Optimal Column Layout for Hybrid Workloads

Manos Athanassoulis[*]
Boston University
mathan@bu.edu

Kenneth S. Bøgh[†]
Uber Technologies Inc.
bgh@uber.edu

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

ABSTRACT

Data-intensive analytical applications need to support both efficient reads and writes. However, what is usually a good data layout for an update-heavy workload, is not well-suited for a read-mostly one and vice versa. Modern analytical data systems rely on columnar layouts and employ delta stores to inject new data and updates.

We show that for hybrid workloads we can achieve close to one order of magnitude better performance by tailoring the column layout design to the data and query workload. Our approach navigates the possible design space of the physical layout: it organizes each column's data by determining the number of partitions, their corresponding sizes and ranges, and the amount of buffer space and how it is allocated. We frame these design decisions as an optimization problem that, given workload knowledge and performance requirements, provides an optimal physical layout for the workload at hand. To evaluate this work, we build an in-memory storage engine, Casper, and we show that it outperforms state-of-the-art data layouts of analytical systems for hybrid workloads. Casper delivers up to 2.32× higher throughput for update-intensive workloads and up to 2.14× higher throughput for hybrid workloads. We further show how to make data layout decisions robust to workload variation by carefully selecting the input of the optimization.

PVLDB Reference Format:
Manos Athanassoulis, Kenneth S. Bøgh, Stratos Idreos. Optimal Column Layout for Hybrid Workloads. *PVLDB*, 12(13): 2393-2407, 2019.
DOI: https://doi.org/10.14778/3358701.3358707

1. INTRODUCTION

Modern data analytics systems primarily employ columnar storage because of its benefits when it comes to evaluating read-heavy analytic workloads [1]. Examples include both applications and systems that range from relational systems to big data applications like Vertica [48], Actian Vector (formerly Vectorwise [84]), Oracle [47], IBM DB2 [17], MS SQL Server [51], Snowflake [30], and in the Apache Ecosystem, the Apache Parquet [9] data storage format.

[*]Part of this work was done while the author was a postdoctoral researcher at Harvard University.

[†]Work done while the author was a visiting student at Harvard University.

Proceedings of the VLDB Endowment, Vol. 12, No. 13
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3358701.3358707

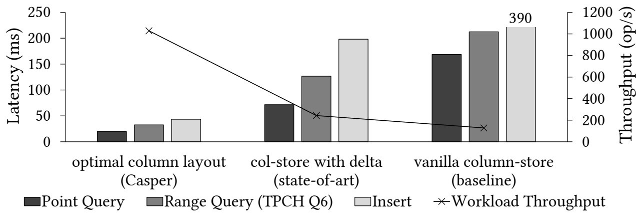


Figure 1: Existing analytical systems have 2× higher performance than vanilla column-stores on hybrid workloads by utilizing a delta-store. Using a workload-tailored optimal column layout, Casper brings an additional 4× performance benefit.

The HTAP Evolution. Analytical applications are quickly transitioning from read-mostly workloads to hybrid workloads where a substantial amount of write operations also needs to be supported efficiently. These hybrid transactional analytical workloads (HTAP) typically target state-of-the-art analytics, while still supporting efficient data ingestion [62, 66]. New designs targeting HTAP workloads are coming both from academic research projects [10, 11, 45, 49, 54, 58, 64, 67] and commercial products like SAP Hana [33, 68, 77], Oracle [61], Microsoft SQL Server [50], and MemSQL [76].

The Problem: Conflicting Design Goals for HTAP. One big challenge when designing systems for HTAP workloads comes from the fact that data layout decisions designed for read-intensive scenarios do not typically work well for write-intensive scenarios and vice versa. Finding the optimal layout can give a massive performance benefit, however, existing data systems come with a fixed design for many core data layout decisions. This means that they are locked into a specific behavior and are unable to approach the theoretical optimal, thus, missing out on significant benefits.

The Solution: Learning Column Layouts. Our insight is that there are certain design choices for which we should not be making a fixed *a priori* decision. Instead, we can learn their optimal tuning to efficiently support HTAP workloads with a single copy of the data. In modern analytical systems there is a set of decisions that always makes sense. For example, nearly all systems choose to store columns as fixed-width arrays, which helps with hardware-conscious optimizations, like vectorization, SIMD processing, and compression. Other decisions are typically fixed, though in practice, the optimal choice depends on the workload. In particular, in this paper, we focus on three prominent design decisions: (i) how to physically order data [40, 78], (ii) whether columns should be dense, and (iii) how to allocate buffer space for updates (e.g., delta-store [38, 48] and ghost values [18, 19, 44]). We show that systems should be tunable along those decisions for a given application, and we demonstrate how to do this in an optimal and robust way.

Example. Figure 1 shows the performance of a workload with both transactional (point queries and TPC-H updates) and analytical (TPC-H Q6) access patterns, when executed using three different approaches: base column-stores (without any write optimizations), state-of-the-art columnar layouts with a delta store, and a workload-tailored optimal column layout proposed by our system, Casper. We implemented all approaches in a state-of-the-art column-store system that exploits parallelism (using all 32 cores). Our implementation supports fast scans that employ tight `for` loops, multi-core execution, and work over compressed data.

The state-of-the-art approach, which sorts the queried columns and maintains a delta-store, leads to a 1.9× increase in throughput when compared to the baseline column-store. In comparison, the optimal layout determined by Casper leads to 8× improvement in overall performance. Casper's physical design decisions combine fine-grained partitioning with modest space for update buffering (1% of the data size in this experiment) which is distributed per partition (details for the partitioning process in §4 and §5).

Challenge 1: Fast Layout Discovery. The fact that we can see massive improvements when we tailor the layout of a system to the access patterns is not by itself surprising. There are several challenges though. First, finding this optimal layout is expensive: in the worst case, an exponential number of data layout strategies needs to be enumerated and evaluated.

To be able to make fast decisions, we formulate the data layout problem as a binary optimization problem and we solve it with an off-the-shelf solver [8]. We keep the optimization time low by dividing the problem into smaller sub-problems. We exploit that columns are physically stored in column chunks, and we partition each chunk independently, thus reducing the overall complexity by several orders of magnitude (§6.3).

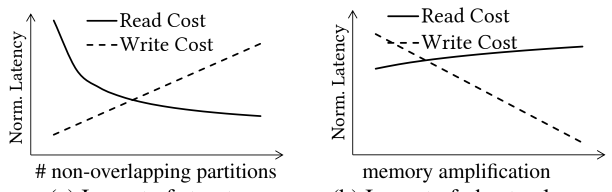
Challenge 2: Workload Tailoring. Second, we rely on having a representative workload sample. We analyze this sample to quantify the access frequency and the access type for each part of the dataset. As part of our analysis, we maintain workload access pattern statistics for both read operations (point queries and range queries) and write operations (inserts, deletes, and updates).

The column layouts can be tailored along many dimensions. The design space covers unordered columns to fully sorted columns, along with everything in between; that is, columns partitioned in arbitrary ways. We also consider the update policy and ghost values, which balance reads and writes with respect to their memory amplification [15]. Ghost values are empty slots in an otherwise dense column, creating a per-partition buffer space. Our data layout scheme creates partitions of variable length to match the given workload: short granular partitions for frequently read data, large partitions to avoid data movement in frequently updated ranges, while striking a balance when reads and updates compete.

Challenge 3: Robustness. Third, when tailoring a layout to a given workload, there is the chance of overfitting. In practice, workload knowledge may be inaccurate and for dynamic applications may vary with time. A system with a layout that works perfectly for one scenario may suffer in light of workload uncertainty. We show that we can tailor the layout for a workload without losing performance up to a level of uncertainty.

Positioning. We focus on analytical applications with relatively stable workloads which implies that a workload sample is possible to achieve. For example, this is the case for most analytical applications that provide a regular dashboard report. In this case, our tool analyzes the expected workload and prepares the desired data layout offline, similar to index advisors in modern systems [3, 83].

For more dynamic applications with unpredictable workloads, an adaptive solution is more appropriate. Our techniques can be extended for such dynamic settings as well, by periodically analyzing the workload online (similar to how offline indexing techniques were repurposed for online indexing [24]) and reapplying the new format if the expected benefit crosses a desired threshold.



(a) Impact of structure (b) Impact of ghost values

Figure 2: Accessing a column is heavily affected by the structure of the column (e.g., sorted, partitioned). Read cost (a) logarithmically decreases by adding structure (partitions), while using ghost values to expand the column layout design space (b) reduces write cost linearly w.r.t. memory amplification.

Contributions. Our contributions are summarized below:

- We describe the design space of column layouts containing partitioning, update policy, and buffer space (§2), a rich design space that supports a diverse set of workloads.
- We introduce (i) the *Frequency Model* that overlays access patterns on the data distribution (§4.2); (ii) a detailed cost model of operations over partitioned columns (§4.4); and (iii) an allocation mechanism for ghost values (§4.6).
- We formalize the column layout problem as a workload-driven, binary integer optimization problem that balances read and update performance, supports service-level agreements (SLAs) as constraints, and offers robust performance even when the training and the actual workloads are not a perfect match (§5).
- We integrate the proposed column layout strategies in our storage engine Casper (§6). We show that Casper finds the optimal data layout quickly (§6.3), and that it provides up to 2.3× performance improvement over state-of-the-art layouts (§7).

2. COLUMN LAYOUT DESIGN SPACE

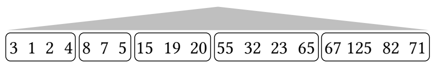
In this section, we define the problem and the scope of the proposed approach. State-of-the-art analytical storage engines, store columns either sorted based on a sort key (i.e., one of the columns), or following insertion order. On the other hand updates are usually applied out-of-place using a global buffer [1, 39, 48, 84]. Casper explores a richer design space that (i) uses range partitioning as an additional data organization scheme, (ii) supports in-place, out-of-place, and hybrid updates, and (iii) supports either *no*, *global*, or *per-partition* buffering as shown in Table 1. By considering a wider design space, we allow for further performance optimizations without discarding the key design principles of analytical engines that benefit hybrid analytical workloads.

Table 1: Design space of column layouts.

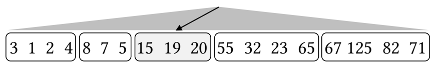
Data Organization	Update Policy	Buffering
(a) insertion order	(a) in-place	(a) none
(b) sorted	(b) out-of-place	(b) global
(c) partitioned	(c) hybrid	(c) per-partition

Scope of the Solution. Casper targets applications with a hybrid transactional analytical nature, with an arbitrary yet static workload profile. Casper uses this workload knowledge to find a custom-tailored column layout that outperforms the state of the art.

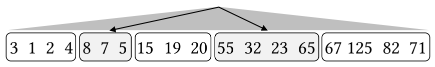
Horizontal Partitioning vs. Vertical Partitioning. Casper's design space includes the traditional global buffering approach of write-stores [78] and operates orthogonally to schemes that employ arbitrary vertical partitioning (e.g., tiles [11]). As a result, our



(a) A range partitioned column.

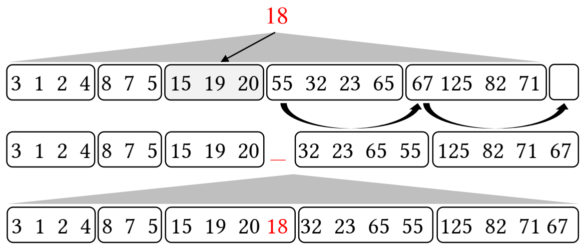


(b) Looking for value 15.

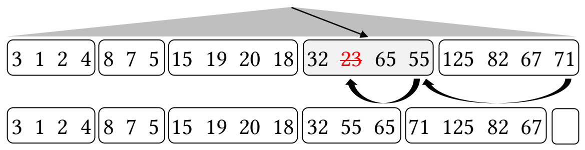


(c) Looking for values in $[6, 43)$.

Figure 3: Maintaining range partitions in a column chunk allows for fast execution of read queries. For point queries (b), the partition that may contain the value in question is fully scanned. For range queries (c) the partitions that (may) contain the first and the last element belonging to the range are scanned, while any intermediate partitions are blindly consumed.



(a) Inserting value 18.



(b) Deleting value 23.

Figure 4: Inserting and deleting data in a partitioned column chunk uses rippling and restricts data movement.

column layout strategies can also be used for tables with tiles or projections [78]. In the case of projections, the read workload is distributed amongst the different projections. As a result, the column layout can be further tailored for each projection leading to potentially greater benefits.

Partitioning as an Update Tuning Knob. The core intuition about why range partitioning can work as a way to balance read and write costs is twofold: read queries with small selectivity favor high number of partitions (more structure) because then they will only read the relevant data, while updates, insert, and deletes, favor a low number of partitions (less structure) because each such operation can operate by moving data across partition boundaries. With respect to read-only workloads, different partitioning strategies have different impact. If a workload has different access patterns in different parts of the domain, equi-width partitioning can lead to unnecessary reads. Narrow partitions are needed for the parts of the data that correspond to point queries, or to the beginning/end of range queries. On the other hand, for the part of the domain which is less frequently queried, coarser partitioning is enough.

The impact of adding structure to the data on the read and write costs is conceptually shown in Figure 2a. For example, when a column chunk with M_C elements is partitioned in k partitions, the estimated cost of a point query is on average the cost of reading $\frac{M_C}{k}$ elements, assuming equi-width partitions. On the other hand, the cost of inserts (and deletes) is on average $k/2$. Hence, the number of partitions k is an explicit tuning knob between read and update performance. Ideally, however, a locally optimized partitioning scheme would allow a workload with skewed accesses in different parts of the domain to achieve optimal latency.

Ghost Values. Delete, insert, and update approaches require data movement that can be reduced if we relax the requirement to have the whole column contiguous. Allowing deletes to introduce empty slots in a partition results in a delete operation that only needs to find the right partition and flag the value as deleted. To better optimize for future usage, the empty slot is moved to the end of the partition to be easily consumed when an insert arrives, or when a

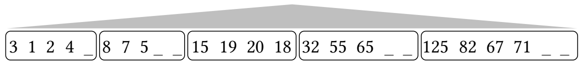


Figure 5: Adding ghost values allows for less data movement; inserts use empty slots and deletes create new ones.

neighboring partition needs to facilitate an insert and has no empty slots of its own. These empty slots, called *ghost values*, require extra bookkeeping but allow for a versatile and easy-to-update partitioned column layout with per-partition buffering. Ghost values expand the design space by reducing the update cost at the expense of increased memory usage, trading space amplification for update performance [14, 15].

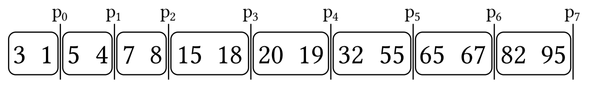
Workload-Driven Decisions. Employing range partitioning and ghost values allows Casper to have a tunable performance that can accurately capture the requirements of hybrid workloads. Casper adjusts the granularity of partitions and the buffer space to navigate different performance profiles. In particular, by increasing the number of partitions, we achieve logarithmically lower read cost at the expense of a linear increase of write cost (Fig. 2a), and by increasing the buffer space, we achieve linearly lower write cost at the expense of a sublinear read performance penalty (Fig. 2b). Contrary to the previous workload-driven approaches, Casper does not care only for what types of operations are executed and at what frequency. Rather, it takes into account the *access pattern distribution with respect to the data domain*, hence making fine-grained workload-driven partitioning decisions. Overall, Casper collects detailed information about the access distribution of each operation in the form of histograms with variable granularity and uses it to inform the cost models that we develop in the following sections.

3. ACCESSING PARTITIONED COLUMNS

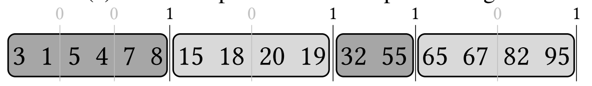
In this section, we describe Casper's standard repertoire of storage engine operations. We further provide the necessary background on operating over partitioned columns, as well as the building blocks needed to develop the intuition about how to use partitioning for balancing read and write performance. Casper supports the fundamental access patterns of point queries, range queries, deletes, inserts, and updates [57]. For the following discussion, we assume a partition scheme P with k partitions of variable size. A fixed-cost light-weight partition index is also kept in the form of a shallow k-ary tree (Fig. 3a).

Point Queries. The partition index provides the partition ID, and then the entire partition is fully consumed with a tight `for` loop scan to verify which data items of the partition qualify, as shown in Figure 3b. The cost of the query is given by the shallow index probe and the scan of the partition. Once the corresponding values are verified the query returns the positions of the qualifying values or directly the values (depending on the API of the select operator) to be consumed by the remaining operators of the query plan.

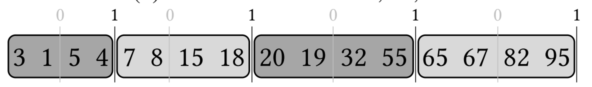
Range Queries. Similarly to point queries, a range query over partitioned data needs to probe the shallow index to find the first and the last partitions of the query result (Fig. 3c). The first and the last partitions are filtered, while the rest of the partitions are accessed when the result is materialized and can simply be copied to the next query operator as we know that all values qualify.



(a) General representation of a partitioning.



(b) Partition boundaries 8, 20, 55.



(c) Partition boundaries 5, 18, 55.

Figure 6: Representing different partitioning schemes (b) and (c) with block size $B = 2$.

Inserts. An insert to a range partitioned column chunk can be executed using the ripple-insert algorithm [41] causing $O(k)$ data accesses (Fig. 4a). When a value needs to be inserted in a partition, the ripple-insert brings an empty slot from the end of the column to that partition. Starting from the last partition, it moves the first element of each partition to the beginning of the next partition. The first movement happens between the first element of the last partition and the (already) available empty slot at the end of the column. If no empty slots are available, the column is expanded. Then, starting from the last partition and moving toward the partition targeted by the insert, the first elements of consecutive partitions are swapped, in order to create an empty slot right after the end of the partition receiving the insert. The new value is inserted there and the partition boundaries of all trailing partitions are moved by one position. When inserting in the m^{th} partition the overall cost is $(k-m)$, and, if data is uniform, $k/2$ on average.

Deletes. When a value (or row ID) to delete is received, the first step is to identify the partition that may hold this value using the shallow index. The next step is to identify the value or values to be deleted and move them to the end of the partition. Then, following the inverse of the previous process, the empty slots are moved towards the end of the column, where they can remain as empty slots for future usage. The data movement is now $(k-m)\cdot del\_card$, where m is the partition to delete from, and del_card is the number of values that qualified for deletion.

Updates. An update is handled as a delete followed by an insert. This approach is followed in state-of-the-art systems and allows for code re-use. Alternatively, the shallow index is probed twice to find the source (update from) and the destination (update to) partitions, followed by a direct ripple update between these two partitions.

4. MODELING COLUMN LAYOUTS

In this section, we provide the necessary modeling tools to view the column layout decision as an optimization problem, which captures the workload access patterns and the impact of ghost values. We model the operations over partitioned columns with ghost values (§4) and we show how to find the optimal column layout (§5).

Problem Setup. Each of the five operations may have an arbitrary frequency and access skew. In this general case, the ideal column layout forms partitions that minimize the overall workload latency, taking into account that each operation is affected differently from layout decisions. For ease of presentation, we first build a model without ghost values, incorporating them later on.

We formalize this as an optimization problem, where we want to find the partitioning scheme P that minimizes the overall cost of executing a workload W over a dataset D:

$$\arg\min_P \; cost(W,D,P) \tag{1}$$

This formulation takes into account both the dataset and a sample workload and aims to provide as output the ideal partitioning. It does so by first combining the distribution of the values of the domain with the distribution of the access patterns of the workload to form the *effective access distribution*, by overlaying the access patterns on the data distribution. In order to formalize this problem accurately, we first need to define a way to represent an arbitrary partitioning scheme (§4.1), and subsequently, a way to overlay on it the representation of an arbitrary workload (§4.2). Next, we present a detailed cost model for each workload operation (§4.4). Finally, we consider how to allocate ghost values (§4.6).

4.1 Representing a Partitioning Scheme

We represent a partitioning scheme by identifying the positions of values that mark a partition boundary using a bit vector. In general, we can have as many partitions as the number of values in the column, however, there is no practical gain in having partitions smaller than a cache-line for main-memory processing. In fact, having a block size of several cache-lines is often more preferable because it naturally supports locality and sequential access performance. Note that duplicate values should be in the same partition.

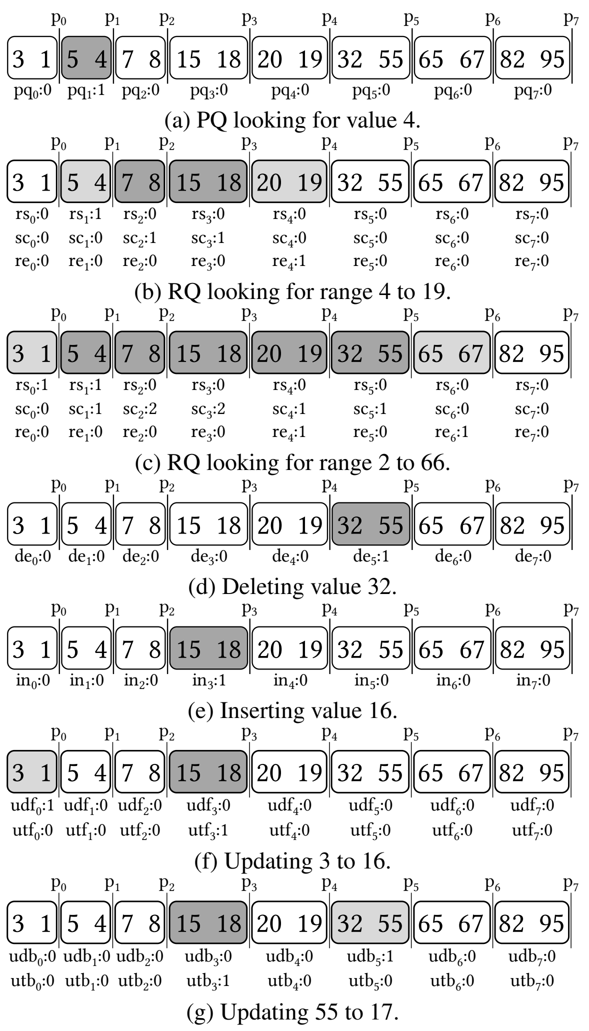
A column is organized into N_b equally-sized blocks. The size of each block is decided in conjunction with the chunk size to guarantee that the partitioning problem is feasible (more in §6.3). We represent a partitioning scheme by N_b Boolean variables $\{p_i$, for $i = 0, 1, \ldots, N_b - 1\}$. Each variable p_i is set to one when block i serves as a partition boundary, i.e., a partition ends at the end of this block. Note that the first block is before p_0. The exact block size is tunable to any multiple of cache-line size, and affects the level of the detail of the final partitioning we provide. The smaller meaningful value is a cache-line, but in practice, a much coarser level of granularity is sufficient. Figure 6a shows an example dataset with blocks of size two and eight Boolean variables $\{p_0, \ldots, p_7\}$. Figures 6b and 6c show two different partitioning schemes. In Figure 6b, the first partition is three blocks wide, the second spans two blocks, the third is a single block, and the fourth spans two blocks. Figure 6c shows another partitioning scheme with four partitions, each one two blocks wide. This scheme captures any partitioning strategy, at the granularity of the block size or any coarser granularity chosen.

4.2 The Frequency Model

We now present a new access pattern representation scheme on top of the partitioning scheme of the previous section. The accessed data is organized in logical blocks (like in Figure 6) and the access patterns of each operation per block are documented. The size of a logical block is tunable, which allows for a variable resolution of access patterns, which, in turn, controls the partitioning overhead. The basic access patterns are produced by the five operations, however, each one causes different types of accesses. Next, we describe in detail the information captured from a sample workload.

Given a representative sample workload and a column split into its blocks we capture which blocks are accessed and by which operation, forming a set of histograms. We refer to this set of histograms as the *Frequency Model* (*FM*) because it stores the frequency of accessing each part of the domain, which translates to access patterns in the physical layer. *FM* captures the accesses on the blocks, while the common cost to locate a partition is not kept as it is shared for each operation. The overall idea is that we capture accesses to each individual column block, in order to synthesize these blocks into partitions in a way that maximizes performance for the specific effective access distribution.

While we model five general operations (point, and range queries, deletes, inserts, and updates), the different ways of accessing each



p0 p1 p2 p3 p4 p5 p6 p7
3 1 | 5 4 | 7 8 | 15 18 | 20 19 | 32 55 | 65 67 | 82 95
pq0:0 pq1:1 pq2:0 pq3:0 pq4:0 pq5:0 pq6:0 pq7:0
(a) PQ looking for value 4.
rs0:0 rs1:1 rs2:0 rs3:0 rs4:0 rs5:0 rs6:0 rs7:0
sc0:0 sc1:0 sc2:1 sc3:1 sc4:0 sc5:0 sc6:0 sc7:0
re0:0 re1:0 re2:0 re3:0 re4:1 re5:0 re6:0 re7:0
(b) RQ looking for range 4 to 19.
rs0:1 rs1:1 rs2:0 rs3:0 rs4:0 rs5:0 rs6:0 rs7:0
sc0:0 sc1:1 sc2:2 sc3:2 sc4:1 sc5:1 sc6:0 sc7:0
re0:0 re1:0 re2:0 re3:0 re4:1 re5:0 re6:1 re7:0
(c) RQ looking for range 2 to 66.
de0:0 de1:0 de2:0 de3:0 de4:0 de5:1 de6:0 de7:0
(d) Deleting value 32.
in0:0 in1:0 in2:0 in3:1 in4:0 in5:0 in6:0 in7:0
(e) Inserting value 16.
udf0:1 udf1:0 udf2:0 udf3:0 udf4:0 udf5:0 udf6:0 udf7:0
utf0:0 utf1:0 utf2:0 utf3:1 utf4:0 utf5:0 utf6:0 utf7:0
(f) Updating 3 to 16.
udb0:0 udb1:0 udb2:0 udb3:0 udb4:0 udb5:1 udb6:0 udb7:0
utb0:0 utb1:0 utb2:0 utb3:1 utb4:0 utb5:0 utb6:0 utb7:0
(g) Updating 55 to 17.

Figure 7: Frequency model in action. Here we show for each operation which partitions are accessed, and consequently, which histogram buckets are updated.

block are, in fact, more complex. *FM* utilizes ten histograms, each storing the frequency of a different sub-operation: (i) *pq* counts block accesses for each *point query*, (ii) *rs* counts block accesses for each *range query start*, (iii) *re* counts block accesses for each *range query end*, (iv) *sc* counts full block *scans* for each range query, (v) *de* counts *deletes* for each block, (vi) *in* counts *inserts* to each block, (vii) *udf* and (viii) *udb* store when a block contains a value to *update from* and it generates a *forward/backward* ripple, and, (ix) *utf* and (x) *utb* storeswhen a block contains a value to *update to* and it generates a *forward/backward* ripple.

Update operations are captured by a set of four histograms to account for the possible ripple action between the block we update from and the block we update into. In addition, we differentiate a forward ripple from a backward ripple to account for subtle modeling details (discussed in §4.4). Each of the ten histograms contains a counter per block to reflect the respective accesses to this block, i.e., each bin of the histogram corresponds to a block. When calculating the histograms from a sample workload, we do not actually materialize the results or modify the data; instead, we capture the access patterns as if each operation is executed on the initial dataset.

Example. We now explain how the histograms are populated using a series of examples in Figure 7. A point query touches a single block, which increments the relevant bin of the *pq* histogram. For example a point query for value 4 would be simulated by incrementing pq_1 (Fig. 7a). In the general case, a range query touches several blocks. In particular, one access is documented for the first block of the range at *rs*, one access for the last block of the range at *re*, and one access for each block between the two increasing the value of the *sc* bins of the intermediate blocks. For example, a query looking for values $4 \leq v \leq 19$ will increment rs_1, sc_2, sc_3, and re_4, because the range query starts in block #1, scans block #2 and block #3, and finishes in block #4 (Fig. 7b). When another range query is documented, the buckets that are accessed by both will be further incremented (Fig. 7c). For each delete operation, the *de* histogram corresponding to the block that contains the value to be deleted is incremented. Thus, the deletion of the value 32 increments de_5 (Fig. 7d). Inserts increment the *in* histogram corresponding to the block that the value would be inserted to. For example, inserting 16 in the previous example results in incrementing in_3 (Fig. 7e). Finally, updates increment *udf* for the old value and *utf* for the new value, or *udb* and *utb* respectively. The choice depends on the relationship of the old value with the new value. If the new value is *larger*, then *udf* and *utf* are incremented, and when the new value is *smaller*, *udb* and *utb* are incremented. For example, updating the value 3 to be 16 results in incrementing udf_0 and utf_3 (Fig. 7f). Similarly, updating 55 to 17 results in incrementing udb_5 and utb_3 (Fig. 7f).

The Frequency Model is built once all operations of the sample workload have contributed to the bins of the ten histograms. The ten histograms are represented by ten vectors. The rs_i bins form the vector $\overrightarrow{RS} = \{rs_0, rs_1, \ldots, rs_{N-1}\}$, and similarly, the vectors $\overrightarrow{RE}$, $\overrightarrow{SC}$, $\overrightarrow{PQ}$, $\overrightarrow{DE}$, $\overrightarrow{IN}$, $\overrightarrow{UDF}$, $\overrightarrow{UTF}$, $\overrightarrow{UDB}$, and $\overrightarrow{UTB}$ represent the corresponding histograms. Using these vectors we build a cost-model that provides the cost of each operation for partitions of arbitrary size (multiple of block size) in §4.4.

Columns and Column-Groups. For ease of presentation our running examples depict only one column, however, the same analysis can be directly applied to groups of columns since the Frequency Model is oblivious to whether each data item consists of a single value or multiple. Thus, the Frequency Model can generate the *effective access pattern* of a workload over a table resulting in multi-dimensional dynamic range partitioning. Once the workload access patterns are documented, any further decisions are taken based on the Frequency Model alone. We collect information on a per block basis regardless of the contents of the block.

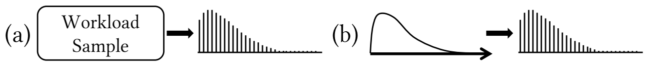


Figure 8: Learning *FM* from (a) samples or (b) distributions.

4.3 Learning the *FM* from Access Patterns

The previous discussion assumes that the Frequency Model uses a sample workload to estimate the histograms of each operations as shown in Figure 8a. The Frequency Model, however, can also use statistical knowledge of the workload access patterns to create access distribution histograms with the desired granularity. Having estimated the distribution of the access pattern of each operation as well as the data distribution, we can efficiently construct a histogram with variable number of buckets as shown in Figure 8b. By default, each bucket corresponds to a memory block, however, this parameter is tunable. Finer granularity leads to better performance, at the cost of longer optimization runtime, and vice versa. In addition, Casper can combine the two approaches and benefit from the recent advancements in learning data distributions from the workload [46], and estimate online the access pattern distributions during normal operation.

4.4 Cost Functions

Now we model the cost of each access operation. Assuming a dataset with M values, and using block size B we have up to $N = \lceil \frac{M}{B} \rceil$ partitions. We assume that every partitioning scheme can support up to N partitions, which in practice, depends on M and can be controlled by choosing B. We assume that accessing blocks comes at a cost following a standard I/O model where we have four main access patterns: random read RR, random write RW, sequential read SR, and sequential write SW. The exact values are determined by micro-benchmarking of the in-memory performance of the system as well as the selected block size. In addition, once an operation falls within a block we account for the cost of reading the whole block because there is no further navigation structure within a block. Finally, while the initial partitioning is only on logical block boundaries, during workload execution the partition boundaries may move freely, allowing for variable partitioning granularity. Next, we introduce the cost functions.

Range Queries. Every range query scans all partitions that may contain values that are part of the result. Using a lightweight partition index we find the first, last, and all intermediate partitions. We differentiate between the three types because the first and the last may contain values that are not part of the result, while all the intermediate partitions are known to contain only qualifying values and the query simply needs to scan them. In practice, this depends on the select operator. For example, to return positions we do not need to scan the middle pieces, however, we do need to scan the corresponding pieces of other columns in this projection if they are part of the selection. We also need to scan the middle pieces of the accessed column if the query needs to fetch its contents.

When the first and the last partitions contain many values that are not part of the result, the range scan performs unnecessary reads. A range query starting at a given block is only negatively impacted if there are blocks before it that belong to the same partition. Consider the third block in Figure 7c. The relevant counter is rs_2. If p_1 is not set, then each range query starting in the third block will have to perform at least one additional read (the leading block) as compared to the optimal scenario. If both p_0 and p_1 are not set then every query starting in the third block will always read the first two blocks without having any qualifying values. On the other hand, if p_1 is set, then no unnecessary reads will be incurred. This is captured mathematically using $rs_2 \cdot RR + rs_2 \cdot SR \cdot ((1-p_1) + (1-p_1)(1-p_0))$. This means that every range query that begins in the third block will pay the cost of a random read to reach its first block and then, depending on where the partition boundaries are present, additional sequential read costs (that can be avoided in the "ideal" partitioning). The second term uses the binary representation of the partition boundaries in $\overrightarrow{P}$. Thus, each of the products will only evaluate to one if all the p_i's contained in it are equal to zero. This also captures that these are sequential reads. This second term is generalized in Eq. 2. Finally, the cost of accessing the block at the beginning of the range is given in Eq. 3.

$$bck\_read(i) = \sum_{j=0}^{i-1} \prod_{k=j}^{i-1} (1-p_k) \tag{2}$$

$$cost\_rs(rs_i) = rs_i \cdot RR + rs_i \cdot SR \cdot bck\_read(i) \tag{3}$$

The end of the range query (re) has a dual behavior. Consider the fifth block in the example in Figure 6a. If there is no partition boundary in p_4 the range queries ending at the fifth block will have to read one more block than what is optimal. If neither p_4 nor p_5 is set, the queries will do two additional reads. We describe this similar pattern as forward reads, and we capture it in Eq. 4. The equation is very similar to Eq. 2, having, however, different limits to reflect that forward reads are towards the end of the column. The overall range end cost is captured in Eq. 5, which is again similar to Eq. 3. The main difference is all reads, including the first, are sequential, because this cost refers to the end of a range query.

$$fwd\_read(i) = \sum_{j=0}^{N-i-1} \prod_{k=i}^{N-j-1} (1-p_k) \tag{4}$$

$$cost\_re(re_i) = re_i \cdot SR + re_i \cdot SR \cdot fwd\_read(i) \tag{5}$$

Finally, all the intermediate blocks are scanned irrespectively of the partitioning scheme, with cost as shown in Eq. 6.

$$cost\_sc(sc_i) = sc_i \cdot SR \tag{6}$$

Thus, given *any* partitioning, we can compute the cost of the range queries of the sample workload, by computing the sum of $cost\_rs$, $cost\_sc$, and $cost\_re$ for all blocks:

$$\sum_{i=0}^{N-1} cost\_rs\,(rs_i) + cost\_sc\,(sc_i) + cost\_re\,(re_i)$$

Point Queries. Every point query scans the entire partition that the value may belong to, because there is no information as to in which part of the partition the value may reside. Ideally, a point query would scan only a single block, however, in case of a partition that spans multiple blocks, all will be scanned. For example, when searching for value 7 in the data shown in Figure 6a, if p_1 and p_2 are partition boundaries (set to one) then the point query will read only one block. If, however, $p_0 = p_1 = p_2 = 0$ and only $p_3 = 1$ then this point query will read all four blocks. The same analysis holds for empty point queries. In general, the cost of a point query is derived similarly to the cost of a range query. If a partition boundary on either side of a given block is not present, it adds one additional read to all point queries of that block, when compared to the optimal. Thus, the definitions of fwd_read and bck_read are re-used to compute the cost of the point queries of a given block (Eq. 7). A point query always incurs at least one random read (the ideal case), which is captured by the first term. Any unnecessary reads are captured by the last two terms.

A key observation of the point query cost is that the more partitions we have (i.e., the more structure imposed on the data), the cheaper the point query becomes, since $fwd\_read(i)$ and $bck\_read(i)$ will have smaller values (0 if a partition is a single block).

$$\begin{aligned} cost\_pq(pq_i) = {} & pq_i \cdot RR + \\ & pq_i \cdot SR \cdot (fwd\_read\,(i) + bck\_read\,(i)) \end{aligned} \tag{7}$$

Inserts. Having adopted the model of partitioned chunks, each insert operation within a chunk is modeled using the ripple insert algorithm as discussed in Section 3 and shown in Figure 4a. An insertion involves touching two blocks in the last partition, and one block in each of the other partitions between the last and the one we insert into. In every partition, a read and a write operation takes place. Contrary to the cost of read queries, the higher the number of partitions the higher the cost of inserts will be (unless they all belong to the last partition). Hence, in general, inserts and read queries favor different layouts, and a partitioning must strike a balance between the two. The cost of an insert is affected differently by the partition boundaries than that of the read operations. Indeed, *any* partition that is present after a given block will affect the cost of inserting in that block, which is captured in Eq. 8. Since each p_i is always either zero or one, the sum captures the number of partitions trailing the given partition. The insert cost is captured in Eq. 9, where for every partition trailing the one we insert into, one block is read and one is written. The term before the sum captures that there is one additional random read and one random write (in the last partition). Since each access "jumps" from partition to partition, all reads and writes are random.

$$trail\_parts(i) = \sum_{j=i}^{N-1} p_j \tag{8}$$

$$cost\_in(in_i) = in_i \cdot (RR + RW) \cdot (1 + trail\_parts\,(i)) \tag{9}$$

Deletes. Similarly to inserts, a delete maintains a dense column by rippling the deleted slot to the end of the column. An example is shown in Figure 4b and the access patterns of a delete are depicted in Figure 7d. In order to find whether there is a value to delete a point query is first executed. Consequently, a "hole" is created in the partition containing the deleted value, which is eventually rippled to the end of the column. The data access patterns are similar to that of an insert operation, however, they are not exactly the same. In general, a delete touches two blocks in the partition we are deleting from, and one in each subsequent partition. In addition, since a point query is first needed to locate the partition containing the deleted value, deletes face conflicting requirements. The initial point query favors small partitions to have a small number of unnecessary reads, and the actual delete operation favors fewer (and thus larger) partitions in order to minimize the cost of rippling. Compared to inserts that favor few partitions, deletes favor more complex partitioning decisions, affected by access skew. These observations are reflected in the cost of a delete.

$$cost\_rd(de_i) = de_i \cdot RW + de_i\,(RR + RW) \cdot trail\_parts\,(i) \tag{10}$$

Each delete requires a point query, followed by the cost of the ripple delete. The ripple delete accounts for the fact that the block containing the deleted value is always updated, and the $trail\_parts(i)$ term accounts for the cost incurred by each partition between the one we deleted from and the last. The ripple delete cost is shown in Eq. 10. The overall cost of a delete operation is given by the point query and the ripple delete cost as shown in Eq. 11, which contains the partitioning representation in the form of the p_i variables.

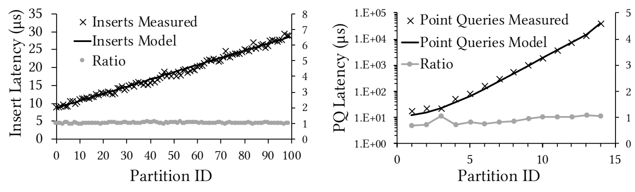
$$cost\_de(de_i) = cost\_pq(de_i) + cost\_rd(de_i) \tag{11}$$

Updates. Finally, we consider updates. In many systems updates are implemented as deletes followed by inserts [41]. A more efficient approach is to perform a ripple update directly from the source block to the target block. We model the latter. For example, as shown in Figures 7f and 7g, an update requires a point query to retrieve the source partition, and then it can ripple either forward or backward until it reaches the target partition. Regardless of the direction, the ripple always touches one block per partition, except for the first partition, in which it touches two blocks – the extra block being the one needed to touch to move the hole to the end of the partition. The Frequency Model captures this by recording deletes incurred by updates (*udf*, *udb*), and the corresponding ripple inserts towards the target blocks (*utf*, *utb*).

Assuming a general update operation of a value that exists in block i to a value that should be placed to block j (w.l.o.g. $i < j$), the update cost will entail a point query to find the old value, and a delete action that will move the newly created hole at the end of the partition: $cost\_pq(udf_i) + udf_i \cdot (RW + RR + RW)$. Next, we need to account for rippling the hole from block i, toward block j (since $i < j$ rippling the hole forward). The number of partitions between block i and j is equal to the sum $p_i + \cdots + p_{j-1}$, which is equal to $trail\_parts(i) - trail\_parts(j)$. Hence, if we keep track of all the blocks we update from (*udf*) and all the blocks we update to (*utf*), the cost is:

$$\begin{aligned} cost\_udf(udf_i) &= cost\_pq(udf_i) + udf_i \cdot (RR + 2 \cdot RW) && (12)\\ &\quad + udf_i \cdot (RR + RW) \cdot trail\_parts(i) \\ cost\_utf(utf_i) &= -\,utf_i \cdot (RR + RW) \cdot trail\_parts(i) && (13)\end{aligned}$$

Similarly, when $i >= j$, the update operation ripples backward hence the sign of the trailing partitions calculation is reversed. Note that the case $i = j$ is correctly handled by either pair of equations; by convention, we pick the latter.



(a) Inserts. (b) Point queries.

Figure 9: Cost model verification for (a) inserts and (b) point queries (10M chunk, exponentially increasing partition size).

$$\begin{aligned} cost\_udb(udb_i) &= cost\_pq(udb_i) + udb_i \cdot (RR + 2 \cdot RW) && (14)\\ &\quad - udb_i \cdot (RR + RW) \cdot trail\_parts(i) \\ cost\_utb(utb_i) &= utb_i \cdot (RR + RW) \cdot trail\_parts(i) && (15)\end{aligned}$$

Overall Workload Cost. After deriving the cost for each operation per block, we can now express the total workload execution cost as a function of any given partition. This cost is given by Eq. 16, which adds the previously missing link of the workload cost in the optimization problem defined in Eq. 1. The Frequency Model now captures the data distribution and the access patterns of the operations and the remaining free variable is the partitioning vector $\overrightarrow{P}$.

$$\begin{aligned} cost\left(\overrightarrow{P}, \mathcal{FM}\right) = \sum_{i=0}^{N-1} \Big(& fixed\_term_i + bck\_term_i \cdot bck\_read(i) \\ & + fwd\_term_i \cdot fwd\_read(i) + parts\_term_i \cdot trail\_parts(i) \Big) \end{aligned} \tag{16}$$

where, $\forall i = 0, 1, \ldots, N-1$:

$$\begin{aligned} fixed\_term_i &= RR \cdot (rs_i + pq_i + in_i + de_i + 2 \cdot udf_i + 2 \cdot udb_i) + \\ & SR \cdot (re_i + sc_i) + RW \cdot (in_i + de_i + 2 \cdot udf_i + 2 \cdot udb_i) \\ bck\_term_i &= SR \cdot (rs_i + pq_i + de_i + udf_i + udb_i) \\ fwd\_term_i &= SR \cdot (re_i + pq_i + de_i + udf_i + udb_i) \\ parts\_term_i &= (RR + RW) \cdot (in_i + de_i + udf_i - utf_i - udb_i + utb_i) \end{aligned} \tag{17}$$

and $\mathcal{FM} = \{\overrightarrow{RS}, \overrightarrow{SC}, \overrightarrow{RE}, \overrightarrow{PQ}, \overrightarrow{IN}, \overrightarrow{DE}, \overrightarrow{UDF}, \overrightarrow{UTF}, \overrightarrow{UDB}, \overrightarrow{UTB}\}$.

Note that the total cost in Eq. 16 is a function of quantities that either (a) depend on the Frequency Model only ($fixed\_term_i$, bck_term_i, fwd_term_i, $parts\_term_i$) or (b) depend on the partitioning strategy and the Frequency Model ($bck\_read(i)$, $fwd\_read(i)$, $trail\_parts(i)$).

4.5 Cost Model Verification

The cost model captures the cost of the basic storage engine operations by accurately quantifying the cost of random and sequential block memory accesses. As a result, the model only needs to tune very few parameters (the random and sequential memory block access costs for read and write operations). For every instance of Casper deployed, we first need to establish these values through micro-benchmarking. Subsequently, we can verify the accuracy of the overall cost model. It suffices to verify the estimated cost of insert and point read operations because they contain the two main cost functions of the model: (i) the number of trailing partitions for each partition (Eq. 8), and (ii) the size of each partition, i.e., the sum of fwd_read (Eq. 4) and $back\_read$ (Eq. 2).

Figure 9 shows that the proposed model accurately quantifies the costs of each operation using the fitted constants of the model. The random read latency and the random write latency in a memory block is 100ns, while the sequential read is amortized, leading to 14× lower cost per block. We further observe that the partition index has a cumulative cost of 8.5μs per operation, which is shared across all operations and does not influence the partitioning process. Figure 9a shows the measured and the model-based insert cost when using chunks of 10M elements, each with 100 partitions. The experiment corroborates a strong linear relation of the cost to the number of trailing partitions. Figure 9b shows the measured and the model-based point query cost. In this experiment, the chunk has 15 partitions with exponentially increasing size, ranging from 2^9 elements for the first partition to 2^{22} elements for the last partition. Here the experiment supports a strong linear relation between the cost of a point query and the partition size. Note that in both figures the grey points indicate the ratio between the measured and the model-based cost which is always very close to 1.0 (y-axis on the right hand-side).

4.6 Considering Ghost Values

Ghost Values Benefit Performance. Ghost values enable a trade-off between memory utilization and data movement. In particular, delete operations simply create a hole in the partition they target. Insert operations do not need to create space by rippling if there are available ghost values. Finally, update operations can create a new empty slot in the partition they update from and simply use available slots to store the new values.

Distributing Ghost Values to Partitions. Given $\overrightarrow{P}$, $\mathcal{FM}$, and a total budget of ghost values GV_{tot}, the goal is to distribute them in a way that minimizes the overall cost. The operations that can benefit from having ghost values in the partitions they target are *inserts* and *updates*. For every partition that an operation inserts into, a ghost value can help to entirely avoid the ripple insert. Similarly, for every update operation (either with forward or backward ripple), the partitions that have incoming updates will avoid a ripple insert (in the worst case) by using the available ghost values. To cover this worst case, in the remainder of the ghost value analysis, we consider that every *insert* and *update to* operation requires a ripple insert. The distribution of ghost values for block i, $GV_{alloc}(i)$ is given by Eq. 18, which uses the data movement per block as a result of inserts and updates ($dm\_part(i)$), as well as the total data movement (dm_tot), to distribute ghost values proportionally to the performance benefit they offer.

$$GV_{alloc}(i) = \frac{dm\_part(i)}{dm\_tot} \cdot GV_{tot} \tag{18}$$

5. OPTIMAL COLUMN LAYOUT

Minimizing Total Workload Cost. Using the column layout cost model, we formulate the following optimization problem:

$$\begin{aligned}
\text{minimize} \quad & \textstyle\sum_{i=0}^{N-1} \Big(fixed\_term_i \\
& \quad + bck\_term_i \cdot \textstyle\sum_{j=0}^{i-1} \prod_{k=j}^{i-1} (1-p_k) \\
& \quad + fwd\_term_i \cdot \textstyle\sum_{j=0}^{N-i-1} \prod_{k=i}^{N-j-1} (1-p_k) \\
& \quad + parts\_term_i \cdot \textstyle\sum_{j=i}^{N-1} p_j \Big) \\
\text{subject to} \quad & p_{N-1} = 1 \\
& p_i \in \{0,1\}, \ i = \in \{0,1,\ldots,N-2\}
\end{aligned} \tag{19}$$

The constraint $p_{N-1} = 1$ guarantees that the dataset forms at least one partition (zero partitions has no real meaning), and $p_i \in \{0,1\}$ guarantees that each p_i is a binary variable.

The binary optimization problem formulated above, however, contains products between groups of variables, and cannot be solved by linear optimization solvers. Products between variables can be replaced by new variables and additional constraints [22]. The final binary linear optimization problem formulation is shown in Eq. 20. Note the new constraints on the two-dimensional variables $y_{i,j}$, which guarantee that each $y_{i,j}$ corresponds to the product it replaced.

Performance Constraints. The minimization in Eq. 20 is also augmented with performance SLAs. In particular, an application may

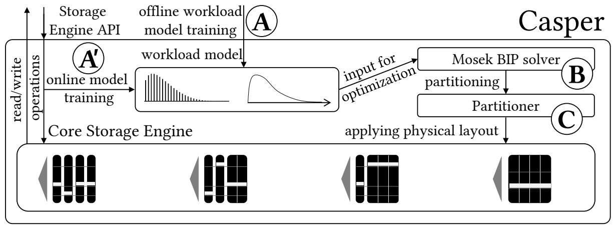


Figure 10: System architecture. (A) Casper uses offline workload information, (B) solves the BIP and (C) applies the partitioning. (A′) During execution monitoring, if the access patterns change, a re-partitioning cycle is triggered.

require read queries to respect a $read_{SLA}$, or updates to respect an $update_{SLA}$. Such constraints are expressed as a function of $\overrightarrow{P}$.

$$\begin{aligned}
\text{minimize} \quad & \textstyle\sum_{i=0}^{N-1} \Big(fixed\_term_i + bck\_term_i \cdot \sum_{j=0}^{i-1} y_{j,i-1} \\
& \quad + fwd\_term_i \cdot \textstyle\sum_{j=0}^{N-i-1} y_{i,N-j-1} \\
& \quad + parts\_term_i \cdot \textstyle\sum_{j=i}^{N-1} p_j \Big) \\
\text{subject to} \quad & p_{N-1} = 1 \\
& p_i \in \{0,1\}, \ i = \in \{0,1,\ldots,N-2\} \\
& y_{i,i} = 1 - p_i, \ i = \in \{0,1,\ldots,N-1\} \\
& y_{i,j} \le 1 - p_j, \ i,j = \in \{0,1,\ldots,N-1\}, i < j \\
& y_{i,j} \ge 1 - \textstyle\sum_{k=i}^{j} p_k \\
& y_{i,j} \in \{0,1\}
\end{aligned} \tag{20}$$

Update Latency Constraint. An update/insert SLA dictates that all update/insert operations are faster than a maximum allowed latency $update_{SLA}$. The most expensive operation has to ripple through all partitions, so this constraint can be expressed as follows:

$$(RR+RW) \cdot (1 + \sum_{i=0}^{N-1} p_i) \le update_{SLA} \Rightarrow \sum_{i=0}^{N-1} p_i \le \frac{update_{SLA}}{RR+RW} - 1$$

Read Latency Constraint. A SLA for read queries dictates the maximum read cost for a point query, which in turn, is quantified by the size of the maximum allowed partition size (MPS), as follows:

$$RR + SR \cdot MPS = read_{SLA} \Rightarrow MPS = \frac{read_{SLA} - RR}{SR}$$

In order to ensure that the largest partition is at most MPS blocks wide, we need to make sure that for every MPS consecutive p_i's at least one of them is equal to 1. In other words $\forall j = 0,\ldots,N-MPS$: $\sum_{i=0}^{MPS-1} p_{j+i} \ge 1$.

Overall, the performance constraints augment the binary linear optimization formulation with the following bounds:

$$\begin{aligned}
\text{bounds} \quad & \textstyle\sum_{i=0}^{N-1} p_i \le \frac{update_{SLA}}{RR+RW} - 1 \\
& \forall j = \{0,1,\ldots,N-MPS\} : \textstyle\sum_{i=0}^{MPS-1} p_{j+i} \ge 1 \\
& \quad \text{where } MPS = \frac{read_{SLA}-RR}{SR} - 1
\end{aligned} \tag{21}$$

Adding the bounds of Eq. 21 to the optimization problem in Eq. 20 completes the formalization of partitioning as a binary linear optimization problem, for which we use the Mosek solver [8].

6. Casper STORAGE ENGINE

We implement the Casper storage engine in C++. Casper fully supports all five access patterns described in Section 3, effectively being a drop-in replacement for any relational scan operator with support for updates. Casper supports all the common access patterns, hence, it provides general support for accessing data for relational operators. The size of the code added to implement the Casper column layout decision method described in Sections 4 and 5 is in the order of 4K lines of code. Figure 10 shows the overall architecture of Casper. The key components are (i) the Frequency Model that maintains an accurate estimation of the access patterns over the physical storage, (ii) the optimization component that employs the state-of-the-art solver Mosek [8], (iii) the partitioner that implements the physical layout decisions, and (iv) the core storage engine that implements the update and access operations over the partitioned data. Casper naturally supports multi-threaded execution since the column layouts create regions of the data that can be processed in parallel without any interference. Some of the operations require to update the contents of other partitions (when a ripple is necessary), hence, correct execution needs a way to protect the atomicity of each operation through snapshot isolation or locking. Ghost values allow Casper to reduce contention by allowing an update, delete or insert, to affect in the best case, only one partition and avoid rippling.

6.1 Transaction Support

Hybrid workloads consist of long-running analytical queries and short-lived transactions. The systems that support hybrid workloads must ensure that the long running queries are executed without being affected by the transactions, neither with respect to performance, nor the correctness of the values read. Casper supports general transactions through snapshot isolation [20], which isolates a snapshot of the database observed at the beginning of each transaction, and works only on that.

Snapshot Isolation through Multi-version Concurrency Control. Recent HTAP systems and storage engines employ variations of multi-version concurrency control (MVCC) [11, 58, 74, 76] that allows for snapshot isolation [20]. Casper takes a similar approach: each transaction is allowed to work on the data by assigning timestamps to every row when inserted or updated, initially maintained in a local per-transaction buffer. For the frequent cases, the short-lived transactions will be operating over disjoint sets of rows hence there will be no conflicts. In the rare case that multiple transactions are trying to access the same object, the first one to commit wins and the other transactions abort and roll back.

Reducing the Ripple Contention. In order to reduce the contention of moving ghost values to the partition we are inserting into, (i) Casper moves a block of ghost values every time one is necessary, which can be used by neighboring partitions in the future, and (ii) we decouple the ghost value rippling from the transaction since it does not affect correctness. Hence, even if a transaction is rolled back, the already completed fetching of ghost values will persist and will benefit future inserts or updates.

6.2 Compression

Casper natively supports the *dictionary* and *frame-of-reference* (or delta) compression schemes, the most commonly employed in modern column-store data systems [1, 2, 85]. First, *dictionary* compression is supported by Casper as-is. The performance constants of the model are changed to reflect the compressed data size, and Casper's behavior remains qualitatively the same. Second, when *delta* encoding is used, a synergy between the partitioning and the compression effort is created. In fact, Casper tends to finely partition areas that attract more queries, thus, enabling better delta compression since the value range of small partitions is also small. This has a multiplicative impact on the savings in memory bandwidth per item. The more we read a partition the more compressed it is, leading to less overall data movement. Casper compresses our micro-benchmark data by 2.5× and TPC-H data by 4.5×.

Another compression approach used in columnar systems is run-length encoding (RLE) [2]. RLE requires the data to be sorted in order to calculate the run lengths, and it always requires a more expensive "decoding" step when updating (similar to bit-wise RLE for bitmap indexes [16]). RLE often has better compression rate

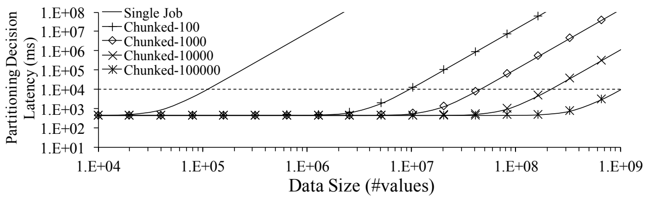


Figure 11: Casper partitioning scales with data size.

than dictionary or delta compression, however, it has higher construction cost (it needs sorting), higher execution cost (decoding) and higher update cost (decode/re-encode). Hence, typically dictionary and delta compression schemes are preferred over RLE [85].

6.3 Scalability with Problem Size

Casper formulates and solves a binary integer problem with N binary variables, having an exponential (2^N) solution space. Modern optimization techniques allow to search this space in polynomial time. In particular, we use the optimization problem solver Mosek, which relaxes non-convex problems using semidefinite optimization [8]. This allows the Casper column layout decision process to have cubic complexity. For a dataset with M values, and B values per block, Casper needs $O((M/B)^3)$ time. In order to maintain the column layout decision cost low, we can either increase the block size or divide the problem into smaller sub-problems by creating C chunks. The former produces solutions with lower quality (coarser partitioning granularity), hence, using column chunks is preferred. The histograms are created per chunk, and, similarly, design decisions are made for each chunk without any need for communication with other chunks. This allows us to arbitrarily reduce the partitioning complexity. For a level of parallelism equal to CPU, we execute CPU sub-problems at the same time and exploit their embarrassingly parallel nature for an overall cost $O((C/CPU)\cdot(M/(B\cdot C))^3)$. In practice, we find the optimal layout of a 10^9-element relation using 64 cores and a block size of 4096 bytes, in about 10 seconds (Fig. 11), while the estimated time without chunking and parallelism is 10^{15} seconds.

Variable Histogram Granularity. Casper can vary the granularity of the histograms to match a multiple of the page size either by aggregating histogram values, or by using data and access distributions to calculate coarser granularity histograms (following §4.3).

Locating Partitions. Casper stores partition-wise metadata to be able to find and discard partitions for each query. For every partition, the minimum and the maximum value of the domain they cover is stored, along with positional information within the chunk. The metadata enable efficient searching with a k-ary search tree. If the chunk size is small or the average partition size is large, the number of partitions is small enough to fit in the higher levels of cache memory. In that case, the metadata can be treated as Zonemaps [35, 55] and they can be very efficiently scanned.

6.4 Casper as a Generic Storage Engine

Casper implements a repertoire of standard storage engine API calls including (i) scanning an entire column (or groups of columns), (ii) search for a specific value, (iii) search for a specific range of values, (iv) insert a new entry, and (v) update or delete an existing entry. This generic API is supported by systems that target either analytical or transactional workloads and is compatible with state-of-the-art storage engines [57]. As a result, Casper can be easily integrated into existing systems. Note that *mixed workloads* refer to interleaving long read queries with short updates/inserts. Similarly to both transactional-optimized and analytical-optimized storage engines, Casper supports all operations. Using workload

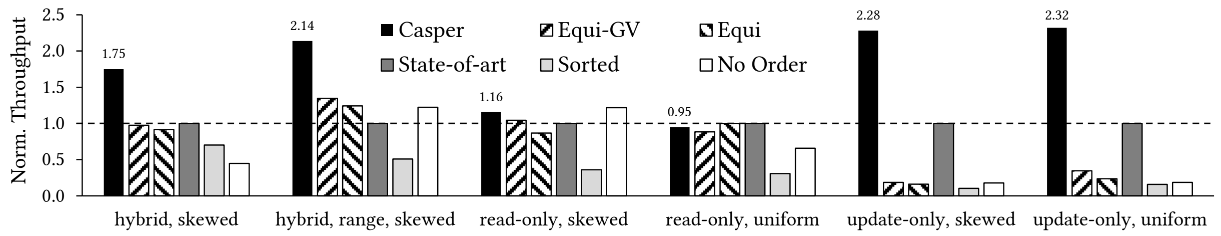


Figure 12: Casper matches or significantly outperforms other column layouts approaches for a variety of workloads (experiments with 16 threads, chunks of 1M values, block size 16KB, and ghost values 0.1%).

knowledge, Casper offers physical layouts for hybrid workloads that outperform state-of-the-art physical designs.

Multi-Column Range Queries. Casper natively supports multi-column range queries. Similar to state-of-the-art storage engines, Casper evaluates the first (typically the most selective) filter and retrieves the qualifying positions to evaluate the subsequent filters (further accelerated using Zonemaps [55]). We refer to Figure 1 for an experiment with multi-column range queries (TPC-H Q6 [81]) that shows Casper outperforming the state of the art by 2.5×.

7. EXPERIMENTAL EVALUATION

Experimental Setup. We deploy Casper in a memory-optimized server from Amazon EC2 (instance type m4.16xlarge) with a 64-bit Ubuntu 16.04.3 on Linux 4.4.0. The machine has 256GB of main memory and two sockets each having an Intel Xeon E5-2686 v4 processor running at 2.3GHz with 45MB of L3 cache. Each processor supports 32 hardware threads (a total of 64 for the server).

Experimental Methodology. For fair comparison, Casper integrates all tested column layout strategies. In particular, Casper has six distinct operation modes: (i) plain column-store with no data organization (*No Order*), (ii) column-store with sorted data in the leading column (*Sorted*), (iii) state-of-the-art update-aware column-store with sorted columns and a delta store for incoming updates (*State-of-art*), (iv) column-store with equi-width partitioned data (*Equi*), (v) column-store with equi-width partitioned data and ghost values (*Equi-GV*), and (vi) *Casper* that puts everything together. For fairness and in order to have low experimental design complexity, we allow Casper to have as many partitions as the equi-width partitioning schemes, but it has the freedom to allocate their sizes according to the optimization problem.

Column Chunks. The storage engine uses a column-chunk based layout. Each column is not a single contiguous column; instead, it is a collection of column chunks, each one stored and managed separately. This technique is employed by numerous modern systems giving flexibility for updates [30], and serves as the main state-of-the-art competitor in our experimentation. We use column chunks that hold 1M values each. Unless otherwise noted, we use 16 threads. Each experiment comprises of 10000 operations. We repeat each experiment multiple times, and we report measurements with low standard deviation.

7.1 HAP Benchmark

We first describe the evaluation benchmark. Since there is currently a scarcity of benchmarks for hybrid workloads we develop our own benchmark that we call Hybrid Access Patterns (HAP) benchmark. HAP is based on the ADAPT benchmark [11], and is composed of queries that are common in enterprise workloads [68], as well as transactional (small) general update queries that are typically tested in hybrid storage engines [25, 31].

The HAP benchmark has two tables, a narrow table with 16 columns and a wide one with 160 columns. Each table contains tuples with a 8-byte integer primary key a_0 and a payload of p 4-byte columns $(a_1, a_2, \dots, a_p)$. In all of our experiments, we load 100M tuples in the database in order to report a steady-state performance.

The benchmark consists of six queries: (Q_1) a *point query* that requests the contents of a row, (Q_2) an *aggregate range query* that counts the rows in a range, (Q_3) an *arithmetic range query* that sums a subset of attributes of the selected rows, (Q_4) an *insert* query that adds a new tuple in the relation, (Q_5) a *delete* query that deletes a specific tuple, and (Q_6) an *update* query that fixes an error in a data entry by correcting its primary key value. The SQL code for each of the queries is available below:

Q_1: **SELECT** $a_1, a_2, \dots, a_k$ **FROM** R **WHERE** $a_0 = v$
Q_2: **SELECT** $count(*)$ **FROM** R **WHERE** $a_0 \in [v_s, v_e)$
Q_3: **SELECT** $a_1 + a_2 + \cdots + a_k$ **FROM** R **WHERE** $a_0 \in [v_s, v_e)$
Q_4: **INSERT INTO** R **VALUES** $(a_0, a_1, a_2, \dots, a_p)$
Q_5: **DELETE FROM** R **WHERE** $a_0 = v$
Q_6: **UPDATE** R **SET** $a_0 = v_{new}$ **WHERE** $a_0 = v$

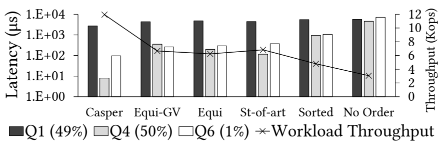
With different values of k, v, v_s, and v_e, we test various different aspects of the workloads including projectivity, selectivity, overlap between queries, and hot and cold regions of the domain. Unless otherwise noted, we experiment with datasets of 100M tuples and 16 columns, with uniformly distributed integer values.

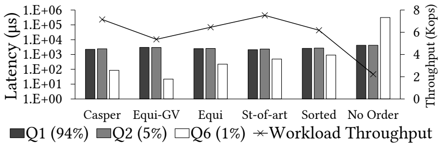
Logical and Physical Benchmarking. We develop a specialized benchmark for storage engines for hybrid workloads, which we view as a "physical" benchmark for the supported access paths. Similar to recent research on new storage engines, we stress-test the access path and the update performance [11, 25, 28, 31].

To stress different aspects of Casper, we test various combinations of queries Q_1 through Q_6. In particular, we synthesize three different types of workloads: (i) *hybrid*, (ii) *read-only*, and (iii) *update-only*. We have two versions of the hybrid workload, one with Q_1 and Q_4 that has equality searches and one with Q_3 and Q_4 that has range searches. For the read-only and the update-only workloads we have two versions: one with uniform accesses and one with skewed accesses. Every workload has a small fraction (1%) of updates (Q_6) uniformly distributed across the whole domain to mimic updates and corrections frequently taking place in mixed analytical and transactional processing.

7.2 Casper Improves Overall Throughput

Figure 12 shows that Casper matches or significantly outperforms most of the different approaches frequently used for handling hybrid workloads. The figure compares the throughput of all tested approaches normalized against the column-store state-of-the-art design that employs a delta store. The first two workloads are hybrid with skewed accessed to more recent data. The main difference between the two is that the first has equality queries (Q_1) and the second has range queries (Q_3). Casper leads to a 1.75× to 2.14× performance improvement over the state of the art. With respect to the different configurations we tried, we observe that equi-width partitioning is slower than the state of the art (equality queries have to read whole partitions even when not needed), while it actually leads to a small performance improvement for


(a) hybrid (Q_1, Q_4, Q_6), skew


(b) read-only (Q_1, Q_2, Q_6), skew

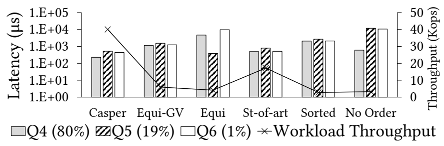

(c) update-only (Q_4, Q_5, Q_6), uniform

Figure 13: Casper offers significant performance benefits. In (a) for a hybrid workload with skewed point queries and inserts, Casper outperforms all approaches by balancing the read and update performance. In (b) for a read-heavy workload with range queries, point queries, and a few inserts, Casper matches the state-of-the-art delta-store design. Finally, in (c) for an update-only workload, Casper significantly outperforms by 2× or more all other approaches .

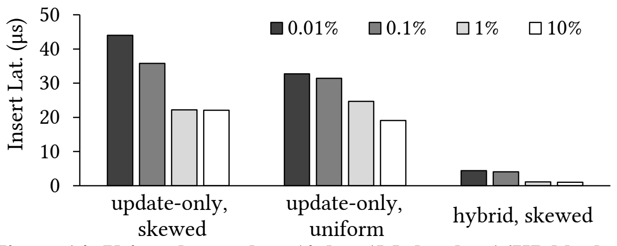


Figure 14: Using ghost values (4 thr., 1M chunks, 16KB blocks).

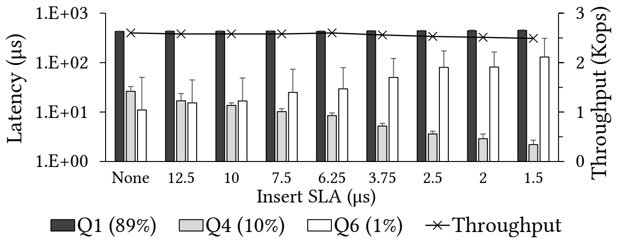


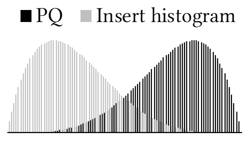
Figure 15: Casper meets performance SLA executing a hybrid workload (Q_1, Q_4, Q_6) (1M chunk size, 16KB block size).

range searches. As Casper targets mixed workloads, it cannot always offer the best performance for read-only workloads (Q_1 and Q_2). In particular, the state of the art has 5% higher throughput for skewed read-only queries. On the other hand, for uniform accesses, Casper leads to 1.44× higher throughput. For these cases, however, even simply equi-width partitioning and maintaining data sorted on the search attribute delivers competitive performance. Finally, for the update-intensive workloads (Q_4 and Q_5), Casper offers 2.28-2.32× higher throughput by exploiting ghost values and avoiding frequent data reorganization (which the state of the art does).

7.3 Casper's Impact on Update Performance

Casper Offers Better Read/Write Balance. Next, we drill-down to understand where the benefits of Casper come from, by measuring the latency of each operation for each workload in Figure 13. For the hybrid workload with equality queries and skewed accesses (Fig. 13a), Casper offers three orders of magnitude faster inserts (Q_4) than the other column layouts without hurting the latency of Q_1. For a read-only workload with both equality searches and range searches (Fig. 13b), even a small number of updates (<1% of the workload) disrupts the performance of all other column layouts except Casper. The reason is that Casper executes read queries without having as many partitions as the equi-width partitioned approaches, nor having the cost to maintain and continuously integrate a delta storage. For the update-only workload, the difference is more pronounced (Fig. 13c); Casper limits the number of partitions and distributes more effectively the ghost values.

Casper Leverages Ghost Values. We now demonstrate that Casper can leverage ghost values to optimize inserts. Figure 14 shows that insert latency scales as we increase the number of ghost values from 0.01% of the dataset to 10%, for update-intensive and hybrid work-


(a) Workload histograms over normalized domain.

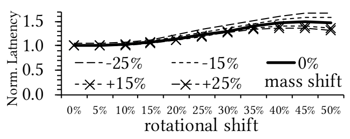

(b) Normalized latency with workload uncertainty.

Figure 16: Testing Casper's robustness.

loads. We observe that in all cases Casper takes advantage of the additional ghost values to further reduce the insert latency; using as low as 1% ghost values, Casper reduces insert latency by 2×.

7.4 Meeting Performance Constraints

Our next experiment showcases how Casper matches a given performance requirement without sacrificing overall system performance using a hybrid workload with equality searches (Q_1), inserts (Q_4), and a tiny fraction of updates (Q_6). Figure 15 shows the latency of each query, as well as the system throughput for different insert SLAs on the x-axis. When no SLA is set, Casper achieves the highest throughput. As we decrease the maximum allowed insert latency, the insert cost decreases proportionally (we report both the average and the 99.9% percentile with the error bars), with negligible impact on the overall system throughput. Interestingly, for lower insert SLA the update cost increases. The lower insert latency is achieved by having fewer partitions. This leads to more expensive point queries to locate the deleted value when executing the delete part of the update. Casper balances the overall throughput and respects the insert SLA with a small performance hit (< 3%).

7.5 Robustness to Workload Uncertainty

Our final experiment tests the robustness of the physical layout proposed and shows that Casper is robust up to a level of workload uncertainty, after which, there is a performance cliff. Figure 16a shows a summarization of the histograms used as input to the Frequency Model for point queries and inserts. Both operations occur with frequency 50%, however, point queries mostly target the latter part of the domain, and inserts the first part of the domain. Next, we consider two types of uncertainty in Figure 16b: (i) mass shift from point queries to inserts (shown as different lines), and (ii) rotational shift with respect to the targeted part of the domain (x-axis). Figure 16b shows that Casper offers a robust physical layout which absorbs uncertainty both in terms of mass shift up to 15% and in terms of rotational shift up to 10% without any significant penalty. As the uncertainty increases, however, we observe a performance penalty of up to 60%. We believe that a new optimization paradigm is warranted when the workload knowledge is given with such uncertainty. We further experiment with various workload profiles and observe that for some workloads the penalty increases earlier, hence, there is an opportunity for higher benefits by developing a new robust tuning paradigm. We leave as future work a new problem formulation using robust optimization techniques [21, 56].

7.6 Discussion

We summarize our observations about column layouts for hybrid workloads. (1) Existing column layouts have different performance profiles. For example, a delta-store efficiently supports update-heavy workloads, and equi-width partitioning supports read-intensive workloads, while Casper balances both. (2) Contrary to standard design for data analytical systems, sorting columns is not always optimal. (3) Equi-width partitioning following the effective access distribution provides a good layout when range queries dominate. (4) Ghost values significantly affect the quality of a column layout. Distributing ghost values equally without re-evaluating the impact on the fundamental operations does not always provide a good solution. Re-partitioning data and re-distributing ghost values adaptively are left as follow-up work. (5) Finally, ghost values bring a significant performance gain for updates with a small overhead for reads. While there is an update vs. read (and vs. space) tradeoff, overall, Casper offers performance savings at the expense of small additional memory consumption.

8. RELATED WORK

Offline Physical Design and Data Partitioning. Offline physical tools offer support for offline analysis for physical design (auto-tuning). They rely on what-if analysis, and are tightly integrated with the query optimizer of each system [4, 27, 34, 37, 3, 83, 59, 65, 82]. Partitioning a relation is NP-hard [72]. Data partitioning covers both the problem of partitioning a relation across multiple servers and within a single server [63, 79, 80]. Partitioning across both rows and columns is introduced by several systems to account for different read access patterns (e.g., on fact tables and dimension tables) [4, 11, 26]. The workload is frequently modeled as a graph [29, 69, 75]. Other approaches extract features from each workload operation based on its predicates [79, 80].

Casper offers a partitioning design tool that supports a general hybrid workload, and uses the access patterns of *both read and update operations* to form an optimization problem that has an exact solution. We keep the partitioning time low by introducing a knob between quality of the solution and partitioning time. Casper is the first approach to navigate a three-way tradeoff between read performance, update performance, and memory utilization, through the partitioning decision and the use of additional space (ghost values) for the update-heavy partitions.

Physical Design by Querying. The cost of offline physical design was addressed by a collection of approaches that use the workload as an advice to perform physical design "as-we-go", like online analysis [23, 24, 73, 52] and database cracking [40]. Online analysis uses cycles between workload execution and offline analysis, while database cracking immediately starts executing queries, and treats each as a hint to further partition data. The problem of updating in the context of database cracking has also been studied [41]. While some online tools evaluate index selection iteratively, cracking [40] and adaptive indexing [36] are gradually partitioning the data in an online, workload-aware manner.

Contrary to past work on cracking and adaptive indexing that uses read queries as hints for physical design, Casper uses *both read and update access patterns* as input to finding the optimal physical design. In cracking, updates are treated as adversarial workload, they are buffered and periodically merged with the main data. Further, adaptive indexing does not control the extent of additional space and, hence, cannot navigate the update vs. space tradeoff.

Learning Database Tuning. More recently, the complete automation of data system design decisions has re-emerged as a research goal [42, 43, 64], aiming to exploit the recent algorithmic advancements in deep learning, hardware, and adaptive database architectures. Recent work reposes the question of physical design [6, 31, 32, 42, 43, 59] and workload prediction [53] using mathematical tools (machine learning, optimization, numerical methods).

Casper is philosophically in the same path, and it focuses on the column physical organization problem for hybrid workloads. To the best of our knowledge, it is the first approach that uses horizontal partitioning, a hybrid update policy and ghost values as a means of balancing read and update cost in a workload-driven manner.

Column-Stores and Updates. Supporting updates in columnar systems has been the object of several research efforts. Several approaches use additional metadata to capture and temporarily store updates. For example, fractured mirrors use both a columnar and a row-wise data representation [70, 71]. Other approaches employ a variation of LSM-Trees [60] which implement differential updates [12, 13, 48, 67, 78]. Data Blocks [49] support a hybrid block layout and store chunks of data along with metadata that allows for efficient searching, querying, and updating. Another approach is to augment a column-store with an index that supports positional updates, a different way to implement *out-of-place* updates that reduces the cost of query processing when using a delta store [38].

Casper enables tuning along three dimensions, balancing read performance, update performance, and memory utilization. In addition, Casper supports constraints along those three metrics. Casper enables principled tuning depending on the exact HTAP scenario as opposed to working with a fixed balance. More interestingly Casper expands the design space as it can be combined with any of the above approaches to provide a hybrid behavior. For example, given that Casper controls the column layout only, any decision across columns can be done in an orthogonal way. That is, the column-oriented part of a fractured mirrors-like approach [7, 11] can use Casper for each column layout to reach the desired HTAP balance and to speed up merging new values from the write-store to the read-store. The same is true for PAX layouts [5]. In this way, Casper opens up new opportunities to think about the design space broadly across all the above options.

Ghost Values. The columnar design of modern systems takes advantage of the dense layout of columns to use a number of optimizations like vectorized processing, SIMD processing, and compression. While this offers superb read performance it makes in-place updates virtually impossible. Past research has studied the use of interspersing empty slots (ghost values) throughout the column to allow cheaper updates, deletes, and inserts [18, 19]. Our work extends the state of the art as it combines ghost values with range partitioning to create a column layout that balances read performance, update performance, and memory amplification.

9. CONCLUSIONS

We show that analytical systems relying on columnar storage can use tailored column layouts to support mixed workloads with both reads and writes (using horizontal partitioning, a hybrid update policy, and ghost values). Performance can be $2-4\times$ better than that of fixed state-of-the-art designs that rely on delta storage. We frame these column layout questions as an optimization problem that, given workload knowledge and performance requirements, provides an optimal physical layout for the workload at hand, while remaining robust to limited workload changes.

Acknowledgments. We thank the reviewers for their valuable feedback and Andy Huynh for his help in further quantifying the robustness of our approach. This work is supported by NSF Grants No. IIS-1850202 and No. IIS-1452595.

10. REFERENCES

[1] D. J. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[2] D. J. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 671–682, 2006.

[3] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1110–1121, 2004.

[4] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 359–370, 2004.

[5] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 169–180, 2001.

[6] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1009–1024, 2017.

[7] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A Hands-free Adaptive Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1103–1114, 2014.

[8] E. D. Andersen and K. D. Andersen. The Mosek Interior Point Optimizer for Linear Programming: An Implementation of the Homogeneous Algorithm. *High Performance Optimization*, 33:197–232, 2000.

[9] Apache. Parquet. *https://parquet.apache.org/*.

[10] R. Appuswamy, M. Karpathiotakis, D. Porobic, and A. Ailamaki. The Case For Heterogeneous HTAP. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017.

[11] J. Arulraj, A. Pavlo, and P. Menon. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2016.

[12] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. MaSM: Efficient Online Updates in Data Warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 865–876, 2011.

[13] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. Online Updates on Data Warehouses via Judicious Use of Solid-State Storage. *ACM Transactions on Database Systems (TODS)*, 40(1), 2015.

[14] M. Athanassoulis and S. Idreos. Design Tradeoffs of Data Access Methods. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tutorial*, 2016.

[15] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 461–466, 2016.

[16] M. Athanassoulis, Z. Yan, and S. Idreos. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2016.

[17] R. Barber, G. M. Lohman, V. Raman, R. Sidle, S. Lightstone, and B. Schiefer. In-Memory BLU Acceleration in IBM's DB2 and dashDB: Optimized for Modern Workloads and Hardware Architectures. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2015.

[18] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-Oblivious B-Trees. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.

[19] M. A. Bender and H. Hu. An Adaptive Packed-Memory Array. *ACM Transactions on Database Systems (TODS)*, 32(4), 2007.

[20] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[21] D. Bertsimas, O. Nohadani, and K. M. Teo. Robust Optimization for Unconstrained Simulation-Based Problems. *Operations Research*, 58(1):161–178, 2010.

[22] J. Bisschop. *AIMMS - Optimization Modeling*. AIMMS, 2006.

[23] N. Bruno and S. Chaudhuri. To Tune or not to Tune? A Lightweight Physical Design Alerter. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 499–510, 2006.

[24] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 826–835, 2007.

[25] B. Chandramouli, G. Prasaad, D. Kossmann, J. J. Levandoski, J. Hunter, and M. Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 275–290, 2018.

[26] C. Chasseur and J. M. Patel. Design and Evaluation of Storage Organizations for Read-Optimized Main Memory Databases. *PVLDB*, 6(13):1474–1485, 2013.

[27] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 146–155, 1997.

[28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, 2010.

[29] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *PVLDB*, 3(1):48–57, 2010.

[30] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The Snowflake Elastic Data Warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 215–226, 2016.

[31] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the*

ACM SIGMOD International Conference on Management of Data, pages 79–94, 2017.

[32] N. Dayan, M. Athanassoulis, and S. Idreos. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Transactions on Database Systems (TODS)*, 43(4):16:1–16:48, 2018.

[33] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Engineering Bulletin*, 35(1):28–33, 2012.

[34] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. Physical Database Design for Relational Databases. *ACM Transactions on Database Systems (TODS)*, 13(1):91–128, 1988.

[35] P. Francisco. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. *IBM Redbooks*, 2011.

[36] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 371–381, 2010.

[37] T. Härder. Selecting an Optimal Set of Secondary Indices. In *Proceedings of the European Cooperation in Informatics (ECI)*, pages 146–160, 1976.

[38] S. Héman, M. Zukowski, and N. J. Nes. Positional Update Handling in Column Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 543–554, 2010.

[39] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[40] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007.

[41] S. Idreos, M. L. Kersten, and S. Manegold. Updating a Cracked Database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–424, 2007.

[42] S. Idreos, K. Zoumpatianos, M. Athanassoulis, N. Dayan, B. Hentschel, M. S. Kester, D. Guo, L. M. Maas, W. Qin, A. Wasay, and Y. Sun. The Periodic Table of Data Structures. *IEEE Data Engineering Bulletin*, 41(3):64–75, 2018.

[43] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 535–550, 2018.

[44] T. Johnson and D. Shasha. Utilization of B-trees with Inserts, Deletes and Modifies. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 235–246, 1989.

[45] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 195–206, 2011.

[46] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 489–504, 2018.

[47] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T.-H. Lee, J. Loaiza, N. Macnaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zait. Oracle Database In-Memory: A Dual Format In-Memory Database. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2015.

[48] A. Lamb, M. Fuller, and R. Varadarajan. The Vertica Analytic Database: C-Store 7 Years Later. *PVLDB*, 5(12):1790–1801, 2012.

[49] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2016.

[50] P.-Å. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, and V. Papadimos. Real-Time Analytical Processing with SQL Server. *PVLDB*, 8(12):1740–1751, 2015.

[51] P.-A. Larson, R. Rusanu, M. Saubhasik, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, and S. Rangarajan. Enhancements to SQL server column stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1159–1168, 2013.

[52] M. Luhring, K.-U. Sattler, K. Schmidt, and E. Schallehn. Autonomous Management of Soft Indexes. In *Proceedings of the IEEE International Conference on Data Engineering Workshops (ICDEW)*, pages 450–458, 2007.

[53] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 631–645, 2018.

[54] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 37–50, 2017.

[55] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 476–487, 1998.

[56] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. CliffGuard: A Principled Framework for Finding Robust Database Designs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1167–1182, 2015.

[57] MySQL. Online Reference for Storage Engine API. *https://dev.mysql.com/doc/internals/en/custom-engine.html*, 2019.

[58] T. Neumann, T. Mühlbauer, and A. Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 677–689, 2015.

[59] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Slalom: Coasting Through Raw Data via Adaptive Partitioning and Indexing. *PVLDB*, 10(10):1106–1117, 2017.

[60] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*,

33(4):351–385, 1996.

[61] Oracle. Introducing Oracle Database 18c. *White Paper*, 2018.

[62] F. Özcan, Y. Tian, and P. Tözün. Hybrid Transactional/Analytical Processing: A Survey. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1771–1775, 2017.

[63] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: Page Latch-free Shared-everything OLTP. *PVLDB*, 4(10):610–621, 2011.

[64] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-Driving Database Management Systems. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017.

[65] A. Pavlo, C. Curino, and S. B. Zdonik. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–72, 2012.

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

[67] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquin, and D. Kossmann. Fast Scans on Key-Value Stores. *PVLDB*, 10(11):1526–1537, 2017.

[68] H. Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–2, 2009.

[69] A. Quamar, K. A. Kumar, and A. Deshpande. SWORD: Scalable Workload-Aware Data Placement for Transactional Workloads. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 430–441, 2013.

[70] R. Ramamurthy, D. J. DeWitt, and Q. Su. A Case for Fractured Mirrors. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 430–441, 2002.

[71] R. Ramamurthy, D. J. DeWitt, and Q. Su. A Case for Fractured Mirrors. *The VLDB Journal*, 12(2):89–101, 2003.

[72] D. Saccà and G. Wiederhold. Database Partitioning in a Cluster of Processors. *ACM Transactions on Database Systems (TODS)*, 10(1):29–56, 1985.

[73] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: Continuous On-Line Database Tuning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 793–795, 2006.

[74] D. Schwalb, M. Faust, J. Wust, M. Grund, and H. Plattner. Efficient Transaction Processing for Hyrise in Mixed Workload Environments. In *Proceedings of the International Workshop on In Memory Data Management and Analytics (IMDM)*, pages 16–29, 2014.

[75] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. *PVLDB*, 10(4):445–456, 2016.

[76] N. Shamgunov. The MemSQL In-Memory Database System. In *Proceedings of the International Workshop on In-Memory Data Management and Analytics (IMDM)*, 2014.

[77] V. Sikka, F. Färber, A. K. Goel, and W. Lehner. SAP HANA: The Evolution from a Modern Main-Memory Data Platform to an Enterprise Application Platform. *PVLDB*, 6(11):1184–1185, 2013.

[78] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column-oriented DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 553–564, 2005.

[79] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained Partitioning for Aggressive Data Skipping. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1115–1126, 2014.

[80] L. Sun, M. J. Franklin, J. Wang, and E. Wu. Skipping-oriented Partitioning for Columnar Layouts. *PVLDB*, 10(4):421–432, 2016.

[81] TPC. Specification of TPC-H benchmark. *http://www.tpc.org/tpch/*, 2018.

[82] E. Wu and S. Madden. Partitioning Techniques for Fine-grained Indexing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 1127–1138, 2011.

[83] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1087–1097, 2004.

[84] M. Zukowski and P. A. Boncz. Vectorwise: Beyond Column Stores. *IEEE Data Engineering Bulletin*, 35(1):21–27, 2012.

[85] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, page 59, 2006.