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

Paschal Igusti & Kamoltat Sirivadhna

Agenda

- Motivation
 - Storage Model
 - Why FSM
 - Tile Based Architecture
 - Concurrency Control
 - Layout Reorganization
 - Experimental Evaluation
 - Conclusion
-

Hybrid Transaction-Analytical Processing

- Organizations need to transform **fresh and historical** data into critical insights
 - Hybrid Transaction-Analytical Processing (HTAP)
 - Analyze a combination of historical data-sets and real-time data
 - Data has immense value as soon as it is created, but diminishes over time
 - **What are some examples of such cases?**
-

HTAP Pipelines

- Many organizations implement HTAP pipelines using separate DBMSs
 - One for transactions and new information (OLTP DBMS)
 - The other for analytical queries (OLAP DBMS)
 - **Many inherent problems** with this implementation
 - Too much time to propagate changes between separate systems
 - Administrative overhead for deploying and maintaining is too heavy
 - Requires a query for multiple systems to combine data from different databases
-

Ideally...

- Use a single HTAP DBMS to support OLTP workloads and OLAP queries to operate on transactional and historical data
 - **Key problem:** executing OLAP workloads that **access old and new data** while **simultaneously executing transactions and updates** the database
 - Separate engines:
 - OLTP: Engine for row-oriented data
 - OLAP: Engine for column-oriented data
-

Bridging the Archipelago

- Cobbling systems together leads to increased complexity and degraded performance
 - This paper presents a method to bridge the architectural gap between OLTP and OLAP systems using a **unified architecture**
 - Store tables using hybrid layouts based on how the DBMS expects the tuples to be accessed in the future
-

Storage Model

What are the different types of Storage Model mentioned in the paper? And what are the pros and cons?

- N-ary Storage
 - Good for transactional queries
 - Bad for analytic queries
- Decomposition Storage
 - Good for analytic queries
 - Bad for transactional queries
- Flexible Storage
 - Good for analytic & transactional queries
 - Need to have a good understanding of the attributes.
(without online query)

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)

**When should we
use FSM?**

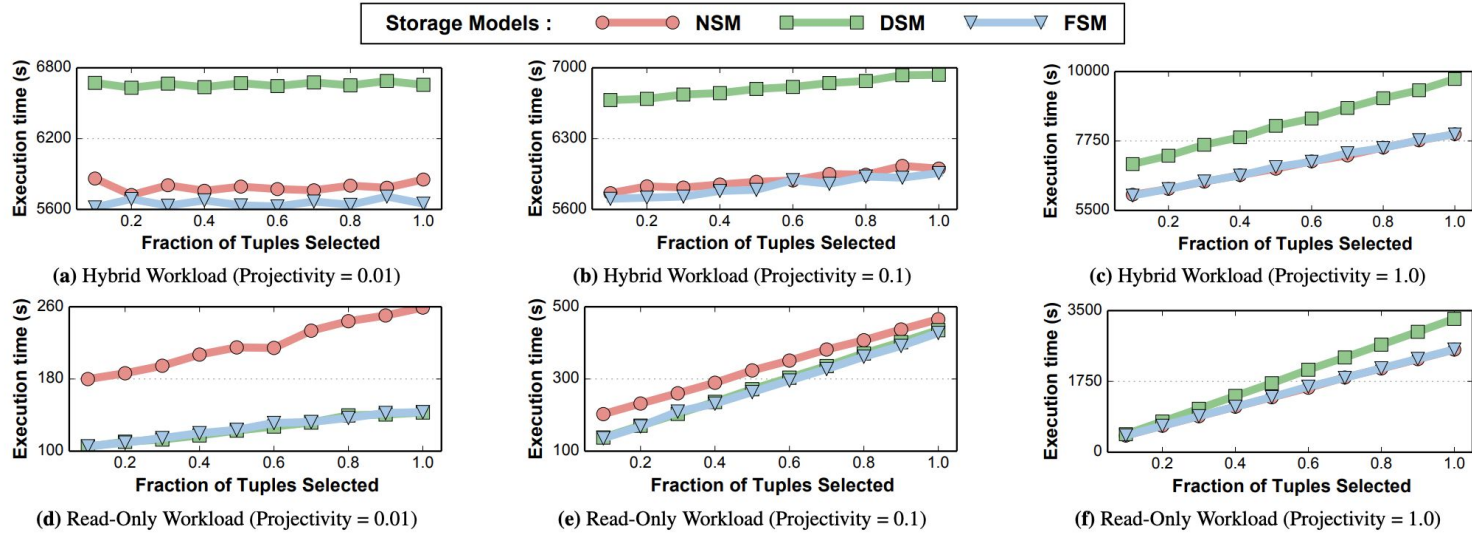


Figure 2: Performance Impact of the Storage Models – Time taken by the execution engine to run (1) a hybrid, and (2) a read-only workload on top of the NSM, DSM, and FSM storage managers.

Q_1 : INSERT INTO R VALUES (a_0, a_1, \dots, a_{500})

Q_2 : SELECT a_1, a_2, \dots, a_k FROM R WHERE $a_0 < \delta$

Tile-Based Architecture

What is a Physical Tile?

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.

ID		IMAGE-ID		NAME		PRICE		DATA	
Tile A-1	101	201	ITEM-101	Tile A-2	10	DATA-101		Tile Group A	
	102	202	ITEM-102		20	DATA-102			
	103	203	ITEM-103		30	DATA-103			
Tile B-1	104	204	Tile B-2	ITEM-104	40	Tile B-3	DATA-104	Tile Group B	
	105	205	ITEM-105	50	DATA-105				
	106	206	ITEM-106	60	DATA-106				
Tile C-1	107	207	ITEM-107	70	DATA-107		Tile Group C		
	108	208	ITEM-108	80	DATA-108				
	109	209	ITEM-109	90	DATA-109				
	110	210	ITEM-110	100	DATA-110				

Tile-Based Architecture cont.

What is a Logical Tile?

Represents values spread across a collection of physical tiles from one or more tables. This abstraction hides the specifics of the layout of the table from its execution engine, without sacrificing the performance benefits of a workload-optimized storage layout.

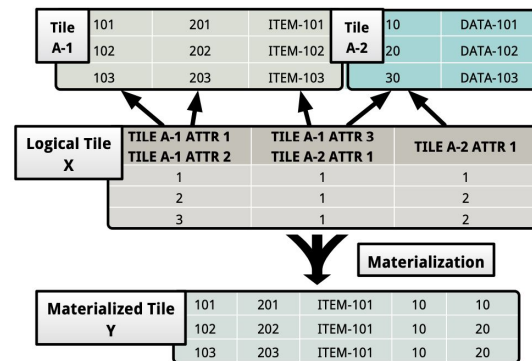


Figure 4: Logical Tile – An example of a logical tile representing data spread across a couple of physical tiles (A-1, A-2).

Tile-Based Architecture cont.

What is Logical Tile Algebra? And what are some of the operators mentioned in the paper?

- Bridge Operators
 - Sequential Scan & Index Scan
- Metadata Operators
 - Projection & Selection
- Mutators
 - Insert & Delete & Update
- Pipeline Breakers
 - Join & Union & Intersect

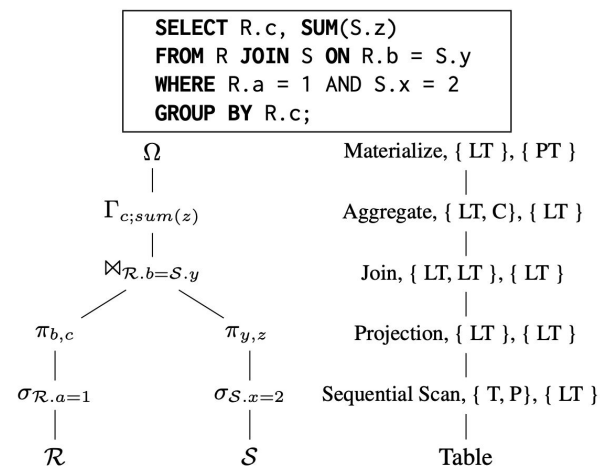


Figure 5: Sample SQL query and the associated plan tree for illustrating the operators of the logical tile algebra. We describe the name, inputs, and output of each operator in the tree. We denote the logical tile by LT, the physical tile by PT, the table by T, the attributes by C, and the predicate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

Tile-Based Architecture cont.

What are the benefits of the logical tile abstraction for an HTAP DBMS?

- Layout Transparency
 - Vectorized Processing
 - Flexible Materialization
 - Caching Behavior
-

Concurrency Control

What does the columns in the table represent?

- 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.
 - PreV: Reference to the previous version, if any, of the tuple.
- (the tile group identifier and the offset of the tuple within that tile group)

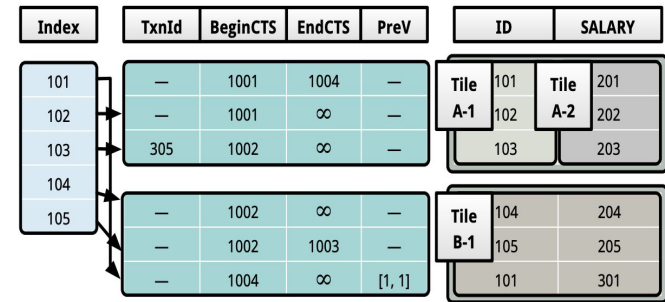


Figure 6: Concurrency Control – Versioning information that the DBMS records in the tile groups for MVCC.

Concurrency Control: Indexes

- Uses B+ trees as the data structure to store primary and secondary indexes
- Uses the PreV field to traverse the version chain to find the newest version of the tuple that is visible to the transaction.
- Index holds a logical location of the latest version of a tuple, they do not store raw tuple pointers since DBMS needs to update during reorganization if store raw tuple

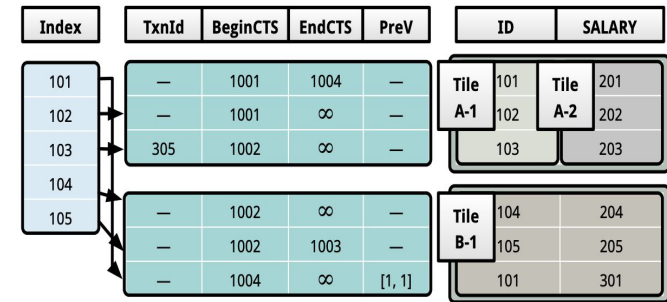


Figure 6: Concurrency Control – Versioning information that the DBMS records in the tile groups for MVCC.

Concurrency Control: Recovery (Future work discussion)

- Uses ARIES recovery protocol
 - Records the changes performed by the transaction in the write-ahead log, before committing the transaction.
 - Periodically takes snapshots that are stored on the filesystem to bound the recovery time after a crash.
 - Does not record the physical changes to the indexes in the log.
 - The DBMS rebuilds all of the tables' indexes during recovery to ensure that they are consistent with the database
-

Layout Reorganization

- All previously mentioned optimizations of the FSM-based DBSM are debatable without smart layout reorganization methods
- Two phase vertical partitioning algorithm: **Clustering** & **Greedy Algo**

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**

$r_j \leftarrow r_j + w \times (q - r_j)$

end if

end for

end for

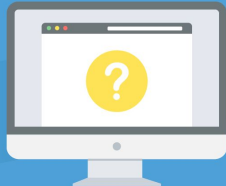
Stage II : Greedy algorithm

 Generate layout for T using r


end function

On-line Query Monitoring

- DBMS uses lightweight monitor that **tracks attributes** that are accessed by each query
- Need to determine **which attributes should be stored in the same physical tile** in new layout
- Collects information about attributes present in **SELECT** and **WHERE** clauses



Component	SELECT	SHOW	TIME
Core	43		0.0211
Plugin: icit-auditlogger			
Plugin: menu-recent-pages	11	1	0.0017
Plugin: ssl-helper	2		0.006



On-line Query Monitoring cont.

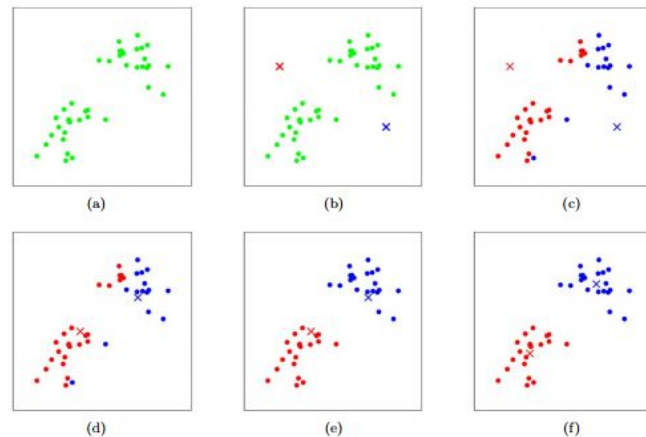
- Stores the information as a time series graph for each individual table
 - Monitor only gathers statistics from **random subset** of queries
 - **Non-biased** towards more frequently observed transactions
 - **Reduces monitor overhead**
-

Clustering Queries & Attributes

- For each table T in the database, the DBMS maintains statistics about queries Q
 - For each $q \in Q$, the DBMS extracts the attributes the query accessed via the metadata
 - The DBMS identifies “important” attributes via k-means clustering
-

Clustering cont.

- For each query q , the clustering algorithm assigns it to the j th cluster, whose mean representative is r_j
- Computes distance metric by the number of unique attributes accessed by two queries divided by the number of attributes in T (disjoint set)
 - Similar queries are part of the same cluster
- Updates r_j to reflect inclusion of q



Clustering cont.

- Algorithm prioritizes each query based on its plan cost
 - Queries with higher I/O cost have stronger influence on layout of table
- Means of clusters drift towards recent samples over time
- c_j is the mean representative query of the j th cluster
 - Vector with length # of attributes in T
- c_0 represents the mean's initial value
- s represents the number of query samples
- w is the weight that determines the rate with which the older query samples are forgotten (higher weights given to older queries)

Runtime

Complexity:

$O(m * n * k)$

Space

Complexity:

$O(n * (m + k))$

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

Greedy Algorithm

- Next phase in algorithm is to use a **greedy algorithm** to derive a partitioning layout for the table using the top k representative queries
 - Iterates over queries in **descending order based on weight** of associated cluster
 - For each cluster, algorithm **groups attributes accessed by representative query into one tile**
 - Continues until each attribute is assigned to a tile
-

Data Layout Reorganization

- Use an incremental approach for data layout reorganization
 - For a given tile group, DBMS **copies over the data to the new layout**
 - **Atomically swaps** in the newly constructed tile group into the table
 - Storage space consumed by physical tiles in old tile group is reclaimed by the DBMS **only when they are no longer referenced by any logical tiles**
-

Data Layout Reorganization cont.

- Reorganization process **does not target hot (transactional) tile groups** that are still being heavily accessed by OLTP transactions
 - Transforms **apply to cold (historic) data**
 - Updated versions of tuples **start off in a tuple-centric layout** (similar to row-store)
 - **Gradually transformed to OLAP-optimized** (similar to column-store) layout
-

Experimental Evaluation

What happened in the experiment?

- 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.
 - Execute the workload five times and report the average execution time. Disable the DBMS's garbage collection and logging components.
 - FSM DBMS can converge to an optimal layout for an arbitrary workload without any manual tuning.
-

Experimental Evaluation cont.

What's ADAPT Benchmark?

- Narrow Table 50 attributes and Wide Tables 500 attributes with a approximate size of 200B with 2KB
- Contains 5 Queries
- 2 Work loads: Read only and Hybrid

Q_1 : **INSERT INTO R VALUES** (a_0, a_1, \dots, a_p)

Q_2 : **SELECT** a_1, a_2, \dots, a_k **FROM R WHERE** $a_0 < \delta$

Q_3 : **SELECT** $\text{MAX}(a_1), \dots, \text{MAX}(a_k)$ **FROM R WHERE** $a_0 < \delta$

Q_4 : **SELECT** $a_1 + a_2 + \dots + a_k$ **FROM R WHERE** $a_0 < \delta$

Q_5 : **SELECT** $X.a_1, \dots, X.a_k, Y.a_1, \dots, Y.a_k$
FROM R AS X, R AS Y WHERE $X.a_i < Y.a_j$

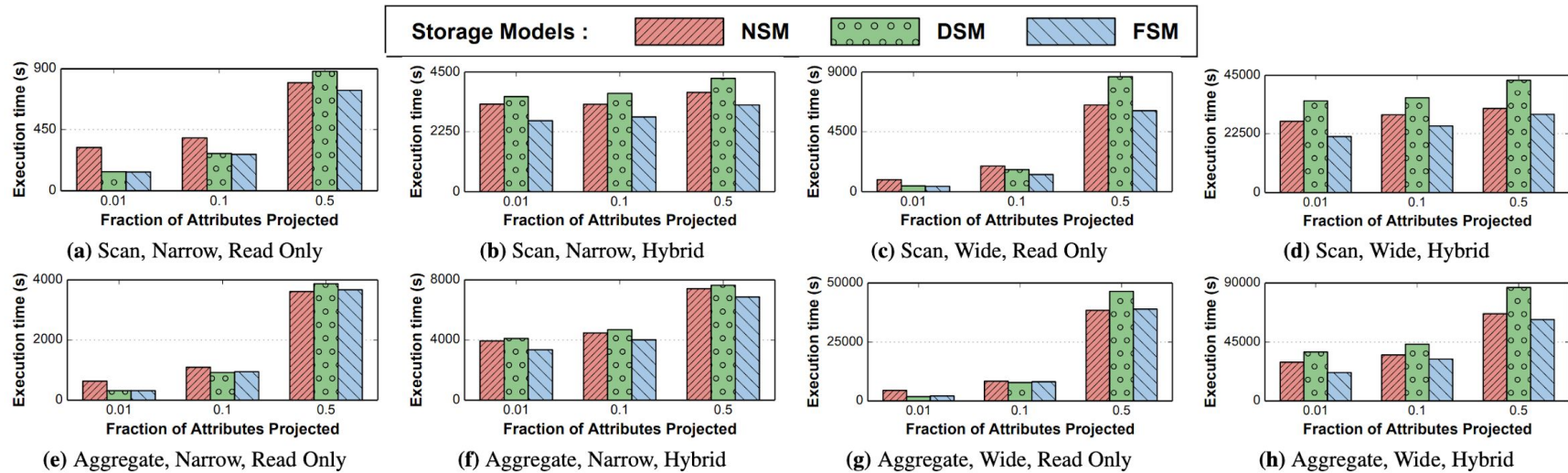


Figure 7: Projectivity Measurements – The impact of the storage layout on the query processing time under different projectivity settings. The execution engine runs the workload with different underlying storage managers on both the narrow and the wide table.

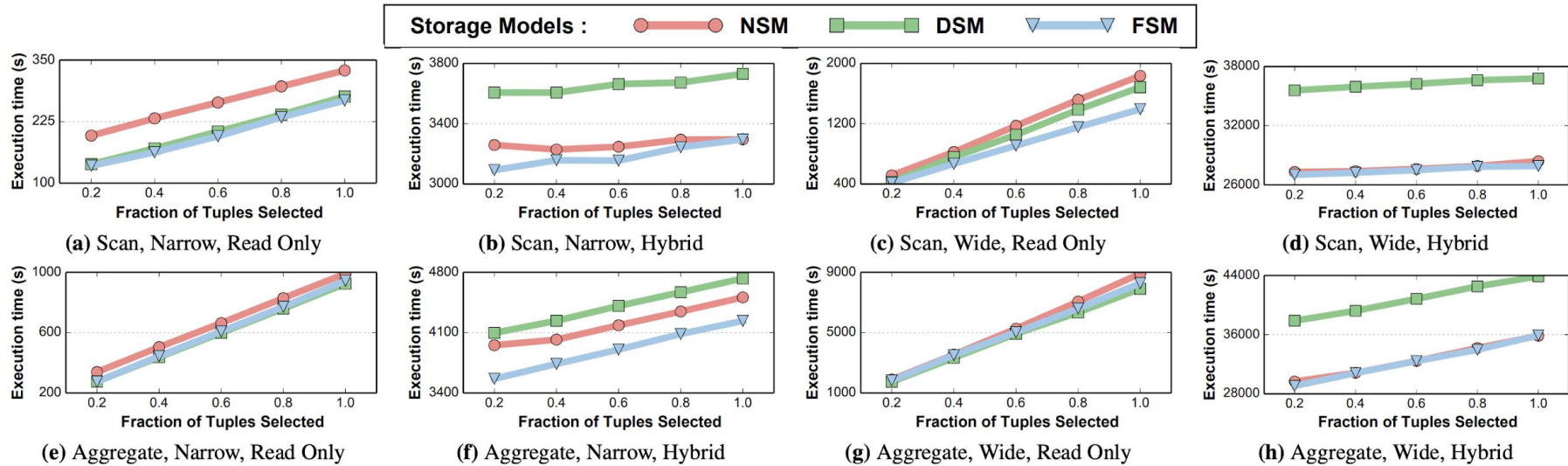


Figure 8: Selectivity Measurements – The impact of the storage layout on the query processing time under different selectivity settings. The execution engine runs the workload with different underlying storage managers on both the narrow and the wide table.

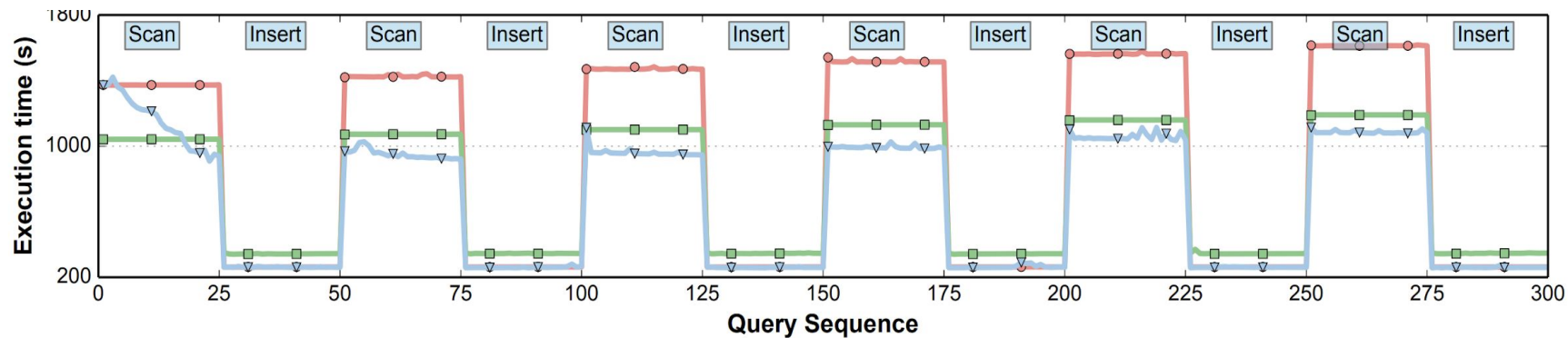


Figure 9: Workload-Aware Adaptation – The impact of tile group layout adaption on the query processing performance in an evolving workload mixture from the ADAPT benchmark. This experiment also examines the behavior of different storage managers while serving different query types in the workload.

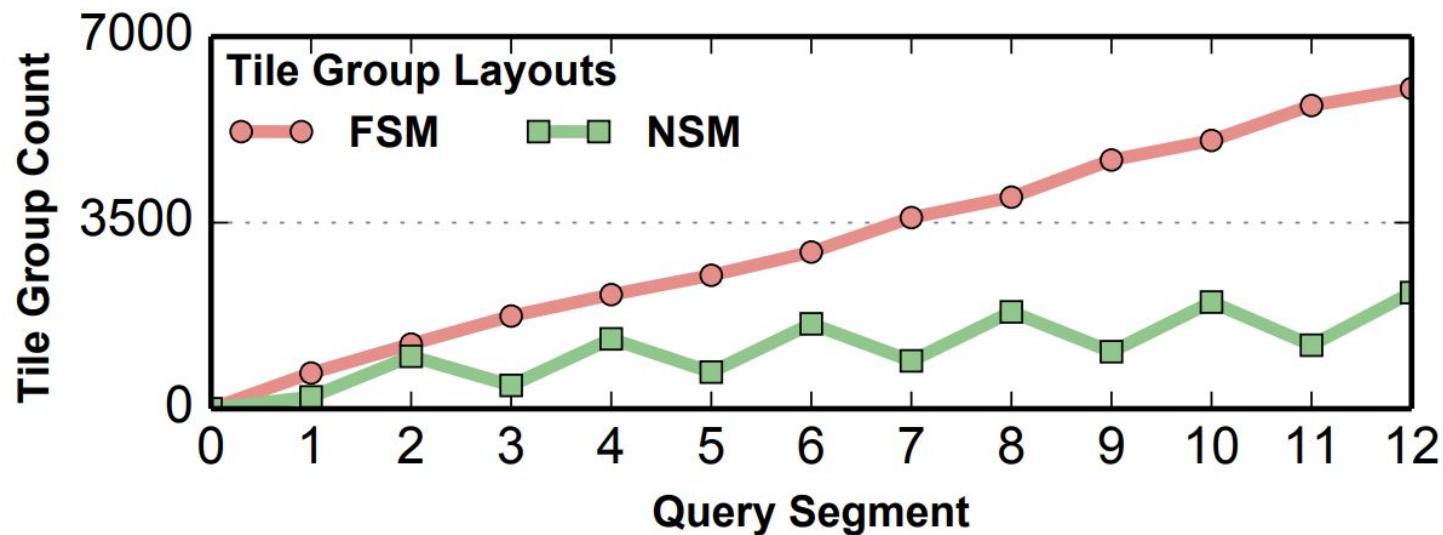


Figure 10: Layout Distribution – The variation in the distribution of tile group layouts of the table over time due to the data reorganization process.

Horizontal Fragmentation

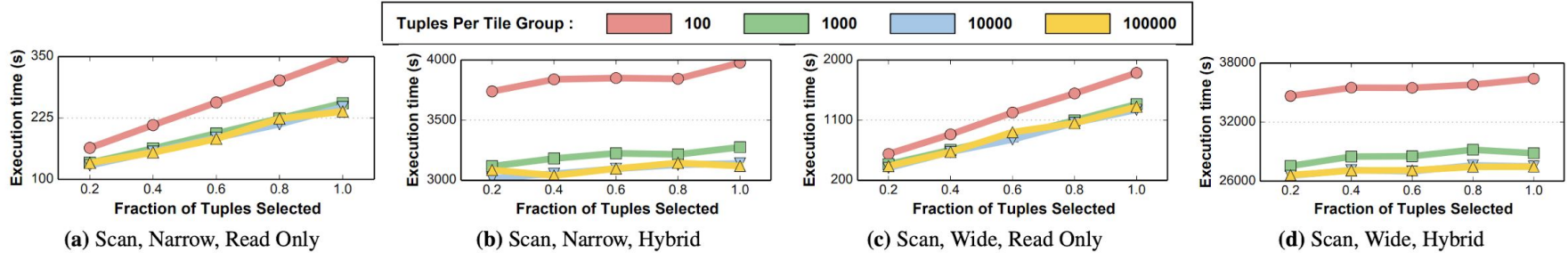


Figure 11: Horizontal Fragmentation – The impact of horizontal fragmentation on the DBMS's performance. We execute the read-only and hybrid workloads comprising of scan queries on the tables in the ADAPT benchmark under different fragmentation settings.

Why the more tuples per tile group, the better the performance?

- This is because of vectorized processing which process data logical tile at a time, reducing the interpretation overhead. The less tuples, the more it is like one tuple at a time execution.

Reorganization Sensitivity Analysis

- Workload contains a sequence of scan queries
- Divided into segments of 1000 queries
- Gradually reduce projectivity of queries from 100% to 10%
- As workload gets executed, updates table to the form of $\{\{a_0\}, \{a_1, \dots, a_k\}, \{a_{k+1}, \dots, a_{500}\}\}$
- k is split point
- Expect k to decrease from 500 to 50

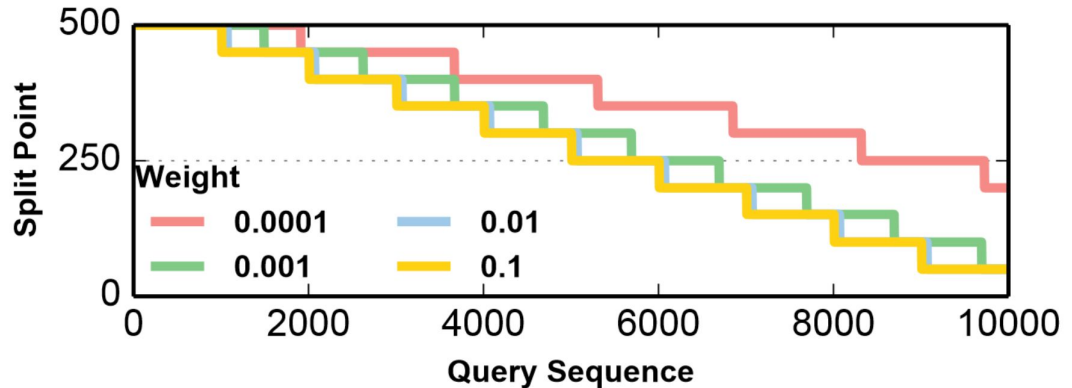


Figure 12: Weight Sensitivity Analysis – The impact of the weight w parameter on the rate at which the clustering algorithm of the data reorganization process adapts with the shifts in the HTAP workload.

Data Reorganization Strategies

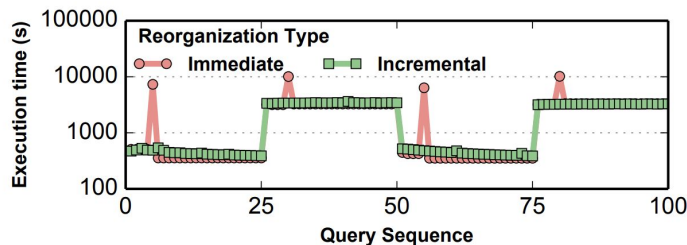


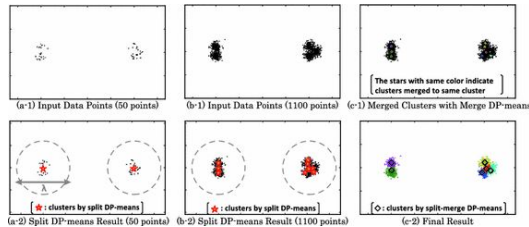
Figure 13: Data Reorganization Strategies – Comparison of the incremental and the immediate data reorganization strategies on a HTAP workload.

What creates these spikes?

- Basically with immediate reorganization 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

Possible Extensions

- One problem with the k-means clustering algorithm used for vertical partitioning is that a **k must be selected**
 - Currently a popular problem within the industry
 - Selecting an incorrect k can have degrading effects
- Can use DP Means Clustering (Lambda Means Clustering) algorithm instead to naturally form clusters within the data
- Lambda Means is **more robust than using a farthest-first heuristic, which requires a user defined k**
- Instead of giving it a parameter k, we give it a parameter λ
- **λ serves as a threshold to define a new cluster**
- The larger the value of λ , the smaller the number of clusters is attained, and vice versa



Algorithm 1. Batch DP-means

Input: Data $\mathcal{X} = \{x_1, \dots, x_n\}$, threshold λ

Output: Centroids $\mathcal{C} = \{\mu_1, \dots, \mu_k\}$

Init: $\mathcal{C} \leftarrow \text{mean}(x_i, x_i \in \mathcal{X}), k = 1$

Init cluster indicators $z_i = 1$ for all $i = 1, \dots, n$

while not converged do

for $x_i \in \mathcal{X}$ **do**

$c \leftarrow \arg \min_c \|x_i - \mu_c\|^2$

if $\|x_i - \mu_c\|^2 > \lambda^2$ **then**

$\mathcal{C} \leftarrow \mathcal{C} \cup x_i$

set $k = k + 1, z_i \leftarrow k$

else

set $z_i \leftarrow c$

for $\mu_c \in \mathcal{C}$ **do**

$\mu_c \leftarrow \text{mean}(x_i, z_i = c)$

Algorithm 2. Online DP-means

Input: New data x , threshold λ

Input: Centroids $\mathcal{C} = \{\mu_1, \dots, \mu_k\}$

Input: The number of assigned data to each cluster $w = \{w_1, \dots, w_k\}$

Output: Updated \mathcal{C} and w

$c \leftarrow \arg \min_c \|x - \mu_c\|^2$

if $\|x - \mu_c\|^2 > \lambda^2$ **then**

$\mathcal{C} \leftarrow \mathcal{C} \cup x$

$w \leftarrow w \cup \{1\}$

else

$\mu_c \leftarrow \frac{w_c \mu_c + x}{w_c + 1}$

$w_c \leftarrow w_c + 1$

Key Points to Remember

- FSM implements tile-based storage
 - Abstraction layer of logical tiles that point to physical tiles
 - FSM is easily parallelizable via metadata
 - Data layout reorganization takes place via k-means clustering
 - FSM is a much better implementation due to data reorganization
-