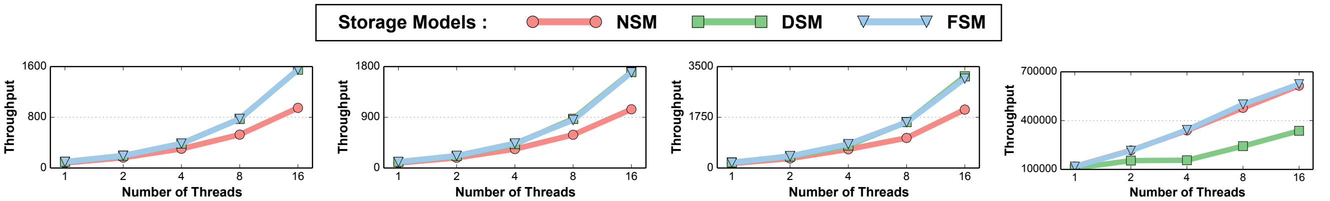
We now focus on the performance overhead associated with the level of indirection in the logical tile abstraction to access the underlying physical tiles. To materialize a logical tile, the storage manager needs to look up the locations of the underlying physical tiles only once for all the tuples contained in the logical tile.

Table 2 presents the time taken to materialize 10000 tuples from the wide table that are represented by one or more logical tiles. We vary the number of tuples represented by each logical tile from 1 through 10000. When every logical tile represents only one tuple, the storage manager materializes 10000 such logical tiles. On the other hand, when the logical tile represents all the 10000 tuples, then the storage manager only materializes one such logical tile. We observe that the materialization time drops by 8× when the number of tuples represented by a logical tile increases from 1 to 10. As HTAP workloads frequently contain queries that produce longer logical tiles, we think that the indirection overhead will be tolerable.

An alternative approach to handle HTAP workloads is to use two separate OLTP and OLAP DBMSs [47]. However, the overhead of an atomic commitment protocol to coordinate transactions that span across data stored in these two different systems is higher than the overhead of indirection associated with the logical tile abstraction.

E. CONCURRENT HYBRID WORKLOADS

In this section, we examine the impact of storage layout on the performance of concurrent HTAP workloads. The workload consists of a random mixture of scan queries (Q_2) and insert queries (Q_1). The selectivity of the scan queries is set to 10% of all the tuples in the table. We configure the projectivity of the scan queries to 10%



(a) Read Only **(b)** Read-heavy **(c)** Balanced **(d)** Write-only

Figure 17: Concurrent Hybrid Workloads – The impact of the storage layout on the query processing time under different concurrent hybrid workloads. The execution engine runs the workloads with different underlying storage managers on the narrow table.

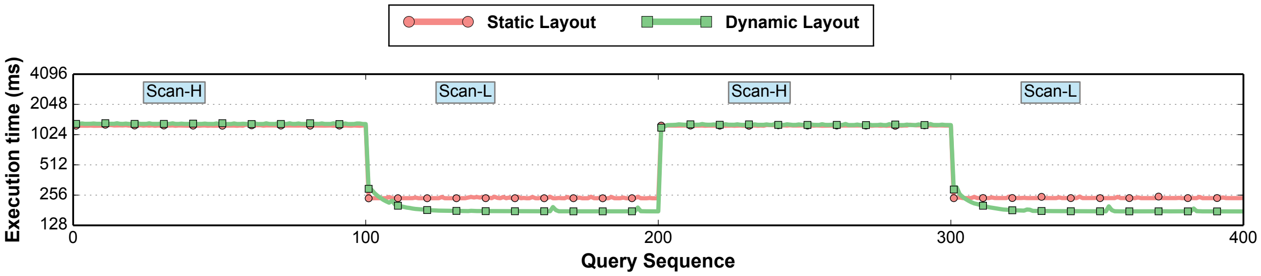


Figure 18: Dynamic Layout Adaptation – Benefits of dynamic layout adaptation on a HTAP workload.

of the attributes in the table. All the storage managers take more time to execute the scan query compared to the insert query. We examine four types of workload mixtures that allow us to vary the I/O operations that the DBMS executes. These mixtures represent different ratios of scan and insert queries:

- **Read-Only:** 100% *scans*
- **Read-Heavy:** 90% *scans*, 10% *inserts*
- **Balanced:** 50% *scans*, 50% *inserts*
- **Write-Heavy:** 10% *scans*, 90% *inserts*

We measure the throughput delivered by each storage manager on the different workload mixtures. On each workload mixture, we vary the number of concurrent clients from 1 through 16. We perform this experiment on a server with eight cores. Figure 17a presents the results on the read-only mixture. We observe that the throughputs delivered by the FSM and DSM storage managers are 1.7× higher than that of the NSM storage manager under higher concurrency settings. We attribute this to their better usage of memory bandwidth by only fetching the attributes required by the scan queries. We observe that the throughput scales linearly with the number of client connections. We notice similar trends on the read-heavy and balanced mixtures in Figures 17b and 17c.

The trend is reversed on the write-heavy mixture shown in Figure 17d. On this workload, FSM and NSM outperform DSM because they perform fewer writes to different memory locations while handling the insert queries. The absolute throughput is more than 400× higher than that observed on the read-only mixture. This is due to the lower overhead incurred by the storage manager while executing an insert query.

F. DYNAMIC LAYOUT ADAPTATION

We evaluate the benefits of dynamic layout adaptation on ad-hoc HTAP workloads in this section. Hyrise partitions the tables into vertical segments based on how the attributes of each table are co-accessed by the queries [22]. It assumes, however, that the access patterns are known a priori, and thus it generates a *static* storage layout for every table. Unlike Hyrise, Peloton dynamically adapts the storage layout based on the changes in the HTAP workload.

We consider an HTAP workload comprising of scan queries (Q_2) with high projectivity on the wide table in the ADAPT benchmark. We configure the projectivity to be 90% of the attributes. Hyrise computes an optimal storage layout for this workload where all the projected attributes are stored in a single physical tile.

In order to emulate a dynamic HTAP workload, we invoke *ad-hoc* query segments comprising of scan queries (Q_2) with low projectivity. These queries project only 5% of the attributes. Each query segment consists of 100 queries. This means that the DBMS executes the high projectivity scan queries in one segment, and then switches to the low projectivity scan queries in the next segment. We measure the execution time of each query in the sequence when we use the static layout computed by Hyrise and the dynamic layouts generated by the FSM storage manager.

The key observation from the time series graph in Figure 18 is that the FSM storage manager converges over time to a layout that works well for the particular segment. For instance, in the first query segment, it performs similar to Hyrise on the high projectivity scan queries. Next, when the workload shifts to the low projectivity query segment, we observe that it outperforms the static layout computed by Hyrise by 1.5×. This is because the layout reorganization process dynamically adapts the layout to work well for this segment.

G. FUTURE WORK

Peloton contains several knobs for controlling the layout reorganization process. Although we attempt to provide good default settings for these knobs, we believe that the DBMS should automatically adjust these knobs based on the HTAP workload. We plan to investigate the design of a self-driving module within the DBMS that dynamically adjusts these knobs to simplify the tuning process. We intend to explore code generation and data compression techniques for optimizing query execution as well as other facets of the DBMS's runtime operations.