Proteus: Autonomous Adaptive Storage for Mixed Workloads

Michael Abebe, Horatiu Lazu, Khuzaima Daudjee
Cheriton School of Computer Science, University of Waterloo
{mtabebe,hslazu,kdaudjee}@uwaterloo.ca

ABSTRACT

Enterprises use distributed database systems to meet the demands of mixed or hybrid transaction/analytical processing (HTAP) workloads that contain both transactional (OLTP) and analytical (OLAP) requests. Distributed HTAP systems typically maintain a complete copy of data in row-oriented storage format that is well-suited for OLTP workloads and a second complete copy in column-oriented storage format optimized for OLAP workloads. Maintaining these data copies consumes significant storage space and system resources. Conversely, if a system stores data in a single format, OLTP or OLAP workload performance suffers. This paper presents **Proteus**, a distributed HTAP database system that adaptively and autonomously selects and changes its storage layout to optimize for mixed workloads. Proteus generates physical execution plans that utilize storage-aware operators for efficient transaction execution. Using comprehensive HTAP workloads and state-of-the-art comparison systems, we demonstrate that Proteus delivers superior HTAP performance while providing OLTP *and* OLAP performance on par with designs specialized for either type of workload.

CCS Concepts

• **Information systems** → *Parallel and distributed DBMSs*; *Autonomous database administration*; *Hybrid storage layouts.*

Keywords

Adaptive storage, Hybrid databases, Mixed workloads

ACM Reference Format:

Michael Abebe, Horatiu Lazu, Khuzaima Daudjee. 2022. Proteus: Autonomous Adaptive Storage for Mixed Workloads. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3514221.3517834

1 INTRODUCTION

Enterprises utilize vast amounts of data to support their organizational needs [12, 49, 50, 54]. There has been a rise in the popularity of large-scale real-time transactions and analytics applications over these data to make effective data-driven decisions. For example, an e-commerce organization must both process new online orders and continuously analyze these orders for trends, such as the effects of a promotional sale. Traditionally, enterprises employ separate, specialized, data management systems to handle this workload: *online transaction processing (OLTP)* systems support high throughput transaction processing while *online analytical processing (OLAP)* systems support complex analytics with low latency [12, 46].

order_id	item_id	quantity	amount	delivery
100	50	6	15	2019/01
100	2	90	3	2019/01
101	17	1	100	2020/03
102	8	13	7	2021/06
102	2	30	45	2021/06

(a) Row Oriented OLTP Storage (N-ary Storage Model)

order_id	item_id	quantity	amount	delivery
100	50	6	15	2019/01
100	2	90	3	2019/01
101	17	1	100	2020/03
102	8	13	7	2021/06
102	2	30	45	2021/06

(b) Column Oriented OLAP Storage (Decomposition Storage Model)

Figure 1: Different storage formats for OLTP & OLAP workloads. Colours indicate contiguously stored data.

To process workloads, OLTP systems store data tuples contiguously as *rows*, using the n-ary storage model [13, 32, 57]. Figure 1a shows a simplified example of data taken from the TPC-C benchmark's *orderline* table [1] stored in a row-oriented format: each row is stored contiguously as indicated by its color shade, followed by the next row. Row-oriented storage is optimized for OLTP workloads that operate on a single record at a time and access many attributes. For example, the insert operation from the TPC-C *NewOrder* transaction shown in Figure 2a updates every attribute but affects only a single row.

Row formats, however, result in poor support for analytical workloads as they access many tuples at a time but only a subset of tuple attributes, leading to entire rows of data being processed instead of only the relevant attributes. For example, in the TPC-H benchmark's [3] *Query 6* (Figure 2b), just three of the columns are accessed by the query. Thus, OLAP systems store tuples attribute-at-a-time in *columns* using the decomposition storage model [6, 14, 18]. Figure 1b shows an example of data stored in a column-oriented format: a single column is stored contiguously as indicated by the shaded color, followed by the next column. Although a column

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517834

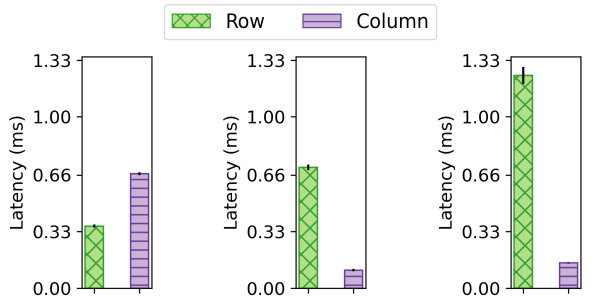
```
insert into orderline
    (200 /* order_id */,  12 /* item_id */, 30 /* quantity */,
     55 /* amount */ , '2021/09' /* delivery */)
```

(a) An OLTP Operation

```
select sum(amount)
from orderline where
    delivery between '2019/01'  and  '2020/12'
    and quantity between 1 and 100000
```

(b) An OLAP Query

Figure 2: OLTP and OLAP operations from TPC-C (*NewOrder* transaction) and TPC-H (Query 6) benchmarks.



(a) Update **(b) Scan (select=10%)** **(c) Scan (select=100%)**

Figure 3: The average latency of 100 updates (3a) and scans of 10,000 data items (3b and 3c) on row and column formats.

format works well for analytics, it is inefficient for OLTP workloads that update multiple columns within a transaction as each stored column is affected. Thus, neither OLTP nor OLAP systems can efficiently support mixed or *hybrid* workloads that span transaction and analytical processing.

We conducted microbenchmark experiments to show that neither row-oriented nor column-oriented storage format is optimal for processing both transactional and analytical workloads concurrently. The microbenchmarks update 100 rows, or perform a scan of 10,000 rows over 1 column out of 10 with 10% selectivity or 100% selectivity. As Figure 3 shows, a row-oriented format supports updates at half the latency of the column-oriented storage. However, column-oriented storage can support analytical (scan) operations 7× faster than row-oriented storage. This experiment demonstrates the performance impact, and importance, of storage format on a hybrid workload. Neither a row format nor a column format alone is suited for a workload consisting of both transactional updates and analytical queries, as the performance of one type of the workload suffers due to the static format of the data.

To mitigate the effect of storage formats on latencies, the traditional architecture for *hybrid* transaction/analytical processing *(HTAP)* workloads periodically migrates new data from an OLTP system to an OLAP system using extract-transform-load (ETL) utilities [46]. While this procedural transformation allows organizations to continue executing OLTP and OLAP workloads concurrently, periodic data migration results in recent (OLTP) updates that are absent in the OLAP system. Thus, organizations cannot obtain real-time insights from their data [12]. Modern HTAP systems address these concerns by storing data in *both* OLTP and OLAP formats and executing queries across both formats in a single integrated system [11, 12, 24, 27, 43]. However, replicating all data within a system consumes at least double the storage resources, particularly for expensive in-memory processing, and requires costly maintenance across replicas to guarantee data freshness and consistency.

1.1 Desiderata for HTAP Systems

Although modern HTAP systems improve performance compared to ETL pipelines, there are four essential aspects of system design that no system has considered integrally. First, a scale-out, distributed HTAP system can meet the data storage and processing demands of large scale workloads that exceed the capabilities of a single node [11, 23, 27, 43, 64]. Additionally, the overheads of distributed transaction coordination mean that applying standard partitioning techniques to existing single-node systems limits scalability. Moreover, scale-out systems allow for parallel execution both within and among queries.

Second, HTAP systems should selectively store the same data item in different formats (e.g., row and column) and in different storage tiers (e.g., memory and disk), simultaneously if desirable, using an efficient replication scheme [11, 43, 51]. HTAP systems that mandate data storage in a single format or storage tier at a time suffer when transaction processing and analytics occur concurrently. Such concurrent workloads frequently occur as a consequence of real-time data analysis (as in our e-commerce example), fraud detection, IoT, and logistics domains [12, 41, 46, 49, 50].

Third, HTAP systems should leverage layout and storage-specific optimizations, such as per-column sort orders and compression [5–7, 29, 30, 33, 66], which reduce CPU and storage overheads, or row splitting [28] to avoid conflicts. Using these optimizations enables HTAP systems to match the performance of specialized systems.

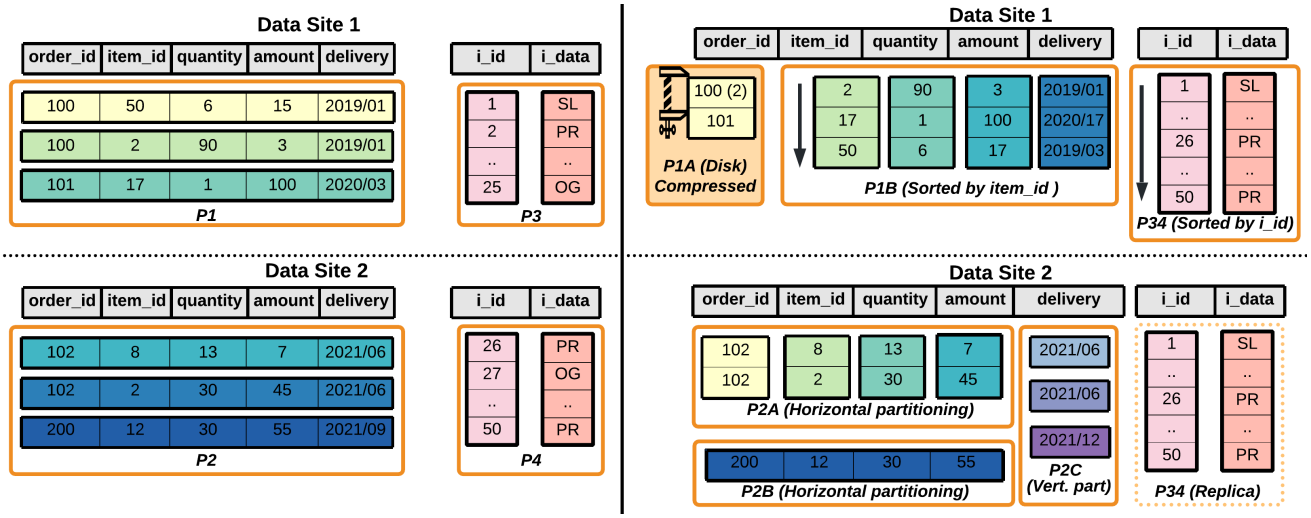
Fourth, based on the workload, HTAP systems should adapt their physical storage layout autonomously, that is, without manual intervention [12, 48]. The rising complexity of workloads means that system administrators cannot decide on a single efficient data layout that both effectively utilizes resources and reduces transaction latency [9, 12]. Furthermore, when workloads change, a layout that works well for one workload is unlikely to work well for another type of workload, as depicted in Figure 3.

1.2 Contributions

We present **Proteus**, a distributed database system that delivers the aforementioned requirements of an *integral* HTAP system. Proteus *adaptively* stores data in *multiple formats* and *storage tiers* to support HTAP workloads efficiently. *Based on the workload*, Proteus makes adaptive storage decisions *autonomously* and leverages *layout-specific optimizations* that enable it to *concurrently* achieve OLTP throughput comparable to row-oriented storage systems and OLAP query latencies that are on par with column-oriented storage systems. Consequently, Proteus significantly outperforms other state-of-the-art distributed HTAP systems.

This paper's contributions are four-fold:

(1) The case for adaptive storage for HTAP workloads through our Proteus system (Sections 2 and 3).
(2) The architecture and design of adaptive storage in Proteus and how transaction and query execution is supported in this environment (Section 4).



(a) Initial State **(b) Adapted State**

Figure 4: An initial storage layout of data in Proteus, and adapted state after a series of storage layout changes.

(3) A model to learn workload patterns and make cost-based layout decisions to support high throughput transactions and low latency queries for HTAP workloads. (Section 5).
(4) An extensive experimental evaluation that demonstrates the effectiveness and performance benefits of Proteus (Section 6).

2 THE CASE FOR ADAPTIVE STORAGE

As Figure 3 demonstrates, neither row nor column format is optimal for an HTAP workload that contains both OLTP transactions and OLAP queries. Furthermore, a system architecture that statically replicates all data in both formats suffers from maintaining replica state efficiently and consistently [12, 46]. In contrast to these architectures, Proteus makes granular storage layout decisions, replicating when it benefits system performance. We elucidate the case for adaptive storage by introducing the set of storage decisions Proteus makes and illustrating its benefits by example.

2.1 Storage Decisions

Proteus distributes and stores relational data among nodes or *data sites* in the system. Proteus supports multiple – row or column – storage formats across multiple – memory or disk – storage tiers in the distributed system. Thus, for a given data item, Proteus adopts a *storage layout* given by its *storage format* and *tier*. Additionally, Proteus supports storage layout optimizations, such as maintaining the data in sorted order or in a compressed form [5–7, 29, 30, 66]. Because Proteus is a distributed database system, for each data item it selects a master (or primary) site where update transactions execute and a storage layout for that site. As Proteus selectively replicates data, it decides at which sites to store replicas of a data item along with an associated layout for each replica.

Deciding on a storage layout for every data item results in a large set of possible decisions to make and manage. Thus, Proteus groups data items together into data *partitions* and decides on a storage layout for each partition copy (replica). A partition is a contiguous range of one or more rows and columns based on the primary key of each row (*row_id*) and column identifiers. For example, in Figure 4a, partition $P1$ spans three contiguous rows. A partition may range

```
select sum(when  i_data like 'PR' then amount
                                  else 0) / sum(amount)
from orderline, item where item_id = i_id
   and delivery between '2019/01'  and  '2020/12'
```

(a) TPC-H Query 14

```
update orderline set delivery = '2021/12'
where order_id = 200
```

(b) TPC-C Delivery Operation

Figure 5: More OLAP and OLTP benchmark operations.

from a single cell to an entire table.

Proteus autonomously manages data by changing its storage layout on-the-fly as the workload executes. These changes include any or all of altering the storage format, storage tier, master and replica locations, and data partition membership. Such changes enable Proteus to adapt to the workload, as we illustrate next using a running example.

2.2 Motivating Example

Figure 4a shows a storage configuration with two data sites that store data from the *orderline* table (Figure 1) along with the *item* table. In this initial configuration, both the *orderline* and *item* tables are partitioned across the two data sites: data partitions $P1$ and $P3$ at the first data site and partitions $P2$ and $P4$ at the second data site. The data partitions for the *orderline* table ($P1$, and $P2$) are in row format, while the partitions storing the *item* table ($P3$ and $P4$) are in column format.

Recall from Section 1 that executing *Query 6* (from Figure 2b) over row-oriented storage requires accessing every attribute from the relevant rows. Furthermore, the results from each site in the distributed system have to be combined to generate the final result. Figure 5 shows two additional transactions: (i) *Query 14* (Figure 5a) from TPC-H [3] that joins the *orderline* and *item* tables to find promotional item order amounts, and (ii) an update to the *delivery* time of a recent order (Figure 5b) from the TPC-C workload [1]. Observe

that executing *Query 14* over this storage layout requires performing a *distributed* join of data, i.e., data in partition *P*1 (*order_id* 100) must be joined with data in partition *P*4 (*item_id* 50). Executing a distributed join requires transferring over the network all of the data necessary to perform the join, which is more expensive than transferring reduced partial (join) results to be merged [22, 38].

In contrast to Figure 4a, Figure 4b shows the *adapted* data storage layout after a sequence of layout changes made by Proteus. Relevant to *Query 14*, this adapted storage layout replicates the *item* table as *P*34 at both sites. Consequently, *Query 14* can perform its join locally, reducing the distributed execution to only the merge of partial results. Selective replication of the *item* table is an efficient choice as the table is read-only and hence does not require update maintenance [9, 63]. A further change at data site 1 is that the layout of the *orderline* table (*P*1*A* and *P*1*B*) is columnar, which accelerates the execution of both queries *6* and *14* by reducing the amount of data accessed when executing the queries. Finally, as the chosen storage layout of both partitions *P*1*B* and *P*34 are sorted by the join key (*item_id*), the system efficiently joins the data using merge-join.

In Figure 4b, Proteus also changes the storage layout of the *orderline* table at data site 2, fragmenting the data (originally *P*2) into three partitions (*P*2*A*, *P*2*B*, and *P*2*C*) via horizontal and vertical partitioning. Hence, there is not one partitioning scheme for the entire orderline table as tables are adaptively partitioned. Data in the *delivery* column is stored in a row format, as is the data associated with *order_id* 200 that was inserted as a result of the *NewOrder* operation (Figure 2a). Note that historical *orderline* data is stored in a column format (*P*2*A*). This storage layout is effective for analytical queries such as *6* and *14*, as these data are less likely to be updated. In contrast, recently (and likely to be) updated data are stored in the OLTP-oriented row format. Vertically partitioning the *delivery* column, a storage optimization also known as *row splitting* [28], reduces data contention from updates to only *delivery* time for recent orders, such as in the *Delivery* transaction (Figure 5b).

In summary, Proteus' adaptive storage layout reduces both OLAP and OLTP execution latency. Using cost-benefit analysis (Section 5), this latency reduction is achieved by (i) distributing and pushing processing to local data sites, (ii) using a storage format aligned with data access patterns, (iii) promoting and prioritizing data locality for in-memory processing over disk-based residency, and (iv) employing optimizations such as sorting and compression, which allows for the selection of query operators to attain efficient execution.

3 PROTEUS OVERVIEW

The last section showed how changes to a storage layout improve transaction execution performance. To support both efficient transaction execution and adaptive storage layout changes, Proteus uses a two-tier architecture consisting of an *adaptive storage advisor* (ASA) and *data sites* (Figure 6), implemented in C++.

Clients submit transactions, one thread per transaction, to the ASA that decides *where* (at which sites) and *how* (execution plan) to execute a transaction. The ASA services a transaction by identifying the locations and layouts of the relevant partitions that the transaction needs to access from stored partition metadata (Section 5.1). The ASA then estimates data access latencies using learned workload models (Section 5.2) to generate an efficient physical execution plan for the transaction and makes cost-based storage layout change

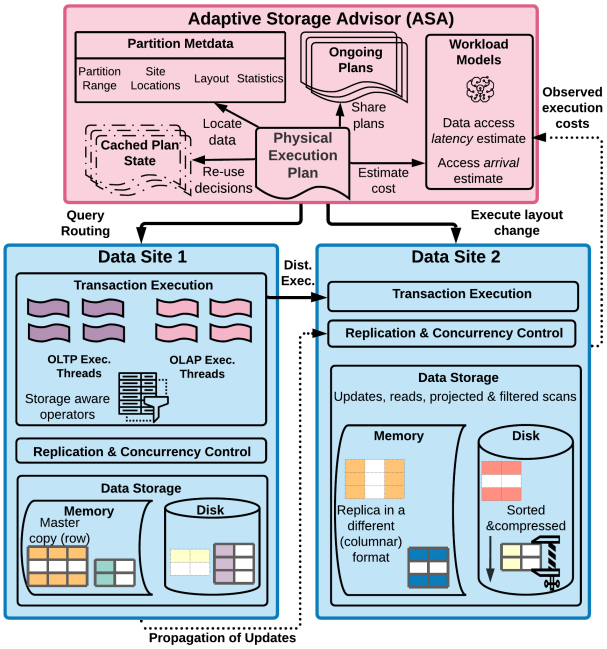


Figure 6: Proteus System Architecture. The adaptive storage advisor (ASA) generates physical execution plans, routing them to data sites for execution and storage layout changes.

decisions based on ongoing and predicted workload characteristics (Section 5.3). To reduce latency, the ASA strives to re-use execution plan decisions from past execution plans (Section 5.3.3). Finally, given the generated plan, the ASA executes the transaction at the selected data sites. Periodically, polling threads asynchronously collect observations from data sites to update the workload models and partition metadata managed by the ASA (Section 5.2.1).

Proteus' data sites store data in the storage layout prescribed by the ASA and execute transactions based on the generated execution plans. Proteus uses OLTP and OLAP-specific thread-pools to execute requests belonging to the respective workloads, isolating compute resources between workloads. Proteus' transaction execution layer performs logical database operations over the stored data, such as joins, using storage aware operators (Section 4.3). As a distributed system, Proteus' transaction execution layer coordinates distributed execution with other data sites using remote procedure calls via Apache Thrift [55]. Proteus ensures transactions observe a consistent state using a partition-based concurrency control scheme and maintains replicas efficiently using a selective per partition replication scheme (Section 4.2). Finally, Proteus stores data partitions in the appropriate storage layout (Section 4.1) to support data updates, point reads, and scans of data ranges including pushing down data projections and filters.

4 PROTEUS SYSTEM ARCHITECTURE

In this section, we describe the design of Proteus with a focus on storage layouts and how Proteus adaptively manages them to execute operations over data efficiently.

4.1 Data Storage

Proteus stores partition data in a row or column-oriented format, and on disk or in-memory. All layouts support reads, writes and

updates, as well as conversion between formats or tiers.

4.1.1 Row-Oriented Storage In memory, Proteus stores each row of a partition using a fixed-size byte array, which is optimized for OLTP transactional access to many cells within a row. To determine the size of the byte array, Proteus uses the table schema and the columns contained in the relevant partition. For example in Figure 4a, Proteus stores each row in *P*1 in a 32 byte array: 4 bytes for each of the integer columns (*order_id* and *item_id*), and 8 bytes for each of the decimal and timestamp columns (*quantity*, *amount*, *delivery*). Proteus stores variable-sized data, such as strings, using 12 bytes with 4 bytes to encode the data size and 8 bytes to store a pointer to the data or the data itself if it fits within the 8 bytes to avoid additional memory accesses. Multi-versioning is used to support efficient updates to data stored in rows. Proteus uses the last 8 bytes in the byte array to store a pointer to a byte array storing the previous version of the row. Thus, once a row is written as a byte array, the data is read-only; updates rewrite the entire row. Proteus stores the partition's data by maintaining an array of pointers to each row's most recently stored version, updating each entry when updates occur.

To store row-oriented data on disk, Proteus divides data into two parts: an index and stored data. Each row's index entry contains an offset into where the row's data is stored. Data for each row is stored similarly to in-memory data; however, any variable-sized data is inlined directly after its length. Proteus' disk-based representation of row data allows for both point-based reads – by reading the index and corresponding offset locations – and data scans.

Proteus supports in-place updates if the update does not change the relative data size; otherwise, Proteus rewrites the entire partition's data on disk. Consequently, Proteus buffers updates in memory and applies them as a batch to disk.

4.1.2 Column-Oriented Storage To store a partition in a column-oriented format in memory, Proteus stores each column in a fixed-sized data array [37] along with two index arrays. For columns with fixed-size data, such as integers, each entry in the array corresponds to the data stored, as shown for the *order_id* column in Figure 1b. For variable-sized data, such as strings, each array entry is stored as a length (using a fixed length of 4 bytes), followed by the bytes containing the actual data stored in its entirety. The first index array is an offset array that stores the corresponding *row_id* for each entry in the data array. The second index array is a position array that stores the offset into the data array that corresponds to the data stored for each *row_id*. These index arrays allow Proteus to efficiently locate where cell data is stored in the column for point reads. Proteus stores column data on disk using a format similar to Parquet [2], first storing the metadata including the two index arrays followed by the values.

Proteus supports flexible ordering of data: each column in the partition may store data in the same order — by *row_id* or based on a total order over the columns such as in partition *P*1*B* (Figure 4b), or each column may be sorted independently. Additionally, Proteus supports compressing data stored in columnar format using run-length encoding [5] by prefixing each entry with 4 bytes to indicate the length of the run, as shown in partition *P*1*A* (Figure 4b).

Proteus buffers updates to column data in a *delta store*, which stores data in memory as rows in a hash-table indexed by *row_id* [36]. Thus, if scans or point-reads require accessing more recent data than stored in the columns, Proteus combines the stored column data with data from the delta store. Updates in the delta store are periodically merged with the column data to create a new version of the data for storage on disk or in memory.

4.1.3 Zone Maps Independent of the storage layout, Proteus maintains zone maps [22] that maintain the minimum and maximum value for each column stored within a partition. Zone maps allow skipping data in a partition if the minimum and maximum values indicate that a predicate in a query cannot be satisfied by any data item in the partition. For example, in Figure 4a, the zone map of partition *P2C* would indicate that there are no *orderline* entries that satisfy the *delivery* predicates in *Query 6* or *14*. Proteus maintains zone maps in memory and in row format, as they are of fixed size and accessed by point queries.

4.2 Concurrency Control and Replication

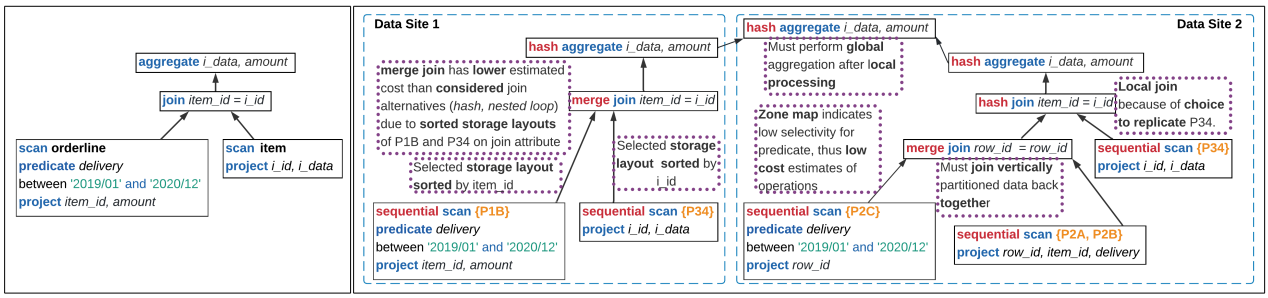
Proteus, like several other HTAP systems [27, 52, 58, 64], supports *snapshot isolation* (SI), which guarantees all transactions see a consistent snapshot of the database. Proteus strengthens this guarantee by providing *strong session snapshot isolation* (SSSI) [20] that prevents transaction inversion and ensures every client sees the effects of all updates from its previous transactions, thereby providing a data freshness guarantee. To provide SSSI, Proteus uses a partition-based dependency-tracking concurrency control algorithm [9] that tracks: (i) per partition version numbers, and (ii) the dependencies among partitions and their versions to ensure that transactions read from a consistent snapshot of data.

Both Proteus' concurrency control and ASA logic make use of transactional read/write information. For analytical queries, clients determine read cell ranges from the columns accessed in each table necessary to execute each query, e.g., *Query 14* (Figure 5a) accesses the *i_data*, and *i_id* columns in the *item* table and the *item_id*, *amount*, and *delivery* columns in the *orderline* table. For OLTP transactions, clients have this information based on primary keys, or if needed execute reconnaissance queries [40, 62].

Proteus selectively and adaptively replicates data lazily using a per partition replication scheme [9]. Data sites replicating a partition subscribe to a per partition redo-log (stored in Apache Kafka [35]) into which the master writes updates on commit. Replication threads manage polling updates from the redo-logs and emplace the updates into per partition queues. Replication threads co-operate with a transaction execution thread to apply propagated and queued updates as necessary to ensure the transaction satisfies SSSI.

4.3 Transaction Execution

Despite the variety of storage layouts supported in Proteus, the system capably presents a uniform transaction execution interface. The ASA uses a query tree (Figure 7a) to develop a physical execution plan for each submitted transaction that specifies where, and how, each operation should execute (Figure 7b). At a data site, the transaction execution layer executes the physical execution plan including accessing stored data, coordinating distributed execution with other data sites, and computing the transaction's result. To do so, Proteus applies operators such as *scan*, *join*, *update*, *insert* and *delete* that iterate over partition(s) (Table 1). Proteus chains these



(a) Query Tree **(b) Physical Execution Plan**

Figure 7: Proteus' query tree and physical execution plan for TPC-H Query 14 (Figure 5a) under storage layout from Figure 4b.

operators together to execute each transactional request.

Proteus' operators come in two forms: storage-aware or storage-agnostic. Storage-aware operators leverage knowledge of the storage layout to optimize execution. For example, in Figure 4b, partitions $P1B$ and $P34$ are sorted on the join attributes (*item_id* and *i_id*, respectively), hence a sorted column-storage aware *join* operator that implements the *merge* join algorithm leverages this knowledge to efficiently execute the join. Generally, our storage-aware operators are designed to (i) use column-specific operators, such as the invisible join algorithm for hash-joins [6], (ii) operate directly over compressed or sorted data, and (iii) use block-based accesses for data residing on disk. Storage-agnostic implementations use the same generic storage interface regardless of how the system stores the relevant partition data. For example, acquiring a lock on a partition is agnostic to the layout. Storage agnostic data accesses and updates use cell-based operations.

As a distributed system, Proteus coordinates transactions that access data spanning multiple sites. To execute distributed analytical queries, Proteus executes joins across sites, coordinating both data transfers and aggregation of results. For example, in Figure 7b, each data site executes a local join, but data site 2 coordinates the transaction by aggregating the results globally. To do so, Proteus leverages the replicated *item* table partition in the join at both data sites (Figure 7b). To ensure the correctness of the join result, at least one side of a join executes over precisely one copy of each partition, which in our example corresponds to partitions $\{P1B, P2A, P2B, P2C\}$ all storing the *orderline* data. Pipelined join execution over different partitions across all sites eliminates the need for duplicate generation of results. Proteus coordinates distributed updates using two-phase commit, if necessary. Since master partition placement is a storage layout decision, the ASA can adaptively change the master placement if its cost-based model determines the change to be beneficial (Section 5.3.2).

Proteus provides fault tolerance using the redo-logs that data sites write to on (i) transaction commit, (ii) partitioning changes, and (iii) mastership changes. Data sites recover by installing snapshotted data partitions from stored checkpoints or replicas and replaying updates from the redo-log, ensuring that distributed transactions recover atomically. The ASA recovers its state by consulting each data site's stored partitions and their layouts.

4.4 Changing Storage Layouts

Proteus supports storage layout adaptivity by changing any or all of: (i) the storage format or tier of a data partition (ii) adding or removing storage optimizations (iii) changing the data partitioning (iv) adding or removing replicas of a partition (v) changing the location of the master copy of a partition. Here, we focus on the mechanism for executing these changes. The next section details how the ASA makes decisions for adapting the system storage layout.

To change the storage format or tier, Proteus reads a consistent snapshot of the data into memory and bulk loads the data into the respective storage layout. For example, to bulk load row format data into memory, Proteus allocates a fixed-size buffer for every row and updates each cell as it reads the data. By contrast, bulk loading row format data that will reside on disk requires dynamic allocation of each row based on variable-sized columns, which Proteus writes to disk sequentially. Enabling sorting entails changing data storage using bulk load operations; removing a sort order does not change the data layout but ceases to maintain the sort order on subsequent updates. Compressing and decompressing stored data result in changing how it is stored, as the run-length encoding scheme prepends the length of each run before each data item.

Changes to data partitioning schemes occur dynamically by merging or splitting partitions, either horizontally (row-wise) or vertically (column-wise). For example in Figure 4b, Proteus forms partition $P2C$ by vertically partitioning $P2$ in Figure 4a, while a subsequent horizontal partitioning forms partitions $P2A$ and $P2B$. By contrast, Proteus forms partition $P34$ (Figure 4b) by merging partitions $P3$ and $P4$ (Figure 4a). Changes to the horizontal partitioning of row formatted data simply require changing row mapping from one partition to another. A similar operation occurs for vertical partitioning of column-format data. By contrast, horizontal partitioning of column data is bulk reloaded into the new partitions.

To add a partition replica, Proteus snapshots the master copy of the partition and installs it as the new replica that begins subscribing to the partition's redo-log to apply updates. By contrast, removing a replica stops its subscription to the redo-log and marks the stored data for deletion, which occurs after ongoing operations have finished accessing the data. Proteus supports changes to partition mastership [8] by routing new update transactions to the new master and blocking their execution until the master has applied all updates from the partition's previous master site.

5 ADAPTIVE STORAGE ADVISOR

Proteus' adaptive storage advisor generates a physical execution plan for each client request based on the system's storage layout, and adapts its layout based on the workload to further improve

Cost Function	Arguments
Storage Layout-Aware	
Bulk Load Insert/Update/Delete Point Read	(i) # Cells accessed (ii) Column sizes
Scan w/ predicate & projection *(Sequential, Sorted, Index)*	(i) Cardinality (ii) Column sizes (input & output) (iii) Selectivity
Sort Hash	(i) Cardinality (ii) Column sizes
Join *(Hash, Nested Loop, Merge)*	(i) Cardinality (left, right, output) (ii) Column sizes (left, right, output) (iii) Join selectivity
Aggregate *(Hash, Sort)*	(i) Cardinality (input, output) (ii) Column sizes
Storage Layout-Agnostic	
Network Request	(i) CPU utilization at source & dest. (ii) # Bytes (sent & received)
Lock Acquisition	(i) Contention of partition
Waiting for Updates	(i) # Updates needed
Commit	(i) # Partitions read and written (ii) # Sites involved in transaction

Table 1: Cost functions and their arguments.

system performance. The ASA accomplishes these tasks through (i) tracking the current storage layout using metadata, (ii) modelling the workload to estimate access latencies under current and adapted storage layouts, and (iii) reusing previous decisions to reduce the latency of planning. We expand on each of these techniques next.

5.1 Partition Metadata

The ASA tracks the metadata state of each data partition in a concurrent hash-table structure for efficient lookups. For each partition, the ASA maintains: (i) the partition bounds (minimum and maximum *row_id* and columns) (ii) the storage layout of each replica of the partition (iii) access frequencies over different time scales (minutes and hours) for updates, point reads, and scans (iv) a zone map (v) the set of partitions frequently co-accessed with the partition as a result of updates or joins. Proteus uses access frequencies to predict upcoming accesses to the partition and to estimate access costs under different storage layouts (Section 5.2). The ASA uses the zone maps to estimate the selectivity of predicates and joins. Tracking co-access likelihood enables the ASA to reduce distributed coordination by co-locating co-accessed partitions.

The ASA also maintains per table column statistics, including each column's average size and per column access rates. Proteus uses these statistics to estimate the storage space required to store a given partition and per column access trends.

5.2 Workload Models

Proteus uses learned workload models to estimate the benefit of storage layout changes. The models learn (i) cost functions that predict the latency of operations and (ii) estimates of access arrivals.

5.2.1 Cost Functions To select a physical execution plan or decide on a storage layout change, Proteus quantitatively evaluates the effects of its decisions. As physical operators enable layout changes, an interpretable way to capture the cost of each operator is the time it takes to execute, i.e. its latency. Using latency to compare the effects of different storage layouts on performance also serves to directly minimize transaction latency [9, 26, 53, 60]. Hence, Proteus predicts the latency of operations in the system using learned cost functions based on the statistics described in Section 5.1.

Table 1 summarizes the different cost functions that are used to estimate the latency of executing a query or a storage layout change. We classify cost functions as storage layout-aware, such as updating data, or layout-agnostic, such as the latency of performing a network request. Proteus learns a single cost function for storage layout-agnostic functions, while for storage layout-aware functions, Proteus learns a cost function per storage layout based on the storage tier, format, and enabled optimizations.

Proteus combines the cost functions to predict the overall latency of a physical execution plan. For example, to predict the latency of the plan shown in Figure 7b, Proteus would predict the latency of executing the sequential scans over column partitions $P1B$ and $P3$4, merge-joining the sorted partitions and performing an aggregation at data site 1. A similar computation would be performed for operations at data site 2, including estimating the network request latency to data site 1 before the final aggregation.

Proteus' cost functions continuously learn based on the observed latency of executing operations. The cost functions use (i) linear regression, (ii) non-linear regression, and (iii) neural network models to predict operation latency, which are implemented using the Dlib library [34]. Data sites track operation latency and periodically report these observations to the ASA to update its cost functions.

5.2.2 Estimate Access Arrival Proteus considers the effect of adapting storage layouts on future requests. Both analytical and transactional workloads exhibit temporal and cyclic trends in requests that arise due to follow-the-sun behaviour, or scheduled reporting [42, 53, 59]. Hence, Proteus uses learned models to predict the likelihood of future data accesses and their arrival time.

Predicting when data will be accessed requires: (i) predicting trends at different temporal granularities, such as daily, weekly or yearly trends (ii) accounting for growth and spikes in requests, which may arise due to specific dates such as the fiscal year-end (iii) adapting to changes in workloads over time, which may occur if the queries or transactions submitted to the system change due to user needs or preferences [42, 61]. Proteus tracks accesses per partition by access type (update, point read, or scan). By default, these statistics are tracked over 5-minute intervals for the past day and hourly for a month and supplied as input to Proteus' predictive models. Two different predictive models are used: (i) sparse periodic auto regression (SPAR) [15] and (ii) a hybrid-ensemble predictor that combines a recurrent neural network (RNN) implemented via libtorch [47], and a linear trend predictor with a custom user-configurable holiday list [42, 61]. The holiday list accounts for recurring events without periodicity, e.g., Black Friday is not on the same date every year. In contrast to the SPAR model, the hybrid-ensemble method automatically learns the periodicity of the workload without requiring a user-defined period.

5.3 Query and Storage Layout Planning

The ASA generates physical execution plans based on its workload models and the current storage layout. Next, we describe how Proteus generates physical execution plans, makes storage layout

decisions, and reuses past decisions to accelerate these processes.

5.3.1 Physical Execution Plans Proteus begins the physical execution plan generation process by starting out with a query tree for the request, which is generated using PostgreSQL's parser and analyzer. Using the query tree, Proteus generates a physical execution plan by (i) replacing accessed tables in leaf nodes with the relevant corresponding partitions at a specific site, (ii) instantiating internal nodes with operators, and (iii) adding additional operators to handle distributed operations as necessary.

Note that in Figure 7a, the leaves of the query tree are accesses to the *orderline* and *item* tables. When generating the associated physical execution plan shown in Figure 7b, Proteus replaces these leaves with accesses to the relevant partition data identified by partition metadata. Figure 4b shows that with respect to the *orderline* table, there are four relevant partitions: $P1B$, $P2A$, $P2B$, $P2C$. Thus, Proteus replaces the leaf with an instantiation of access to partition $P1B$. Due to vertical partitioning of the data, accesses to $P2A$, $P2B$, and $P2C$ result in rewriting of the leaf nodes with separate accesses to the partitions. Proteus inserts an internal *join* node to make the required columns available together to the parent operator. Proteus also selects the sites where data access to the partition will occur, enforcing that at least one side of any join operation executes at exactly one site for each relevant partition.

Given an assignment of the site and data access to each leaf node in the query tree, Proteus assigns a physical operator to each node in the query tree. For example, Proteus selects to join partitions $P1B$ and $P34$ using a merge-join algorithm over other alternatives such as a hash or nested-loop join algorithm because its cost functions estimate the merge-join to have the lowest cost. Proteus makes this decision greedily to effectively reduce the physical execution plan search space. Operator(s) to the plan are added if a selected physical operator needs them, such as inserting sort operators before merge-join. Proteus adds nodes to combine and coordinate distributed operations, e.g., Figure 7b's distributed aggregation at data site 2.

5.3.2 Layout Changes Proteus plans layout changes in response to three stimuli: (i) generating a physical execution plan for a request (ii) requests predicted to arrive in an upcoming 10-minute (configurable) interval (iii) when a data site nears storage constraints, e.g., memory or disk capacity limits. In each case, Proteus integrates the workload models (Section 5.2) into one equation, the net benefit of a change defined by $N(S)$ that quantifies how beneficial a change is to the system overall. Proteus computes $N(S)$ by estimating (i) the upfront cost $U(S)$ to execute the storage layout change based on the operations needed to perform the change (via the cost functions in Table 1), and (ii) the expected cost effect $E(S)$ on requests predicted to arrive and on requests currently executing ($C(S)$). Proteus computes $E(S)$ and $C(S)$ by computing the difference between predicted transaction latency under the current and proposed layouts, weighted by the likelihood of the transaction executing. The ASA computes the net benefit[1] $N(S) = \lambda(E(S) + C(S)) - U(S) > 0$.

In our running example (Figure 4b), Proteus chose to vertically partition $P2$ to produce partition $P2C$ to mitigate the contention effects of OLTP operations, such as the delivery transaction (Figure 5b). The ASA estimated the proposed layout will reduce transaction latency, resulting in a positive $E(S)$. To perform this vertical partitioning necessitates an upfront cost ($U(S)$), primarily induced by the cost of scanning the original partition and bulk loading the new partitions. This change affects other queries, such as the join in Query 14, slightly increasing its latency under the proposed change as it induces another join (Figure 4b). However, Proteus estimates the impact is small due to the selectivity of predicates over $P2C$, resulting in a low join cost and a small negative $C(S)$. Thus, $N(S)$ is positive so Proteus proceeds with the change.

After generating a physical execution plan, the ASA performs a top-down search to find the leaf that contributes (through data access) the highest cost to the overall plan. In our running example, the latency induced by contention when accessing partition $P2$ is the highest leaf cost, and thus the ASA considers it a candidate for a storage layout change, in this case vertical partitioning. The ASA considers if any storage layout changes that affect the high-cost leaf induce a positive net benefit ($N(S)$). If so, Proteus initiates the storage layout change and updates its physical execution plan. Proteus repeats this process, stopping when further changes do not result in a positive increase to the value of $N(S)$. In our running example, vertically partitioning $P2$ to produce $P2C$ is beneficial, so it is added to the plan. Proteus horizontally partitions $P2$ to produce $P2A$ and $P2B$ but stops after this change as further changes do not improve $N(S)$.

To make effective changes without sacrificing efficiency, Proteus tracks the average estimated leaf access cost per type of operation, normalized by the number of cells accessed, and performs this planning step (i) if this normalized cost is above average or (ii) probabilistically with probability inversely proportional to the normalized cost. Thus, Proteus reduces cost by changing the storage layouts of partitions with high access costs to generate higher potential latency savings.

Proteus periodically predicts upcoming accesses and considers storage layout changes if the predicted access pattern differs from the recent pattern of accesses. For example, if Proteus predicts frequent updates to a partition because of a cyclical access pattern but the partition has been infrequently updated recently, then Proteus adapts the partition storage layout. To do so, a physical execution plan is generated for a placeholder transaction that accesses the partition. Proteus then predictively plans storage layout changes using the aforementioned procedure. The placeholder transaction is instantiated with accesses based on recorded partition co-access statistics (Section 5.1). Proteus constrains the search space of potential predictive layout changes by considering the workload in only the upcoming 10-minute (configurable) interval. As we show experimentally in Section 6.3.3, Proteus executes layout change plans efficiently, in sub-second latencies on average, and hence can execute many changes predictively within such a time window.

Data sites track their storage usage per tier and report to the ASA as they approach the capacity limit within a tier. In response, Proteus considers executing storage layout changes that reduce the consumed storage capacity by: (i) removing replica partitions (ii) changing partition mastership to another site (iii) compressing partition data (iv) moving partition data to a lower storage tier. Proteus estimates the expected benefit of each of these options for data

[1] $\lambda > 0$ controls the importance of the expected benefit of the storage layout change.

partitions stored in the tier and selects the option that maximizes the expected benefit $N(S)$. To avoid considering all partitions, data partitions are grouped into tiers by their access statistics, including estimated access arrival times and estimated $N(S)$ for each decision for a partition group as a whole. Once Proteus selects a partition group and makes a decision, each partition within the group is repetitively considered under the decision, executing storage layout changes until the site is under its capacity limit(s).

5.3.3 Plan Reuse To reduce planning overhead, Proteus caches previously used physical execution plans for reuse. A plan is reused in its entirety if the current storage layout satisfies the layout used in the cached plan. For example, consider Query 14 executed on the storage layout shown in Figure 4b using the physical execution plan shown in Figure 7b. If Proteus receives this same request again with the same storage layout, then it would reuse that same physical execution plan. As Proteus adaptively changes its storage layout, a single change invalidates a plan. Hence, Proteus carries out plan decision reuse. To do so, the input arguments for each operator are bucketed and the decisions made given these arguments are cached. If a subsequent decision has similar inputs, then Proteus reuses the decision with the lowest estimated cost. Proteus also uses this technique to reuse decisions for storage layout changes.

6 EXPERIMENTAL EVALUATION

In this section, we present our experimental evaluation that demonstrates the effectiveness of Proteus' storage layout adaption and how it significantly boosts HTAP system performance for varying HTAP mixes, access patterns and load.

6.1 Methodology and Benchmarks

Our experiments are conducted on up to 18 data sites (default 6 sites), with each site having 12-cores, 32 GB of RAM and a 1 TB hard disk. Proteus uses 1 machine for the ASA and 2 machines to run Apache Kafka. A 10 Gbps network connects all machines. Results are averages of at least five 20 minute OLTPBench [21] runs with 95% confidence intervals shown using error bars.

We conduct experiments using three HTAP workloads: the **CH-benCHmark** (CH) [16], transactional **YCSB** [17] and **Twitter** [21] benchmarks. CH consists of the TPC-C OLTP [1] and TPC-H OLAP workloads [3]. The transactional YCSB workload consists of two types of transactions: a 10-key (multi-key) read-modify write OLTP transaction and an OLAP query that scans 500,000 rows, evaluates a predicate, and aggregates the results. The Twitter workload models a social networking application featuring heavily skewed many-to-many relationships among users, their tweets and followers. The workload contains six OLAP transactions and three OLTP transactions with four transactions added from the Twitter API [4] that update followers, get tweets from followers, get tweets within a timespan, and get tweets starting with specific text, introducing more OLAP and OLTP transactions.

For all workloads, we follow CH's model of a client submitting either OLTP or OLAP transactions at any one time. As in prior work [16], we evaluate using three mixes: *OLTP heavy* (90% of clients submit OLTP transactions), *balanced* (50% OLTP), and *OLAP heavy* (10% OLTP). To conduct workload execution time experiments, we fix the number of transactions executed by each client. For YCSB, each client issues 1 OLAP transaction followed by a proportional number of OLTP transactions by mix (10 for OLTP heavy, 6 for balanced, and 3 for OLAP heavy) 10,000 times. For CH, each client issues 220 TPC-H queries and the OLTP to OLAP proportions are 999:1 (OLTP heavy), 99:1 (balanced), and 19:1 (OLAP heavy). For Twitter, each client issues 300 OLAP queries and the OLTP to OLAP proportions are 1000:1 (OLTP heavy), 100:1 (balanced) and 10:1 (OLAP heavy).[2] The YCSB database (50 GB) consists of 50 million rows and 10 columns each storing 100 bytes. The YCSB workload uses skewed OLTP accesses to generate contention and load imbalance. CH uses a scale factor of 100 (100 GB) while Twitter stores 10 million user accounts (80 GB).

6.2 Evaluated Systems

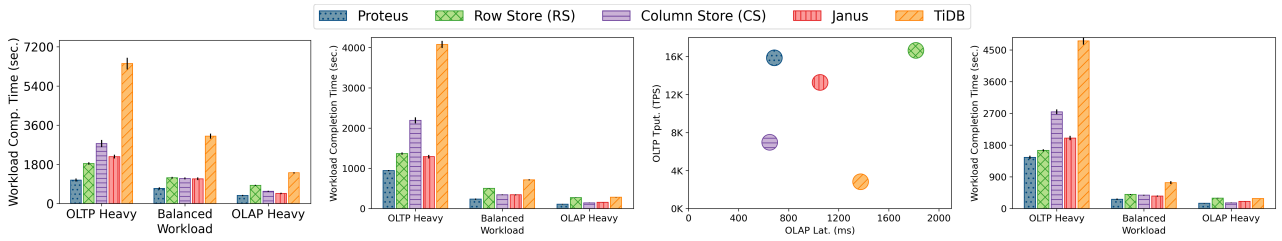
We evaluate Proteus against the alternative distributed HTAP database system architectures of **Janus** [11] and **TiDB** [27], as well as a **row**-oriented distributed database (row store or **RS**) targeted to optimize OLTP workload execution and a **column**-oriented distributed database (column store or **CS**) designed to optimize OLAP workload processing. To ensure an apples-to-apples comparison of techniques, we implement the RS and CS architectures in Proteus, which provides for an in situ comparison of the two specialized storage formats. We implement Janus in Proteus with Proteus' adaptive features disabled. We use the commercial open source version of TiDB. TiDB and Janus fully replicate data between their OLTP-optimized and OLAP-optimized stores. Janus executes OLTP transactions on the RS and OLAP transactions on the CS [11], and updates are propagated lazily from the OLTP to the OLAP store. By contrast, TiDB uses Raft as its replication algorithm and a cost model to determine where to run a given transaction [27].

We advantage each comparison system implemented in Proteus with an optimized replication and partitioning scheme using Schism [19] that uses *a priori* knowledge of the workload, including whether a table is read-only. These systems use the Least-Recently-Used (LRU) scheme to determine storage tier placement as LRU is appropriate for both skewed and uniform access patterns and is a popularly-used storage tier policy [25, 45].

6.3 Results

6.3.1 Workload Execution/Completion Time To establish Proteus' ability to process HTAP workloads efficiently, we evaluate all systems with the 3 YCSB HTAP workload mixes. We measured the (execution) time to complete these workloads, shown in Figure 8a. Proteus executes the balanced workload 4.4× and 1.6× faster than TiDB and Janus, respectively, and completes the workload faster than all competitors for all mixes. Proteus achieves these results because its adaptive storage techniques result in superior OLTP throughput compared to RS and OLAP latency that is competitive with CS (Figure 9). Only Proteus achieves this high level of performance for both OLTP and OLAP workloads. Proteus' superiority on hybrid workloads is also evident for CH (Figure 8b). Proteus reduces the time to execute the balanced workload by more than 33% compared to CS, 32% compared to Janus, more than 50% compared to RS, and 70% compared to TiDB. In all mixes, Proteus executes the workload faster than all of its competitors because its OLAP

[2] Ratios represent the proportion of executed OLTP to OLAP transactions.



(a) YCSB Completion Time **(b) CH Completion Time** **(c) CH Latency vs. Throughput** **(d) Twitter Completion Time**

Figure 8: Proteus executes hybrid workloads faster than all competitors for all YCSB, CH-benCHmark (CH) and Twitter mixes.

performance remains competitive with that of CS while delivering equivalent OLTP performance to that of RS, as shown in Figure 8c.

In the Twitter workload, Proteus has the lowest workload completion time for all mixes, and reduces the time to execute the balanced workload by more than 25% compared to Janus, 30% compared to CS, 45% compared to RS, and 65% compared to TIDB. Similar to the CH workload, Proteus is superior to its competitors in the Twitter workload because its OLAP latency is similar to that of CS while delivering OLTP throughput competitive to that of RS (Figure 11). **Takeaway:** Proteus achieves superior performance for hybrid workloads, completing them faster than all competitors for all mixes.

6.3.2 YCSB We now present Proteus' performance for the dimensions of OLTP throughput and OLAP latency. Figures 9a, 9b and 9c show throughput of the OLTP operations for each system on the three HTAP mixes. Proteus achieves the highest throughput, outperforming competitor systems by between 9.3× and 1.4× in the OLTP heavy workload. In this experiment, Proteus' closest competitors are RS and Janus, which execute the OLTP operations on row-oriented storage. Examination of Proteus' storage layouts shows that the ASA learns a suitable layout: frequently updated data is stored in row format with infrequently updated data stored in column format. Due to the skew in the OLTP workload, most updates execute over data stored in a row format.

Proteus' adaptive storage techniques, including vertical and horizontal partitioning of data (Section 4.4) to mitigate contention effects within and across rows, result in Proteus outperforming both RS and Janus that use static storage layouts. As the workload becomes more OLAP heavy (Figure 9c), Proteus adapts its storage design, trading off OLTP performance in favour of OLAP performance. We present the effectiveness of vertical and horizontal partitioning techniques via an ablation study in Section 6.3.7.

Figures 9e, 9f and 9g show the average latency of the OLAP operations for each system on the three HTAP mixes. Proteus achieves an average OLAP latency on par (within 10ms) with CS and significantly reduces the OLAP latency compared to all other competitors by between 3.1× and 1.3×. As the workload becomes OLAP heavy, Proteus shrinks the latency gap from CS. Proteus achieves comparable latency to CS because Proteus stores most of the data in a column-only format to support uniform data scan accesses across the table. Consequently, Proteus needs to execute OLAP operations only across data stored in a row-only format for the update-heavy parts of the database. Importantly, Proteus achieves these results while delivering more than 4× the OLTP throughput of CS.

Janus and TiDB replicate all data twice, consuming on average 1.33× more space than Proteus and imposing memory constraints on the system. By contrast, Proteus selectively and judiciously replicates partition data into both a row and column format when there are roughly equal OLTP and OLAP accesses, reducing the amount of stored data in the system. Proteus further reduces space consumption by adaptively employing compression on the most infrequently updated data in the system (detailed in Section 6.3.7). **Takeaway:** Proteus is the ideal choice for the hybrid YCSB workload, providing on par OLAP latency and outperforming OLTP competitors.

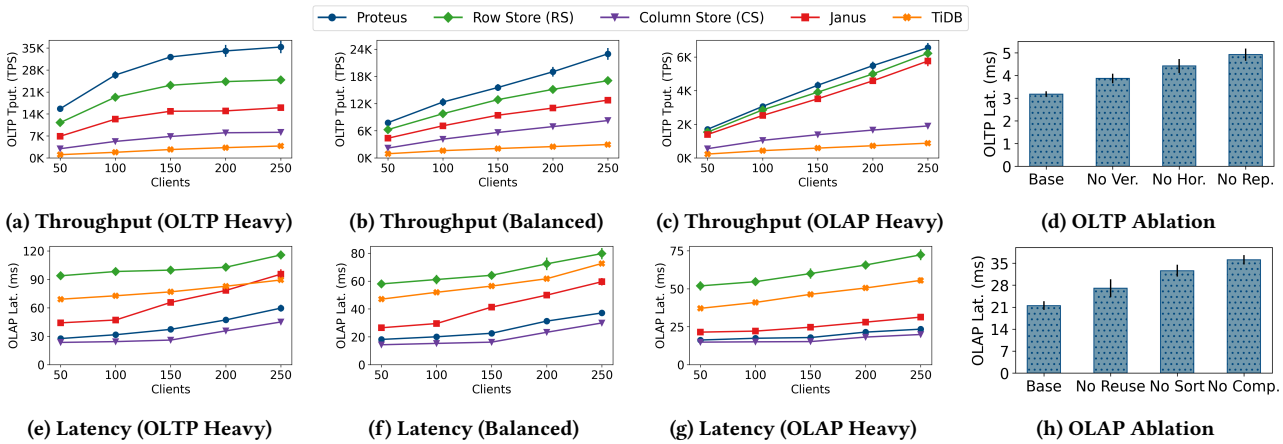
6.3.3 CH-benCHmark We evaluate Proteus using CH, an HTAP workload derived from 22 TPC-H OLAP queries and 5 TPC-C OLTP transactions. For all mixes, Proteus achieves throughput comparable with the top-performing OLTP system – within 5% of RS OLTP throughput (Figure 10a) *and* within 8% of the CS OLAP latency (Figure 10b). Neither RS nor CS can achieve anywhere near this combined high performance for *both* OLTP throughput and OLAP latency. These results demonstrate that Proteus' adaptive storage is well-suited for hybrid workloads.

We observe that for the OLAP workload Proteus has similar query latency to CS. For some queries (e.g. Query 7), Proteus takes slightly longer than CS to execute as these queries involve joins across many tables, complex predicates and aggregations. However, adaptive storage allows Proteus to remain competitive with CS on the overall OLAP workload while sustaining more than 2.2× its OLTP throughput, resulting in superior performance on the hybrid CH workload.

Without a priori knowledge of the workload, Proteus makes effective adaptive storage layout decisions because its workload models capture how data is accessed and their access costs. To understand Proteus' performance, we discuss its storage choices.

First, Proteus heavily replicates read-only fact tables, such as the *nation*, *region*, *supplier* and *item* tables. This replication enables Proteus to execute many joins and aggregations locally, thereby reducing distributed data transfer. Proteus primarily stores these tables in a compressed column format in memory. Proteus places some of the partitions storing the *item* table on disk. The TPC-C workload has skew in the *items* ordered, and Proteus judiciously places infrequently ordered *items* on disk, relying on its zone-maps when performing joins to reduce disk accesses.

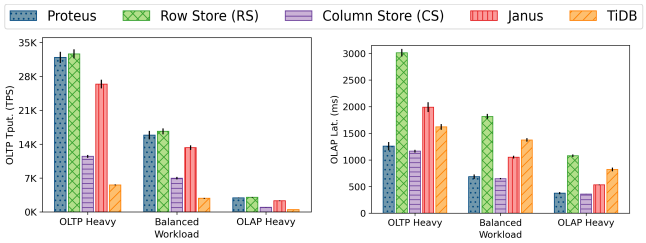
Second, as illustrated in our running example (Figure 4b), updates to the *orderline* table have a temporal relationship: recent *orderlines* are more likely to be updated. Tracking access frequencies over time allows Proteus to infer this relationship. The access patterns for the *orderline* and *order* tables are similar to the access pattern in YCSB: skewed OLTP operations access recently updated data



(a) Throughput (OLTP Heavy) (b) Throughput (Balanced) (c) Throughput (OLAP Heavy) (d) OLTP Ablation

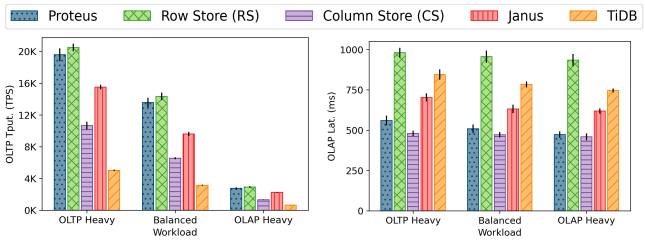
(e) Latency (OLTP Heavy) (f) Latency (Balanced) (g) Latency (OLAP Heavy) (h) OLAP Ablation

Figure 9: YCSB benchmark results over three HTAP mixes (OLTP heavy, balanced and OLAP heavy) showing OLTP throughput (9a–9c) and OLAP latency (9e–9g). 9d and 9h show ablation effects on Proteus system latency.



(a) TPC-C Throughput (b) TPC-H Query Latency

Figure 10: CH-benCHmark (CH) results showing OLTP throughput and OLAP latency for three HTAP mixes.



(a) Twitter OLTP Throughput (b) Twitter OLAP Latency

Figure 11: Twitter results showing OLTP throughput and OLAP latency for three HTAP mixes.

with uniform OLAP queries over entire tables. Thus, Proteus makes suitable storage layout decisions — it employs row format for recent data, column format for read-mostly portions of the tables, and both row and column formats for partitions with relatively balanced accesses. Proteus leverages its decision reuse capability to repeatedly make decisions for partitions with similar access statistics. Proteus achieves OLAP latency similar to that of using only CS for OLAP queries featuring predicates that examine historical order information. Storing data in columnar format provides significant advantages for queries where scan costs dominate query latency (e.g., Query 6 that aggregates a column based on multi-column predicates). Proteus' storage layout adaptions allows it to execute OLAP queries over data primarily stored in a columnar format using similar efficient execution plans as CS but *without any* pre-configuration.

Third, Proteus leverages co-access likelihoods in terms of which data items are updated or joined together, e.g., *stock* and *orderlines* belonging to the same warehouse, to co-locate the storage of these partitions to the same site. This co-location of co-accessed data minimizes the overheads of distributed processing by allowing local joins of data[3] and single-site update transactions that avoid distributed commit. Unlike competitor systems that we advantage with this information ahead of time to place data among sites, Proteus learns access patterns as the workload executes, making it robust to changes in these patterns (Section 6.3.6).

[3]Co-access frequencies are beneficial in equi-joins with foreign-key relationships, guaranteeing that if there is a local join match, then it is the sole match.

Fourth, Proteus maintains tables accessed mostly by OLTP transactions (e.g., *warehouse*, *district*, *history*) in a row-oriented format, adaptively partitioning and replicating data as necessary to mitigate contention and load effects. These storage layout decisions allow Proteus to execute OLTP transactions over data stored in a similar format to RS. By contrast, Janus' full data replication results in more data placed on the disk tier, which increases storage access costs, lowers OLTP throughput and raises OLAP query latency.

Proteus performs storage layout changes efficiently, e.g., format changes take 14 ms on average, or about as quickly as it takes for an OLTP transaction to complete. Decision reuse makes for low latency layout change plan generation (1.6 and 56 ms for OLTP & OLAP transactions on average) even though generating and executing layout change plans is infrequent Per-partition operations execute layout changes efficiently (30 and 235 ms for OLTP and OLAP transactions on average). Hence, Proteus amortizes layout change costs and spends less than 5% of its time adapting storage.

Takeaway: Proteus' learned adaptive storage techniques deliver the best overall HTAP performance while remaining competitive on both OLTP and OLAP aspects of the workload.

6.3.4 Twitter We evaluate Proteus using the Twitter benchmark and show average OLTP throughput and OLAP latency for all mixes in Figure 11. Proteus achieves throughput performance comparable with RS for all mixes (within 5% of RS OLTP throughput). Proteus has similar (within 7%) OLAP latency to CS in the balanced and OLAP heavy mixes. Remarkably, only Proteus achieves this high performance for *both aspects* of the hybrid workload.

Inserting new tweets dominates the OLTP workload, which results in significant contention on a small number of partitions. Consequently, we find that Proteus keeps recently inserted tweets in small partitions, in row format and on memory. Over time, Proteus merges these partitions into larger partitions and stores them as columns, as once inserted tweets become read-only. Consequently, OLTP transactions primarily execute over row format data while OLAP transactions execute over columnar data except for recent tweets. Moreover, Proteus rarely replicates data for this workload, so Proteus maintains all but the oldest tweets in memory, significantly improving performance over the fully replicated Janus.

The Twitter workload features a many-to-many relationship in its schema, making it difficult to partition the workload. The OLAP workload requires joining data in the presence of this many-to-many schema; for example, given a user u, get tweets from users that u is following. Data shuffling across nodes to perform joins reduces the relative effects of storage layout on OLAP latency. However, Proteus' ability to adapt data placement on nodes based on access patterns reduces the amount of data shuffling, allowing Proteus to remain competitive with CS in terms of OLAP latency while executing more than 2× as many OLTP transactions.

Takeaway: Proteus provides the best overall HTAP performance on a skewed workload and schema with a many-to-many relationship.

6.3.5 Scalability In Figure 12a, we measure OLTP throughput and OLAP latency while scaling the number of data sites from 3 to 18 in step with the number of clients (30 per site) on the balanced YCSB workload. Proteus improves OLTP throughput by 5.3× as the number of data sites grows by 6×. Increasing the number of data sites reduces OLAP query latency by 2.2×, with the steepest fall in latency occurring when the number of sites grows from 3 to 9; with 9 sites, Proteus maintains most data in memory but at 3 sites, a majority of data resides on disk.

Takeaway: Proteus is a scalable distributed HTAP system: throughput increases by more than 5× as the number of sites increases 6×.

6.3.6 Adaptivity We study Proteus' adaptive capabilities by examining its OLTP throughput and OLAP latency over time to understand its behaviour as it learns both the workload access pattern and cost model. Figure 12b shows Proteus' OLTP throughput and OLAP latency in the balanced YCSB workload. Proteus increases its OLTP throughput by 5.4× over the course of the workload while decreasing its OLAP latency by 7.9×. In this experiment, it takes Proteus roughly 3 minutes to reach within 15% of its peak OLTP throughput, and roughly 10 minutes to reach within 15% of its minimum OLAP latency. This difference is due to the skew in OLTP accesses compared to the uniform OLAP accesses; Proteus executes more layout changes for data primarily accessed by OLAP transactions. During this period, Proteus rapidly builds both its workload model to understand data access patterns and its cost model to estimate operation latencies. Even on a cold start, Proteus' cost model is accurate and averages a root mean squared error (RMSE) of 11% of the observed average latency, allowing Proteus to distinguish between good and poor layout change decisions.

Figure 12c repeats the experiment from Figure 12b with three changes: (i) the centre of the OLTP skew shifts every 5 minutes following an hourly cycle (ii) Proteus' data access latency models are initialized using the end model state resulting from the experiment in Figure 12b (iii) Proteus' access arrival estimate model is pre-trained using the historical access pattern of the workload. Compared to Figure 12b, Proteus reaches within 15% of its peak OLTP throughput in just 1 minute, and within 15% of its minimal OLAP latency in 6 minutes (Figure 12c). Slight shifts in performance are visible both before and after the 5, 10, 15 and 20-minute marks due to the workload shifts occurring at these same time points. Proteus begins executing storage layout changes predictively in anticipation of the workload shift due to high confidence in changes to the workload access pattern. The small performance shifts arise primarily due to (i) storage layout changes consuming resources and (ii) predictive storage layout changes that amortize costs over time to provide beneficial layouts for the future.

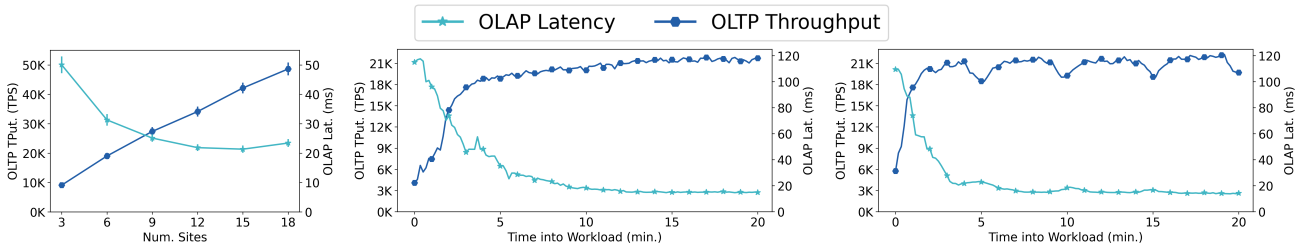
In Figure 13, we examine Proteus' ability to predict and respond to shifts in the workload mix over time. In this experiment, we follow the same methodology used for the experiments in Figure 12c but shift the workload mix every 2,000 OLAP transactions[4] in Figure 13a and every five minutes in Figures 13b and 13c. We measure workload completion time, OLTP throughput and OLAP latency over time. Observe that Proteus completes this workload faster than all of its competitors, including 1.6× faster than Janus, which does not adapt to the workload but keeps copies of all data in column and row form. Examining performance over time, we see that as in Figure 12c, Proteus rapidly improves both OLTP and OLAP performance as it adapts to the workload. A key difference between Proteus and its competitors is how Proteus behaves before the workload shift occurs: Proteus predictively and autonomously begins to change storage formats in anticipation of the workload change. For example, when shifting from the balanced to OLTP heavy mix, we see that both Proteus' OLAP latency and OLTP throughput increase. These performance changes occur as Proteus predictively executes layout changes from columnar to row format data.

Takeaway: Proteus adapts to the workload via its learned workload and cost models. Proteus leverages predicted access arrival estimates to adapt its storage layouts to changes in the workload predictively.

6.3.7 Ablation Study Figures 9d and 9h show an ablation study on Proteus' ability to make adaptive storage decisions by independently and categorically removing different techniques while using the YCSB workload. Shedding Proteus' ability to vertically and horizontally partition increases OLTP latency by 1.2× and 1.4×, respectively, as these techniques help mitigate contention effects within, and across, rows. Removing Proteus' ability to add or remove replicas also affects its OLTP latency. Proteus leverages replicas for two purposes: (i) replicating frequently updated data among sites to distribute load, and (ii) replicating data partitions with roughly equal OLTP and OLAP access frequency in both row and column format to provide storage formats for both workloads.

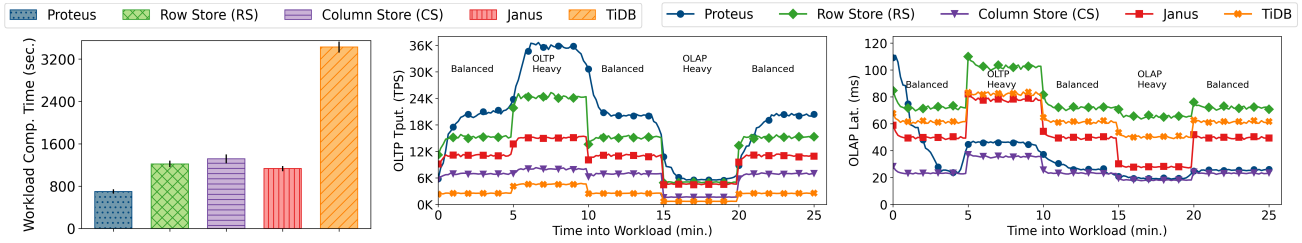
Figure 9h shows the effects of the ablation study on OLAP latency. Removing compression in Proteus increases OLAP latency by 1.7×: less data is kept in memory and cannot be operated on in compressed form. Proteus often stores columnar data using per column sort-orders as the OLAP workload features inequality predicates;

[4] 2,000 OLAP transactions are used since the same number of OLAP transactions execute over the five shifts as in the other experiments.



(a) Scalability (b) Performance Over Time (c) Performance with Shifting Hotspots

Figure 12: Sensitivity experiments using a balanced YCSB workload showing OLTP throughput and OLAP latency.



(a) Workload Completion Time (b) Performance Over Time: OLTP Throughput (c) Performance Over Time: OLAP Latency

Figure 13: Shifting workload mix experiments using YCSB — mix shifts every 5 mins over the course of the experiment.

removing this increases OLAP latency by 1.5×. Proteus' decision reuse benefit is also shown: applying execution plans and layout changes to partitions with similar access statistics reduces latency. **Takeaway:** Proteus adaptively employs different storage optimizations that reduce OLTP and OLAP latency.

7 RELATED WORK

Proteus is the first *distributed HTAP* database system that uses *adaptive* storage techniques to execute hybrid workloads efficiently.

Single node HTAP systems such as HYRISE [24] and flexible storage manager (FSM) [12] support hybrid workloads by storing a data item in either row or column-oriented format on a single node. HYRISE stores data in variable-width columns based on access frequency, while FSM stores recently written data as rows and (other) read-only data as columns. FSM uses logical tile algebra to support query processing over its data layout. By contrast, Proteus distributes data among sites and replicates data in different layouts to allow efficient OLTP and OLAP execution over the same data. H_2O [10] and OctopusDB [31] store data in both rows and columns based on the workload; however, H_2O considers only read-only workloads and materializes views in different formats to optimize scans and joins, and OctopusDB relies on user hints to decide on storage formats in simulation. Proteus is a distributed system that autonomously adapts tiered storage in both row and column formats on-the-fly by sorting, compressing, and changing data partitions and their replicas.

Distributed HTAP systems such as TiDB [27], Janus [11], F1 [64], BatchDB [43] and fractured mirrors [51] statically replicate all data in two formats: a row format master-copy for OLTP transactions and a read-only column format replica. By contrast, Proteus selectively and dynamically replicates data, leveraging replicas to store data in different storage formats, sort orders, and storage tiers. TiDB uses a cost model to decide where to execute a request but only Proteus truly adapts storage by modelling the workload.

OLAP-focused systems popularized storing data as columns [14, 56]. Column stores present optimization opportunities such as column-specific sort orders based on query predicates (cracking) [29, 30] and column-specific compression [5, 7, 66] to reduce space usage while supporting querying over compressed data. Proteus uses these techniques to execute operations over column formats efficiently while also supporting HTAP workloads. Jigsaw [33] uses prior knowledge of the workload to recursively partition data into irregular partitions to execute OLAP queries. Unlike Proteus, Jigsaw does not support updates, supports only full scans and not joins for OLAP queries, and is a single-node system without replication. Recent systems [22, 26, 39, 44, 65] adaptively partition for OLAP workloads to minimize distributed join processing or data accesses, but do not support HTAP workloads, changing storage tiers or formats, or use optimizations like compression.

OLTP-focused systems such as MorphoSys [9], E-Store [60], and STOv2 [28] partition data adaptively to mitigate contention effects. These systems focus on OLTP workloads and store data using only row formats. Proteus utilizes adaptive partitioning techniques to reduce contention while also supporting OLAP queries using column formats to execute hybrid workloads efficiently.

8 CONCLUSION

We presented Proteus, a distributed HTAP database system that adapts data storage layouts to deliver excellent performance for mixed workloads. Proteus autonomously decides on suitable row- and column-storage formats, storage tier, and whether to employ optimizations such as sorting or compression in addition to data replication and partitioning schemes. Proteus makes these decisions on-the-fly using learned workload models. Proteus reduces HTAP workload completion time by up to 70% over prior approaches while foregoing the use of static storage layouts.

Acknowledgements: This project was supported by funding from NSERC, WHJIL, CFI, and ORF.

REFERENCES

[1] 2010. The Transaction Processing Council. TPC-C Benchmark (Revision 5.11).

[2] 2018. Apache Parquet.

[3] 2018. The Transaction Processing Council. TPC-H Benchmark (Revision 2.18).

[4] 2021. API reference index | Twitter API. https://developer.twitter.com/en/docs/api-reference-index

[5] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 671–682.

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

[7] Daniel J Abadi, Daniel S Myers, David J DeWitt, and Samuel R Madden. 2007. Materialization strategies in a column-oriented DBMS. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 466–475.

[8] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. DynaMast: Adaptive dynamic mastering for replicated systems. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, IEEE, 1381–1392.

[9] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. MorphoSys: Automatic physical design metamorphosis for distributed database systems. *Proceedings of the VLDB Endowment* 13, 13 (2020), 3573–3587.

[10] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: a hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 1103–1114.

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

[12] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, New York, NY, USA, 583–598.

[13] Mike W Blasgen, Morton M Astrahan, Donald D Chamberlin, JN Gray, WF King, Bruce G Lindsay, Raymond A Lorie, James W Mehl, Thomas G Price, Gianfranco R Putzolu, et al. 1981. System R: An architectural overview. *IBM systems journal* 20, 1 (1981), 41–62.

[14] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*, Vol. 5. 225–237.

[15] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. 2008. Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services.. In *NSDI*, Vol. 8. 337–350.

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

[17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 2010 ACM symposium on Cloud Computing (SoCC)*. ACM, 143–154.

[18] George Copeland and Setreg Khoshafian. 1985. A Decomposition Storage Model. ACM Press.

[19] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.* (2010).

[20] Khuzaima Daudjee and Kenneth Salem. 2006. Lazy Database Replication with Snapshot Isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*. 715–726.

[21] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB* 7, 4 (2013), 277–288.

[22] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD/PODS '21)*. ACM, New York, NY, USA, 418–431.

[23] Anil K Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. 2015. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1716–1727.

[24] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. HYRISE: a main memory hybrid storage engine. *Proceedings of the VLDB Endowment* 4, 2 (2010), 105–116.

[25] Herodotos Herodotou and Elena Kakoulli. 2019. Automating Distributed Tiered Storage Management in Cluster Computing. *Proc. VLDB Endow.* 13, 1 (2019), 43–56.

[26] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a partitioning advisor for cloud databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 143–157.

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

[28] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for optimism in contended main-memory multicore transactions. *Proceedings of the VLDB Endowment* 13, 5 (2020), 629–642.

[29] Stratos Idreos, Martin L Kersten, and Stefan Manegold. 2009. Self-organizing tuple reconstruction in column-stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, New York, NY, USA, 297–308.

[30] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. 2007. Database Cracking.. In *CIDR*, Vol. 7. 68–78.

[31] Alekh Jindal. 2010. The mimicking octopus: Towards a one-size-fits-all database architecture. In *VLDB PhD Workshop*. 78–83.

[32] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB* 1, 2 (2008), 1496–1499.

[33] Donghe Kang, Ruochen Jiang, and Spyros Blanas. 2021. Jigsaw: A Data Storage and Query Processing Engine for Irregular Table Partitioning. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD/PODS '21)*. ACM, New York, NY, USA, 898–911.

[34] Davis E. King. 2009. Dlib-ml: A Machine Learning Toolkit. *Journal of Machine Learning Research* 10 (2009), 1755–1758.

[35] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*.

[36] Per-Ake Larson, Cipri Clinciu, Campbell Fraser, Eric N Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L Price, Srikumar Rangarajan, Remus Rusanu, et al. 2013. Enhancements to SQL server column stores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1159–1168.

[37] Per-Åke Larson, Eric N Hanson, and Susan L Price. 2012. Columnar Storage in SQL Server 2012. *IEEE Data Eng. Bull.* 35, 1 (2012), 15–20.

[38] Guy M Lohman, C Mohan, Laura M Haas, Dean Daniels, Bruce G Lindsay, Patricia G Selinger, and Paul F Wilms. 1985. Query processing in R. In *Query processing in database systems*. Springer, 31–47.

[39] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. 2017. AdaptDB: Adaptive Partitioning for Distributed Joins. *Proc. VLDB Endow.* 10, 5 (2017), 589–600.

[40] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: Scaling Transactions through Asymmetric Replication. *PVLDB* 12, 11 (2019), 1316–1329.

[41] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD/PODS '21)*. ACM, New York, NY, USA, 2530–2542.

[42] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, New York, NY, USA, 631–645.

[43] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, New York, NY, USA, 37–50.

[44] Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, and Solomon Garber. 2018. NashDB: an end-to-end economic method for elastic database fragmentation, replication, and provisioning. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, New York, NY, USA, 1253–1267.

[45] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.

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

[47] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[48] Andrew Pavlo, Matthew Butrovich, Lin Ma, Prashanth Menon, Wan Shen Lim, Dana Van Aken, and William Zhang. 2021. Make your database system dream of

electric sheep: towards self-driving operation. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3211–3221.

[49] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. 2014. Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. *Gartner* (2014).

[50] M Pezzini, D Feinberg, N Rayner, and R Edjlali. 2016. Real-time Insights and Decision Making using Hybrid Streaming, In-Memory Computing Analytics and Transaction Processing. *Gartner* (2016).

[51] Ravishankar Ramamurthy, David J DeWitt, and Qi Su. 2003. A case for fractured mirrors. *The VLDB Journal* 12, 2 (2003), 89–101.

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

[53] Zechao Shang, Xi Liang, Dixin Tang, Cong Ding, Aaron J Elmore, Sanjay Krishnan, and Michael J Franklin. 2020. CrocodileDB: Efficient Database Execution through Intelligent Deferment.. In *CIDR*.

[54] Vishal Sikka, Franz Färber, Anil Goel, and Wolfgang Lehner. 2013. SAP HANA: The evolution from a modern main-memory data platform to an enterprise application platform. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1184–1185.

[55] Mark Slee et al. 2007. Thrift: Scalable cross-language services implementation.

[56] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*. VLDB Endowment, 553–564.

[57] Michael Stonebraker and Lawrence A Rowe. 1986. The design of Postgres. *ACM Sigmod Record* 15, 2 (1986), 340–355.

[58] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. 2019. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proceedings of the VLDB Endowment* 13, 2 (2019).

[59] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. 2018. P-store: An elastic database system with predictive provisioning. In *Proceedings of the 2018 International Conference on Management of Data*. 205–219.

[60] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.

[61] Sean J Taylor and Benjamin Letham. 2018. Forecasting at scale. *The American Statistician* 72, 1 (2018), 37–45.

[62] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM International Conference on Management of Data (SIGMOD)* (Scottsdale, Arizona, USA). ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[63] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. 1997. An adaptive data replication algorithm. *ACM Transactions on Database Systems (TODS)* 22, 2 (1997), 255–314.

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

[65] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 193–208.

[66] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-scalar RAM-CPU cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 59–59.

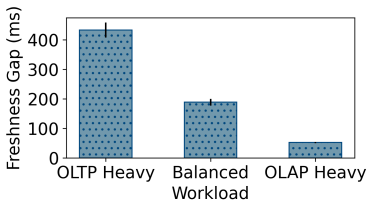


Figure 14: The freshness gap when performing OLAP scans.

A COMPUTING NET BENEFIT OF ADAPTION

Section 5.3.2 introduced how Proteus computes the net benefit of a storage layout change S, as $N(S) = \lambda(E(S) + C(S)) - U(S) > 0$, where $E(S)$ is the expected benefit of the storage layout change on upcoming queries, $C(S)$ is the expected benefit on ongoing queries, and $U(S)$ is the upfront storage cost. We now formalize the computation of $N(S)$.

Table 2 summarizes each storage layout change in Proteus and the cost functions that the ASA combines to estimate the upfront cost of the change. For example, in Figure 4b to change the storage format of partition $P1B$ from a row to column format to row requires: (i) the ASA making a network request to data site 2, (ii) locking the partition, (iii) reading all the data from the row formatted partition via a scan, and (iv) bulk loading the data into a column format. The ASA uses the layout-specific cost functions, including row-specific scan and column-specific bulk loader estimators. Recall from Section 4.4 that horizontal partitioning of rows, or vertical partitioning of columns, does not need to perform full data scans or bulk data loading, simply the reassignment of pointers. Consequently, we differentiate the upfront cost for these changes to data partitions from the generic change of partitions.

To compute the expected effect of a storage layout change S on a request T, Proteus estimates the latency of the request under the current layout $L_{current}(S, T)$ and the latency of the request under the adapted layout $L_{adapt}(S, T)$. Proteus estimates L_{adapt} by using different storage layout-aware cost functions — in our example for partition $P1B$ (Figure 4b), the ASA replaces the row format scan cost function with the sorted column scan cost function. Alternatively, storage layout changes may alter the input arguments to the same function; for example, the vertical partitioning of partition $P2C$ lowers the estimated contention of the partition, an argument for the lock acquisition cost function. Table 3 summarizes the effect that each storage layout change in Proteus and the effect that these changes have on transaction execution based on the ASA's cost functions.

Proteus weighs the estimated effect of the storage layout change on T by the likelihood of T's request arriving ($Pr(T)$), and the time to T's arrival ($\Delta(T)$). Formally we compute the estimated effect of the storage layout change S on T as $E(S, T)$, as defined in Equation 1.

$$E(S,T) = \left(L_{current}(S,T) - L_{adapt}(S,T)\right) \cdot \frac{Pr(T)}{(\Delta(T)+1)} \quad (1)$$

Observe that $E(S, T)$ is positive if $L_{current}(S, T) > L_{adapt}(S, T)$, which indicates that the ASA expects the storage layout change to reduce the latency of executing the request. However, the magnitude of $E(S, T)$ is determined by: the relative change in execution latency, how likely the request is to arrive, and the estimated time to request arrival.

Proteus computes $E(S)$ and $C(S)$, by summing $E(S, T)$ for each request that are ongoing (in which case $\Delta(T) = 0$ and $Pr(T) = 1$), or predicted to arrive. Observe that $E(S, T) \approx 0$ if $L_{current}(S, T) = L_{adapt}(S, T)$, $Pr(T) \approx 0$, or $\Delta(T)$ is sufficiently large. Consequently, Proteus restricts the set of requests that it considers to those that access data affected by the storage layout change, or are likely to arrive in an upcoming window.

B ADDITIONAL EXPERIMENTAL RESULTS

We now present additional supplementary experimental results that explain different aspects of Proteus' behaviour.

B.1 OLAP Freshness Gaps

Proteus targets real-time analytical processing, where transactional updates are immediately available to analytical queries. To measure the effectiveness of Proteus at providing real-time analytical processing, we measure the average freshness gap of OLAP queries. Specifically, we modify our YCSB benchmark so that: (i) OLTP transactions set every updated value to a timestamp, (ii) OLAP queries return the smallest value read in the scan (i.e., oldest timestamp observed). We record the values set by OLTP transactions and values returned by OLAP transactions, along with the (real) time that the OLTP transaction committed and the OLAP transaction began. After the experiment, we combine this observed state and record the difference between (i) the smallest value read in each OLAP query (oldest timestamp observed) and (ii) the most recent commit time before the beginning of the OLAP transaction that updated values in the range of the OLAP scan. Hence, if the OLAP scan reads the freshest data available, we record 0 and otherwise record a value that indicates how stale the OLAP scan was. The average of these values represents Proteus' *freshness gap*.

Figure 14 presents the freshness gap for Proteus over the three YCSB workloads. Observe that in the balanced YCSB workload, Proteus' freshness gap is less than 200 ms. In the OLTP heavy and OLAP heavy workloads, the freshness gap is approximately 450ms and 50ms, respectively. Hence, OLAP queries in Proteus observe a fresh state, satisfying the real-time analytical processing requirement. Proteus' low freshness gap is primarily due to (i) efficient update propagation and (ii) performing OLTP transactions directly on column data if OLAP queries primarily access the data. Proteus has a higher freshness gap for the OLTP heavy workload because it propagates and applies more updates to replicas.

Takeaway: Proteus has a low freshness gap for OLAP queries and hence provides real-time analytical processing.

B.2 Storage Layout Change Operation Overhead

Table 4 summarizes the proportion of time spent on transactions and storage layout change operations in the balanced CH-benCHmark (CH) workload. Proteus balances the proportion of time spent executing OLTP and OLAP transactions, demonstrating the benefit of using OLTP and OLAP-specific thread-pools to execute requests. These storage layout changes are efficient as they execute about as quickly as an OLTP transaction. The most frequent changes involve changing formats, tiers, and partition

Storage Layout Changes	Cost Function						
	Bulk Load	*Scan*	*Sort*	*Network Request*	*Lock Acquisition*	*Waiting for Updates*	*Commit*
Change Format	✓	✓		✓	✓		
Change Tier	✓	✓		✓	✓		
Sort	✓	✓	✓	✓	✓		
Compress	✓	✓		✓	✓		
Change Partitioning (Horizontal in Row) (Vertical in Column)				✓	✓		✓
Change Partitioning (Generic)	✓	✓	✓ *(If necessary)*	✓	✓		✓
Add Replica	✓	✓		✓ *(Source & Dest.)*	✓ *(Dest.)*	✓	
Remove Replica				✓			
Change Master				✓ *(Source & Dest.)*	✓ *(Source & Dest.)*	✓	✓ *(Source & Dest.)*

Table 2: The upfront costs of different storage layout changes, which are computed by combining different cost functions.

Storage Layout Changes	Cost Function										
	Bulk Load	*Insert/ Update/ Delete*	*Point Read*	*Scan*	*Sort/ Hash*	*Join*	*Aggregate*	*Network Request*	*Lock Acquisition*	*Waiting for Updates*	*Commit*
Change Format	✓	✓	✓	✓	✓	✓	✓				
Change Tier	✓	✓	✓	✓	✓	✓	✓				
Sort	✓	✓	✓	✓	✓	✓	✓				
Compress	✓	✓	✓	✓	✓	✓	✓				
Change Partitioning	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
Change Replication			✓	✓	✓	✓	✓	✓		✓	✓
Change Master								✓			✓

Table 3: The expected change in transaction execution latency per cost function for each storage layout change.

Operation	Proportion of Time Spent	Avg. Latency (ms)	Frequency (per 1000)
OLTP Transaction	47.1%	6.37 ± 0.1	991 ± 24
OLAP Transaction	48.1%	685 ± 40	9.4 ± 0.6
Storage Format Change	2.14%	14.0 ± 1.1	20.4 ± 0.8
Storage Tier Change	0.51%	12.9 ± 0.8	5.3 ± 0.2
Sort or Comp. Change	0.04%	20.7 ± 1.6	0.26 ± 0.01
Partition Change	0.07%	4.56 ± 0.4	2.2 ± 0.12
Replication Change	1.96%	36.9 ± 2.9	7.1 ± 0.6

Table 4: The proportion of time spent, average latency and frequency per operation for the CH-Benchmark workload.

replicas. In Table 5, we summarize the proportion of time spent planning transaction execution, layout change plans, and executing layout change plans. We find that Proteus' decision reuse allows efficient selection of physical execution plans for transactions, which take about 1% of overall system time. For fewer than 10% of transactions, Proteus generates a layout change plan. Layout change plans in response to OLAP transactions take longer to develop than OLTP transactions due to the number of data items accessed in OLAP transactions. Finally, about 1% of transactions execute layout change plans, which execute in 30 ms and 235 ms on average for OLTP and OLAP transactions, respectively. These plans execute quickly due to the low latency of individual layout change operators. Together, Proteus spends less than 5% of the time planning and performing storage layout change plans as Proteus amortizes the costs of layout changes across transactions.

Takeaway: Planning and executing storage layout changes is efficient and has low overhead on system performance.

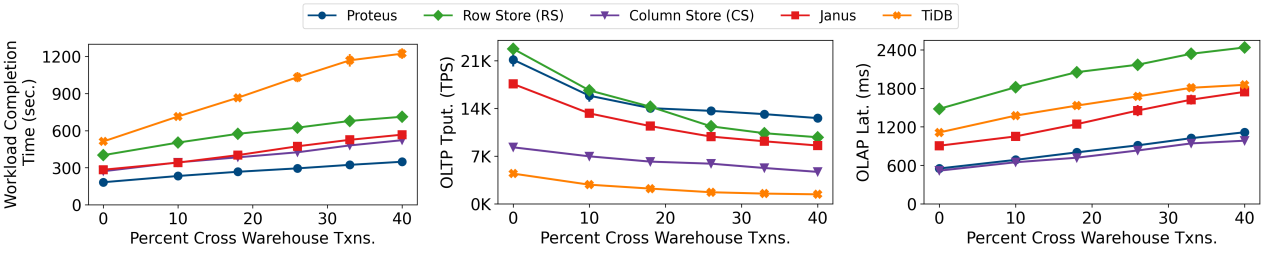
B.3 CH Cross Partition Transactions

We next study Proteus' sensitivity to the percentage of cross warehouse transactions in CH. Recall that in CH, clients are associated with warehouses, and *NewOrder* transactions vary the stock of ordered items, which are kept on a per-warehouse basis. By default, 10% of *NewOrder* transactions place orders to a different warehouse than the client's warehouse. Because of this locality, the best data placement scheme as determined by Schism [19] is co-locating data by the warehouse. Consequently, as the number of cross warehouse transactions increases, OLTP transactions increasingly become distributed transactions. Moreover, increasing the percentage of cross warehouse transactions also increases the number of distributed joins, as several TPC-H queries (e.g. Query 7) join orders with stocks, which using Schism's data placement are not co-located at the same site.

Figure 15 shows the experimental results for CH we vary the percentage of cross-warehouse transactions. Figure 15a shows the workload completion time as the percentage of cross-warehouse transactions increases. Observe that the workload completion time for all systems increase as the percentage of cross-warehouse transactions increases because the performance of both OLTP transactions (Figure 15b) and OLAP transactions (Figure 15c) decreases. However, Proteus' relative reduction in workload completion time to its next closest competitors increases from 1.45× to 1.63× as the percentage of cross warehouse transactions increases from 0% to 40%. Proteus achieves this relative increase in overall performance because Proteus adapts its storage layout, resulting in OLTP

Operation	Proportion of Time Spent	Avg. Latency (ms)	Frequency (per 1000)
OLTP Physical Execution Plan Generation	1.32%	0.18 ± 0.01	991 ± 24
OLAP Physical Execution Plan Generation	0.88%	12.7 ± 1.1	9.4 ± 0.6
OLTP Layout Change Plan Generation	1.02%	1.62 ± 0.8	84.9 ± 5.7
OLAP Layout Change Plan Generation	1.14%	56.8 ± 4.2	2.7 ± 0.34
OLTP Layout Change Execution	3.01%	30.8 ± 3.7	13.1 ± 0.96
OLAP Layout Change Execution	1.19%	235 ± 27	0.68 ± 0.03

Table 5: The proportion of time spent, average latency and frequency of planning and executing layout changes for the CH-Benchmark (CH) workload.



(a) CH Workload Completion Time **(b) CH OLTP Throughput** **(c) CH OLAP Latency**

Figure 15: Experiments that vary the percentage of cross warehouse transactions in the CH-benCHmark (CH).

throughput outperforming the RS by nearly 1.3× and achieving OLAP latency within 13% of the CS. Specifically, we find that Proteus increasingly replicates warehouse, district, customer and stock data among sites, which allows Proteus to (i) reduce distributed join processing for OLAP queries, and (ii) dynamically change data mastership efficiently to reduce 2PC, allowing OLTP transactions to execute more efficiently. Proteus can perform this data replication because the decrease in OLTP throughput decreases the growth rate in the amount of stored data and because Proteus does not mandate data replication in two formats like Janus or TiDB.

Takeaway: Proteus maintains its performance advantage as the percentage of cross partition transactions increases.