SageDB: An Instance-Optimized Data Analytics System

ABSTRACT

Modern data systems are typically both complex and general-purpose. They are complex because of the numerous internal knobs and parameters that users need to manually tune in order to achieve good performance; they are general-purpose because they are designed to handle diverse use cases, and therefore often do not achieve the best possible performance for any specific use case. A recent trend aims to tackle these pitfalls: *instance-optimized* systems are designed to automatically self-adjust in order to achieve the best performance for a specific use case, i.e., a dataset and query workload. Thus far, the research community has focused on creating instance-optimized database components, such as learned indexes and learned cardinality estimators, which are evaluated in isolation. However, to the best of our knowledge, there is no complete data system built with instance-optimization as a foundational design principle.

In this paper, we present a progress report on SageDB, our effort towards building the first instance-optimized data system. SageDB synthesizes various instance-optimization techniques to automatically specialize for a given use case, while simultaneously exposing a simple user interface that places minimal technical burden on the end user. We show that our prototype outperforms a commercial cloud-based analytics system by up to 8× on end-to-end query workloads and up to 4 orders of magnitude on individual queries. SageDB is an ongoing research effort, and we highlight our lessons learned and key directions for future work.

1 INTRODUCTION

Most modern data management systems fall on a spectrum between general-purpose and application-specific. For example, PostgreSQL [3] is extremely general purpose, and powers a diverse range of analytical and transactional workloads. Apache Spark is slightly specialized towards analytic tasks, but can still handle a wide variety of use cases (e.g., batch reporting, ad-hoc interactive queries, data science, and ML) and low-level workloads (e.g., I/O-bound, CPU-bound, in-memory, on-disk, in the cloud). On the other hand, systems like Google's Mesa [25] and Napa [5] were custom-built to power Google Ads, and are not suitable for any other application. While these systems improve efficiency, these bespoke systems require years of intense engineering effort and are only achievable by large corporations with significant resources.

Ideally, users should be able to have the efficiency of specialized systems along with the flexibility of general-purpose systems. Tuning configuration options ("knobs") is easier than building an entirely new system, and can bridge some of the performance gap. However, experienced engineers and database administrators still go through the time-consuming and error-prone tuning process for each application. Recent research proposes techniques for automatic knob tuning [10]; however, the performance impact of tuning such knobs is still limited. For example, users can only adjust the size of a data block, not how data is laid out on disk. Fundamentally, general-purpose systems are designed to be task agnostic, so for most tasks a tuned general-purpose system will perform worse than a custom-tailored system.

Recent work has shown that existing system components can be replaced with *instance-optimized* or *learned* components[1], which are able to automatically adjust to a specific use case and workload (see [1] for an overview). For example, learned index structures [17, 34] offer the same read functionality as traditional index structures (e.g. B+ trees) while providing better performance in both latency and space consumption. Instance-optimized data storage layouts [62] are able to improve scan performance by skipping data with greater effectiveness than traditional sorting-based partitioning techniques.

However, these instance-optimized components have largely been designed and evaluated in isolation, and there have only been a few efforts to integrate them into an end-to-end system. Bourbon [13] replaces block indexes in an LSM-tree with learned indexes and demonstrates latency improvements. Google integrated learned indexes into BigTable [4] with similar findings, mainly due to a smaller index footprint and fewer cache misses when traversing the index. While these are useful initial studies, it is still unclear how multiple instance-optimized components would work together in concert. In fact, it is easy to imagine a number of learned components destructively interfering with each other. Is it possible to build a system that autonomously custom-tailors its major components to the user's requirements, approaching the performance of a bespoke system but with similar ease of use as a general-purpose system?

To the best of our knowledge, there is no end-to-end data system built with instance-optimization as a foundational design principle. We previously presented our vision and blueprint for such a system, called SageDB [33]. In this paper, we present our first prototype of SageDB, and show how three carefully selected components can work together in practice. These instance-optimized components are (1) (multi-dimensional) data layouts and partial aggregation caching, (2) pre-materialization, and (3) join indexes and partial denormalization. These techniques minimize I/O when scanning data from disk and maximize computation reuse through intelligent pre-materialization of partial results and views. While the ultimate goal is to automatically trigger self-optimization whenever necessary, for the current prototype we decided to expose a single single easy-to-use command to the user — `OPTIMIZE` — with an optional space budget. Doing so gives the user control over when SageDB should start to instance-optimize the internal components to improve performance for the user's workload while respecting the space constraint.

Building a usable database takes years and several attempts (e.g., Oracle took until version 7 to become stable), so this paper should largely be regarded as a progress report on how to integrate learned components and the potential benefits they can provide when combined. As such, this paper aims to inform the research and industry communities about the potentials, limitations, and future research challenges of learned instance-optimization.

In summary, we make the following contributions:

[1] We acknowledge that the term "instance-optimized" or "learned" is overloaded. However, the name is in line with previous works [17, 21, 32–34, 36, 62].

(1) We integrate three instance-optimized techniques for query-time data pruning and computation reuse. Specifically, we design a policy that integrates them in a way that both improves performance on an individual query level and maximizes coverage on a workload level (i.e., maximizes the fraction of the workload that benefits).
(2) We implement an extremely simple and intuitive interface for the user to interact with SageDB's instance-optimization capabilities: the user only needs to decide when to issue the `OPTIMIZE` command, and SageDB will automatically decide how to simultaneously configure all internal instance-optimized components.
(3) We present an evaluation of our prototype implementation of SageDB against other systems, including a commercial cloud-based data warehouse product, which SageDB outperforms by up to 8× on end-to-end query workloads and up to 4 orders of magnitude on individual queries.

2 SAGEDB

In this section, we provide a brief overview of the state of research on instance-optimized systems. Then we describe our motivations and design principles for building SageDB.

2.1 Background

Instance-optimization (a term inspired by the definition of instance-optimal algorithms [54]) refers to specializing a system based on the dataset and workload to achieve performance close to specialized solutions [33]. While there exists many possible ways to create instance-optimized components, a common approach is to tightly couple a model of the user's workload (e.g., queries, data) with a novel data structure designed to take advantage of that model. Sometimes, this approach is also referred to as learned systems or algorithms with predictions/oracles [27]. For example, learned indexes [34] model the user's data to accelerate searches on that dataset. Instance-optimized data layout techniques [48, 62] create workload-specific physical designs that minimize I/O during query execution. Past work tended to improve performance for a single instance-optimized component in isolation, but not for the entire database. For example, learned indexes were evaluated on single-key lookup workloads instead of complete transactional workloads, and data layouts were evaluated on selective scan-heavy queries. Note that instance-optimized systems are fundamentally different from automatic knob-tuning approaches. Knob-tuning optimizes the hyperparameters of a system and is agnostic to the underlying data distribution. Instance-optimization designs systems that take advantage of knowledge about the specific data and/or workload distribution.

2.2 Motivations for SageDB

We had two motivations for building SageDB. First, we aim to show that instance-optimization can provide benefits for end-to-end workloads with diverse query patterns instead of just database components evaluated in isolation. Second, we hoped that building and evaluating SageDB on real data and workloads would identify the most important pain points and roadblocks and guide us towards the most impactful directions for future work in instance-optimized systems.

Like many existing learned components [34, 44, 62], we focus on analytic workloads as well. We leave investigation of instance-optimization for transactional workloads to future work.

2.3 Design Principles

We used several general principles to guide our design:
(1) **Maximize coverage.** We want to incorporate instance-optimized components that collectively provide benefits for as wide a range of query patterns as possible. Ideally, across a workload of diverse queries, nearly every query should benefit from SageDB.
(2) **Avoid regression.** One of the biggest deterrents to the adoption of instance-optimized techniques in practice is the fear that they might result in catastrophic failures or performance regressions under changing or even adversarial workloads. This fear of regression often outweighs the promise of potential performance improvements. In SageDB, we err on the side of caution: we must consider a component's downsides just as carefully as its upsides, and it must be simple to disable the component if necessary. The worst case should be no impact—not negative impact.
(3) **Simplicity first.** It is tempting to use SageDB to showcase the latest and greatest research ideas in instance optimization, but first we must ensure that SageDB operates in a way that is user-friendly, predictable, and interpretable. Therefore, we aim to maintain a simple overall architecture.
(4) **Minimize the burden on the user.** Configuring the components should be as automatic as possible and should require as little as possible from the user, both in terms of interaction and understanding. The complexity of incorporating new instance-optimized components into SageDB should be completely hidden from the user—they should not need to read more documentation or issue new commands in order to make use of those new components. Accordingly, SageDB is designed such that the user only needs to issue a single `OPTIMIZE` command to trigger all optimizations.
(5) **Avoid negative interference.** When combining a number of learned components, it is natural to worry that optimizing each component individually might not lead to an optimal global configuration. In the worst case, different learned components might "step on each other," degrading system performance. Optimizing a large number of components simultaneously is an interesting, but extremely difficult, problem. Instead, in this first prototype we carefully select the set of instance-optimized components and integrate them in a way that guarantees that they cooperate with each other even when optimized individually.

3 SYSTEM OVERVIEW

In this section, we provide a high-level overview of SageDB as a system, without discussing instance optimization. In Section 4, we introduce SageDB's instance-optimization techniques, and Section 5 covers the optimization procedure. Fig. 1 provides an overview.

3.1 Storage Layer

SageDB by default stores data in columnar format, although row-store format is also available. The records of a table are divided into *horizontal partitions*. Each partition is stored as a separate file;

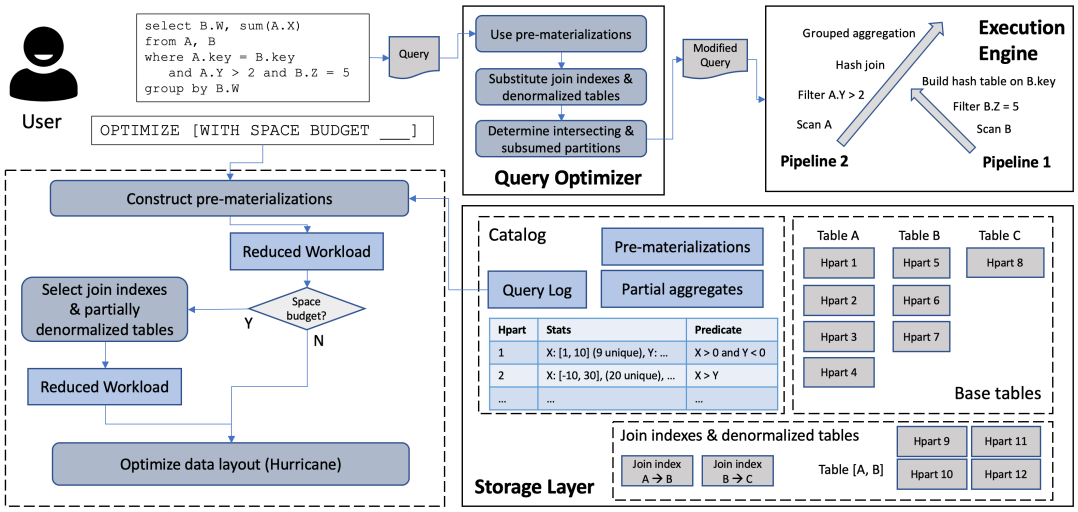

Figure 1: A user query passes through the rule-based optimizer, which determines if and how to incorporate SageDB's instance-optimized components, then runs on SageDB's vectorized execution engine. Users can issue an `OPTIMIZE` command at any time, which prompts SageDB to automatically configure instance-optimized components to maximize performance based on the user's query history.

Table 1: Instance-optimized components in SageDB.

Component	Purpose	Interactions
Hurricane (§4.1)	reduce I/O by optimizing data layout	impacts caching effectiveness
Partial aggregation caching (§4.2)	computation reuse for repeated aggregations	effectiveness depends on Hurricane
Pre-materialization (§4.3)	computation reuse for repeated query patterns	reduces filters relevant for Hurricane
Join indexes & partial denormalization (§4.4)	computation reuse for repeated joins	can also benefit from Hurricane

each column of each horizontal partition can be accessed individually, without reading other columns. String columns are dictionary encoded, and integer columns are compressed using bit-packing.

For each horizontal partition, we store statistics used for execution-time data skipping, including the minimum value, maximum value, and number of distinct values for each column. In addition, we optionally store a predicate for each partition, with the property that all records in the partition are guaranteed to satisfy the predicate (see Section 4.1 for details). When a query scans from a table, SageDB compares the query's filter with the per-column statistics and the optional predicates to determine the set of horizontal partitions that can be skipped, i.e., the partitions for which the statistics and predicate guarantee that no row can match the filter. SageDB uses memory-mapped file I/O for data files stored on local SSD or disk. For long-term persistence, data files are stored on AWS S3 or other cloud object stores.

However, what distinguishes SageDB from traditional systems is the amount of customization the system offers on how to partition data. SageDB is able to partition the data in a multi-dimensional way based on the workload and cleverly combine the partitioning with partial aggregates and pre-materialized results far beyond the capabilities of traditional systems (see Section 4).

Like traditional systems, SageDB has a catalog to store schema information (tables, columns, and data types) and information about the physical database, including the names of files (i.e., horizontal partitions) belonging to a table. SageDB also uses the catalog for storing additional metadata which are created as part of our instance-optimized techniques, such as partial aggregates and pre-materializations.

3.2 Execution Engine

SageDB has a vectorized execution engine that processes a chunk of data at a time. SageDB uses non-compiled pipelines with push-based execution. The first pipeline for each table involves scanning data from disk, for which the granularity of a chunk is a horizontal partition. Each pipeline may involve a projection over the columns or a filter over the rows. SageDB supports lazy materialization by maintaining a bitmap of relevant rows and passing the bitmap through the pipeline.

When evaluating filters, SageDB applies each filter sequentially, only on the rows that satisfy all previous filters. Therefore, the order of applying filters matters. During query optimization, SageDB estimates the selectivity of each filter and executes them in increasing order of selectivity, so that earlier filters eliminate the most rows.

SageDB supports hash join and a special indexed join that we will discuss in Section 4.4. SageDB implements aggregation push-down [61], so that aggregations which typically are performed after

a join can be pushed down directly to the base tables, which can significantly reduce the size of the input to the join operation.

SageDB uses multi-threaded parallel query execution. For pipelines where the sink is mergeable (e.g., aggregations on mergeable operators such as `COUNT` and `SUM`), SageDB will divide the source into equal parts (e.g., all relevant horizontal partitions in a table are divided among threads), which are projected and filtered in parallel, before merging the resulting sinks.

Fig. 1 shows an example of query execution in SageDB. The user's query first passes through a rule-based query optimizer, which determines which columns to read from each table, determines which horizontal partitions to scan from each table, orders tables for hash joins so that the largest table is the probe side, and constructs the execution pipelines. The query optimizer also determines how to take advantage of SageDB's instance-optimized components; we discuss this in depth in Section 5.2. The constructed pipelines are then executed in sequence. However, to date our execution engine itself does not make use of algorithms with predictions (e.g., learned joins or sorting [36]).

3.3 SQL Support

SageDB supports a command-line SQL interface as well as a Python connector library. Users can load data into tables from either CSV files or Parquet files. SageDB returns query results to the user in JSON format. SageDB currently supports a limited subset of SQL. All of SageDB's instance-optimized techniques aim to improve performance for `SELECT` queries, with an emphasis on aggregation queries, both with and without a `GROUP BY`. Supported aggregation functions include `COUNT`, `COUNT DISTINCT`, `COUNT APPROX DISTINCT` (using HyperLogLog [22]), `SUM`, `AVG`, `MIN`, and `MAX`. Queries can also can contain `ORDER BY`, `HAVING`, `LIMIT`, `DISTINCT`, and analytic functions. SageDB only supports inner equijoins. We are currently working to also support subqueries and CTEs. SageDB also supports `INSERT`, but it is not a focus of the current design.

4 INSTANCE-OPTIMIZED COMPONENTS

In this section, we describe the individual instance-optimized components in depth, which are summarized in Table 1. In Section 5, we describe how SageDB exposes this complexity to the user through the `OPTIMIZE` command.

4.1 Data Layout

As described in Section 3.1, SageDB uses per-partition column statistics and an optional predicate to skip horizontal partitions, and thereby reduce I/O, during query execution. The effectiveness of data skipping depends on the data layout. For example, if a table containing data about order shipments is sorted by the date of the shipment, but queries typically filter by the price of the order, we are unlikely to skip many horizontal partitions.

SageDB has a data layout optimization module called Hurricane, which is able to create complex data layouts and combines the ideas of qd-trees [62] and MTO [16], both techniques to create instance-optimized block layouts, with the ideas of Flood [48] and Tsunami [18], techniques to create instance-optimized multi-dimensional in-memory data layouts. Based on the observed workload, Hurricane detects what records are commonly accessed together and creates a partitioning layout which minimizes the number of required partitions per query. Moreover, Hurricane can further

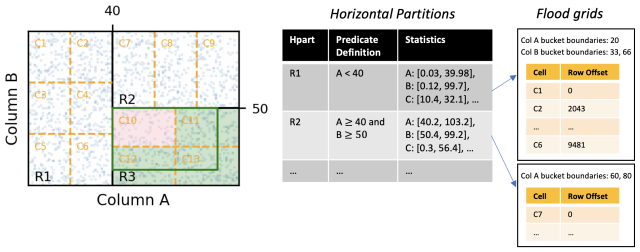

Figure 2: An example data layout with three horizontal partitions. When processing a query that filters for records lying inside the green-outlined rectangle, the execution engine skips partitions 1 and 2 and only scans the intersecting cells of partition 3 (shaded green). SageDB avoids scanning subsumed cells/partitions (shaded pink) whose partial aggregations exist in the cache.

sort the rows within each horizontal partition in such a way that minimizes the number of rows accessed. The reorganized layout replaces the original layout, so there is no long-term storage overhead, although there may be temporary storage overhead when creating but before switching to the new layout. We now describe the data layout construction algorithm in more depth.

4.1.1 Phase 1: Creating Horizontal Partitions. SageDB defines a target number of rows per horizontal partition, which by default is set to 2M rows based on the latencies we observed for Amazon S3[2]. Unlike many other systems, this target is an average, not a minimum. Therefore, a table with 200M rows will result in a data layout with 100 horizontal partitions, but some partitions may have much fewer than 2M rows. We allow for this flexibility because Hurricane will automatically determine if it's worth having a very small, frequently accessed partition.

The first phase of the construction algorithm will split the table's records into horizontal partitions. We begin with all the records in a single horizontal partition. The algorithm proceeds iteratively: in each iteration, we grow the number of partitions by one by splitting an existing partition into two child partitions. To split a partition, we evaluate a predicate on each record in the partition, assign records that evaluate to true to one child, and assign records evaluating to false to the other child. The predicate used to split an existing partition is chosen from the set of candidate predicates, which is the set of filter predicates that appear in the query log. When choosing the predicate for splitting a partition, we use the one amongst the candidate predicates that minimizes the cost (i.e., number of partitions that cannot be skipped) of the resulting data layout over the queries in the query log. Whenever a new partition is created, we compute and store the reduction in cost that would be achieved if the partition were split. At each iteration, we choose to split the partition that results in the largest cost reduction. We continue iterating until the average partition size has reached the target average region size. The conjunction of predicates that were used to construct a partition is saved as the optional partition predicate (Section 3.1).

[2]Future versions of SageDB will automatically tune this parameter based on the observed performance.

This phase is similar to prior work [16, 62], with the key difference that we have an average partition size constraint instead of a minimum size constraint, which means that we need to decide not only how best to divide a partition (which is a local decision), but also which existing partition to divide (which is a global decision). The benefit of having an average partition size instead of a minimum size is that it is a more permissive constraint and expands the design space: it is possible that a small but extremely hot area of data space merits its own partition. However, extremely small partition sizes are undesirable, e.g., due the filesystem block size and compression rates. Therefore, SageDB by default does define a minimum partition size of K records, a parameter which, like the target partition size, has to be tuned based on the hardware (in all our experiments it is set to 50K).

4.1.2 Phase 2: Sorting Within Horizontal Partitions. The second phase of the construction algorithm will sort the records within each horizontal partition using the Flood construction algorithm [48], which defines a grid-based multi-dimensional data layout based on the anticipated workload and data distribution. Structurally, Flood will create a grid over partition's data space (Fig. 2). To create a grid over n columns, Flood divides the domain of each column's values into equally-sized buckets, much like an equi-depth histogram: for each column $i \in [0,n)$, Flood creates b_i buckets by setting the boundary values between buckets in such a way that $1/b_i$ records fall in each bucket. Flood determines b_i automatically based on the data and workload; $b_i = 1$ is equivalent to not including column i in the grid. When combined, the buckets of each column form a n-dimensional grid with $\prod_{i \in [0,n)} b_i$ *cells*, which are ordered. The records within each cell are stored contiguously when sorting the partition.

During query processing, Flood uses the bucket boundary values in each column to efficiently determine which cells intersect with the query's filter (i.e., which cells might contain records matching the query's filter). Only intersecting cells are scanned, and all other cells are skipped. Therefore, even for partitions that are not skipped, the execution engine only needs to access a subset of the rows.

4.1.3 Data Layouts Are Not Enough. Optimized data layouts produced by Hurricane are useful for skipping irrelevant data, but it is less useful if the amount of relevant data is large, i.e., if a query has a non-selective filter or no filter at all. For example, if a query performs an aggregation over all shipments made before the current week, the only data that could be skipped are the records from the current week, which is likely only a small fraction of the overall data.

Therefore, this remainder of this section describes SageDB's other instance-optimized components, which are largely framed around the idea of computation reuse. In other words, if processing a piece of data is unavoidable, we aim to at least not process the same data in the same way more than once.

4.2 Partial Aggregation Caching

If an aggregation query's filter completely subsumes a horizontal partition, i.e., every record in the partition is guaranteed to match the query's filter based on the partition's statistics, then SageDB can cache the *partial aggregation*, i.e., the result of the aggregation when performed over only that partition's data. If a future query with the same aggregation has a filter that subsumes the same partition, then the cached partial result can be reused.

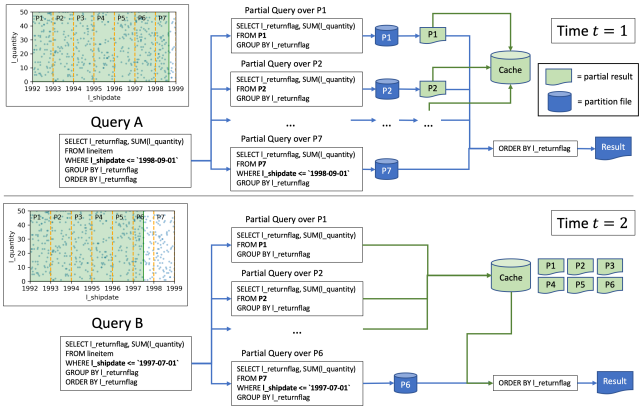

Figure 3: Partial aggregation caching on a partitioned dataset allows the engine to reuse computation over subsumed data partitions from query A (top) when executing query B (bottom).

This technique is different from query result caching because only the result of the aggregation is cached, not the result of the entire query. It is also different from most previously-explored forms of intermediate result caching because it caches at the granularity of a data partition, not of the entire table; this difference is important for workloads consisting of query templates (see Section 5.1), as we show in the following example.

Consider a simplified form of TPC-H Q1, which scans the `lineitem` table, filters the records using a predicate over the `l_shipdate` column, and then performs an aggregation over the remaining records:

```
SELECT l_returnflag, SUM(l_quantity) FROM lineitem
WHERE l_shipdate <= ? GROUP BY l_returnflag ORDER BY l_returnflag
```

This is a query template where a changeable placeholder, represented by ?, appears in the filter. A user may execute this template repeatedly with different values for the placeholder on each invocation. Therefore, computation reuse methods that cache the entire query result will provide little benefit because the query is rarely executed with the same placeholder value twice, and methods that cache intermediate results will provide little benefit because most intermediate results will also be different.

However, partial aggregation caching has a big impact. For example, assume that we partition the `lineitem` table on the `l_shipdate` column into seven partitions, labeled P1 through P7 (Fig. 3). Assume the user first executes the query template with the placeholder set to `1998-09-01`. This query can be divided into seven partial queries, one over each partition. Each partial query reads from its partition and then the partial results are merged (i.e., the rows are fed through an `ORDER BY l_returnflag` operator) to create the final query result. Notice that partitions 1 through 6 are *subsumed*[3] within the query's filter, i.e., every record in those partitions is guaranteed to match the filter. SageDB caches the partial aggregates computed over each of those subsumed partitions.

[3]In this paper, we use "subsumption" to describe when the data space occupied by a partition of data is completely contained within a query's filter, as opposed to the ability for a query to be computed based on the result of a different query.

Assume the user later executes the same query template with the placeholder set to `1998-07-01`. Again we divide the query into six partial queries (we ignore partition 7 because none of its data can match the query filter). The engine can now reuse the cached partial aggregate from any subsumed partitions, which in this case is partitions 1 through 5, and it only needs to scan the base data from P6. This example illustrates that if the query workload involves performing the same aggregations over the same subsumed partitions repeatedly, partial result caching can avoid redundant computation.

However, the effectiveness of partial aggregation caching depends on the data layout, i.e., how many partitions each query subsumes and whether different queries subsume the same partitions. For example, if we take the partitioning from Fig. 3 and execute the same aggregation but now filter by a different column (e.g., `WHERE l_extendedprice > ?`), then none of the partitions are subsumed, and partition aggregation caching is useless. Therefore, Hurricane's data layout construction algorithm is cache-aware: when creating horizontal partitions, subsumed partitions do not count towards the cost (Section 4.1.1). We also store partial aggregates at the Flood cell level and therefore similarly adjust the cost model for constructing Flood grids (Section 4.1.2).

While the idea of partial aggregation caching has been previously proposed [15], prior work did not more globally optimize the data layout in the presence of partial aggregation caching. However, if considered together, partial aggregation caching on the block level allows us to significantly increase performance at very little additional memory overhead. This demonstrates that it is important to design components in the context of the system they are used in.

4.3 Pre-materialization

Partial aggregation caching has the downside that its effectiveness depends on the data layout. If a partition is never subsumed by the query filter, then a partial aggregation cannot be built on it even if the aggregation is repeated many times. To allow more degrees of customization, SageDB also has the ability to fully or partially materialize results. While most database systems support fully materialized results, which usually have to be manually created by an administrator and require huge amounts of storage, we found that partial materialization is an extremely powerful concept but one that requires auto-tuning as it quickly becomes overwhelming to manually manage and is thus rarely found in traditional systems. Partial pre-materialization refers to the technique of pre-materializing arbitrary sub-results of a query. Note that a partial result might be a subset of the result of a single operator (e.g., a partial sum) and thus does not have to be a subquery. Hence, queries which use partially materialized results can skip some of the processing but might still require processing the "remainder query" by scanning the base data.

What makes SageDB unique is its ability to automatically decide what partial results to create based on the observed workload. In order to do this optimization, we analyze the query history, extract query patterns, and create pre-materialized results for individual query patterns or templates (Section 5.1). Here, we aim to construct each pre-materialization in only one pass over the data. For a given query template, we determine if a pre-materialization can be constructed as follows (see Table 2 for examples):

(1) If the template does not have a `GROUP BY` and all placeholders occur as part of equality predicates, we can compute results for the entire domain of placeholders using a `GROUP BY` query, which produces a materialized view. Any future instances of the query template can be fully answered directly from this materialized view.
(2) If the template includes at most one placeholder in an inequality predicate, and all other placeholders appear in equality predicates, we can compute results for the entire domain of placeholders using an analytic query. This also produces a materialized view which can be directly used to fully answer future queries.
(3) If the query's placeholders all occur within a single IN-list predicate, we can create a full pre-materialization by issuing the same query but additionally grouping by the predicated column. We then separate the resulting relation based on the predicated column. Therefore the pre-materialization in this case is not a single materialized view, but rather a mapping from each unique value of the predicated column to a materialized view.
(4) If the query's placeholders all appear in predicates over the same column (either equality or inequality), create a partial pre-materialization by issuing the same query but additionally grouping by a `CASE` expression that evaluates to different ranges over the predicated column. As with the previous case, we then separate the result based on the output of the case expression. When executing a query, we can directly use any partial results over subsumed ranges, execute the "remainder query" over the base data, and merge the results. For example, if pre-materialized results exist for the ranges $[0, 10)$ and $[10, 20)$ over the predicated column, and the query filters for $[0, 15)$, we can directly use the partial result over the first range and modify the query to only access base data over $[10, 15)$.

If any condition matches, then we immediately construct the pre-materialization accordingly and do not consider further conditions. To restrict space usage, we only construct a full pre-materialization resulting in a single materialized view (i.e., conditions 1 and 2) if the full domain of placeholders is less than 100K; we only construct condition 3 if the domain is less than 1000; and we only construct partial pre-materializations over at most 100 ranges, whose boundaries are randomly sampled over the data distribution of the templated column.

Even though the examples given in Table 2 are single-table queries, these techniques also apply to queries over multiple tables. Although these techniques can extend to more complex combinations of templated filters, e.g., a partial pre-materialization over a template with multiple predicated columns using inequalities, we limit ourselves to the above rules in order to limit space overhead and reduce the chance for regression.

4.4 Computation Reuse for Joins

To support computation reuse for repeated join patterns, SageDB automatically creates join indexes and partial denormalization. A join index is between two tables, which we call a probe table and a target table. For each row of the probe table, the join index stores the row indexes for matching rows in the target table, assuming an equijoin over specified join keys. Then an indexed join will proceed similarly to a hash join, except that constructing the hash table is unnecessary because the join index already points each probe side row to matching rows on the target side. This is different from traditional foreign key indexes which only support one-to-many mappings but not the other way around. It is a form of "lazy" denormalization.

Table 2: Examples for creating pre-materializations, given a query template whose variable placeholders match certain conditions.

Type	Conditions on "?"	Example query template	Query for constructing pre-materialization
Full	No group-by, equality predicates only	SELECT COUNT(*) FROM table WHERE p = ? AND q = ? AND r > 10	SELECT p, q, COUNT(*) FROM table WHERE r > 10 GROUP BY p, q ORDER BY p, q
Full	Equality predicates and at most one inequality	SELECT p, COUNT(*) FROM table WHERE q >= ? AND r = ? AND s >= 1 GROUP BY p ORDER BY p	SELECT r, q, p, COUNT(*) OVER (PARTITION BY p, r ORDER BY q) FROM table WHERE s >= 1 ORDER BY r, q
Full	Single equality column	SELECT q, COUNT(*) FROM table WHERE p IN (?, ?, ?, ?) GROUP BY q	SELECT p, q, COUNT(*) FROM table GROUP BY p, q
Partial	Single inequality column	SELECT q, COUNT(*) FROM table WHERE p BETWEEN ? AND ? AND r > 10 group by q	SELECT [case expression], q, COUNT(*) FROM table WHERE r > 10 GROUP BY [case expression], q

Partial denormalization takes this idea one step further and directly materializes selected columns from the join result of multiple tables. The denormalized table is constructed using the result of a `SELECT` query. Denormalized tables can replace their corresponding base tables during query execution (Section 5.2).

5 THE `OPTIMIZE` COMMAND

Given the various instance-optimized components described in the previous section, one natural design question is how to expose this functionality to the user. In SageDB, we aim to make the user interface as simple and as automated as possible. Therefore, we expose a single command, `OPTIMIZE`, which the user can issue through the SQL interface or Python connector. The command has a single optional argument, which is a budget for the amount of space on disk that SageDB is allowed to use to store extra data and metadata.

The user's only responsibility is to decide when to issue the `OPTIMIZE` command. We envision that the user runs the command during a time of low system load, so that the optimization process does not affect performance of concurrently running queries; this is the same advice that data warehouse providers typically give to users when suggesting knob tuning recommendations. Ideally, the user should have already issued a representative set of queries on SageDB, because the optimization will require examining and modeling the user's query history. For example, if the user uses SageDB to run a daily batch reporting job, then they may want to run the first day's batch, then issue the `OPTIMIZE` command overnight, so that the next day's batch can take advantage of performance improvements.

5.1 Optimization Algorithm

When the user triggers the `OPTIMIZE` command with an optional space budget, SageDB needs to configure all of its instance-optimized components simultaneously. The algorithm is as follows (Fig. 1):

(1) The catalog stores a log of all past user queries. We first examine that history and cluster queries into *templates*. A template is a query for which constant literals in the query filter are replaced by placeholders. Some example query templates are shown in Table 2. Within each template, if a certain placeholder always has the same constant value, we remove the placeholder and simply use the value. We expect that many real workloads (e.g., daily batch reporting jobs, dashboard queries) have repeated query patterns and are naturally composed of templates. Furthermore, SageDB allows users to explicitly define prepared statements, which are treated as templates.
(2) For all templates found in the previous step, attempt to create a full or partial denormalization, according to the rules specified in Section 4.3. Even if the user does not specify a space budget, this step is performed, because the domain size limits for pre-materializations automatically limit their space consumption to a small fraction of the overall data size (we confirm this empirically in Table 4).
(3) With the remaining space budget, automatically create the highest-impact join indexes and partially denormalized tables, using an algorithm we will describe in Section 5.1.1. Importantly, we do this in such a way that maximizes performance not over the full query history, but rather over a reduced workload consisting of "remainder queries." That is, if a query can be fully answered from pre-materializations, then it is not included in the reduced workload. If a query is partially answered, then we include a modified version of the query that only includes the portion of the filter that needs to access the base data.
(4) Use Hurricane to optimize the layout on both base tables and on any partially denormalized tables constructed in the last step. Each table is optimized only for the queries that filter on that table. For each multi-table query, we push down all single-table predicates to their respective tables and create a single-table query for each table. If a query can make use of one of the denormalized tables (see Section 5.2), then the query will be used to optimize the layout for the denormalized table, not the base tables.

Note that in all cases, the execution engine will skip partitions and use the partial aggregation cache based on the existing data layout, even before the first `OPTIMIZE` call is issued.

5.1.1 Space Budget Allocation. If the user gives a space budget, SageDB is responsible for deciding which join indexes and partially denormalized tables to create under that space constraint. It prioritizes selecting techniques that maximize *efficiency*, defined as the ratio of performance improvement to space overhead. This is similar to a well-known bin-packing heuristic; however, the optimizer's task differs from the standard bin-packing problem, since adding some optimizations may render others unnecessary. For example, adding a denormalized table over a set of columns S_c in tables A and B may preclude the need for a join index over A and B, or for another denormalized table over a subset of S_c.

Computing efficiency. SageDB's optimizer computes the efficiency of a particular optimization by estimating the required space overhead and improvement in query latency. The space overhead is determined as:

$$S = |J| \cdot \sum_{C \in S_c} V(C)$$

where $|J|$ is the number of rows in the joined table J, and $V(C)$ is the average size of a value in column C. If the join predicate involves

a column with unique values in one table, as is often the case with foreign keys, $|J|$ is simply the number of rows in the other table, since we work only with inner joins. Otherwise, we compute $|J|$ as $m_v \cdot n_v$ where m_v,n_v are the number of times the join key v appears in each table.

The optimizer estimates the relative performance of a hash join before and after the optimization as:

$$P(J) = \alpha_1|A| + \alpha_2|B| + \alpha_3|J||S_c|$$

The three terms are the costs of building hash table, probing the table, and materializing the joined rows, respectively, with constants α_i that weight the terms based on their empirically determined runtimes (we omit details of this measurement for brevity). For denormalized tables, $P(J) = 0$, and for join indexes, $\alpha_1 = \alpha_2 = 0$. The performance improvement of a particular optimization is the difference between the estimated $P(J)$ with and without the optimization.

Selecting Optimizations. The candidate set of optimizations is derived from queries in the reduced workload (described above) and comprises:

(1) A join index between any two joined tables in a query in the reduced workload.
(2) A denormalized table between any number of tables and any subset of columns from those tables.

We note that adding a set of columns to a denormalized table is worthwhile only if doing so makes the denormalized table usable on a query that couldn't use it before, i.e., if the denormalized table now contains all the columns relevant to that query. Therefore, we limit the candidate denormalized tables to those that are the union of columns from some subset of the queries.

The optimizer ranks each optimization by its efficiency, and repeats the following process, until we either exhaust the space budget or no candidates remain:

(1) Choose the candidate with the highest efficiency.
(2) Eliminate candidates that are subsumed by this candidate.
(3) Re-compute efficiencies for the remaining candidates.
(4) Remove candidates whose space overhead, in addition to the chosen candidates, would exceed our space budget.

In particular, in step (3), the space overhead and performance improvement of each candidate must be recomputed *relative* to the current set of chosen optimizations. Our implementation uses optimizations that avoid re-computing efficiencies for candidates that are unaffected.

5.2 Post-`OPTIMIZE` Query Flow

After SageDB configures its instance-optimized components, those components are applied to subsequent queries as part of query optimization. A query passes through the rule-based optimizer with the following steps (Fig. 1):

(1) If the query matches a query template for which a pre-materialization exists, use it. If this completely answers the query (i.e., if it is a full pre-materialization), then the following steps are unnecessary. Otherwise, pass the "remainder query" to the next step.
(2) Substitute denormalized tables for base tables in an iterative greedy fashion. We substitute the denormalized table that replaces the most base tables first. If multiple denormalized tables replace the same number of base tables, we use the denormalized table over the largest base table. Note that we can only substitute a denormalized table if it contains all columns relevant to the query that appear in the respective base tables.
(3) If join indexes exist for any remaining joins, always use an indexed join instead of a hash join.
(4) Use the per-partition statistics and optional predicate definition to skip horizontal partitions and determine the ranges to read within each remaining horizontal partition, using the Flood grid. Furthermore, if any partitions or Flood cells are subsumed, directly read their partial results from the cache. If the partial result is not in the cache, compute the partial result and save it to the cache.
(5) Send the modified query to the execution engine, merge the result with any partial results from partial pre-materialization or the partial result cache, and return the final result to the user.

Table 3: Dataset and workload characteristics.

	Gaming	Stack Overflow	TPC-H
num tables	5	1	8
num rows in largest table	3.06B	507M	600M
uncompressed size (GB)	426	52	100
num templates	13	13	9

6 EVALUATION

In this section, we present the results of an experimental study that compares SageDB with other data analytics systems on both real and synthetic datasets and workloads. Overall, this evaluation shows:

(1) SageDB outperforms a commercial cloud-based analytics system by up to 8× on end-to-end query workloads and up to almost 10000× on individual query templates (Section 6.2).
(2) SageDB's instance-optimized components benefit different types of queries to different degrees, but almost all queries benefit from at least one instance-optimized component (Section 6.3).
(3) SageDB's optimizations rarely result in regressions for individual queries, and the `OPTIMIZE` command can easily be completed as a nightly job (Section 6.4).

6.1 Setup

We run SageDB on a EC2 machine with 4 vCPUs and 32GB RAM (`i3en.xlarge`). We compare against a popular cloud data warehousing product, which we call System X, running on a single node with the same number of cores and memory. We also run Umbra [49], a high-performance on-disk analytics system, on the same EC2 node.

We evaluate using three datasets and workloads (Table 3). Here, we summarize each dataset and workload at a high level. We include a full dataset schema and workload specifications in our extended report [2].

(1) **Gaming** is a real-world dataset from the gaming division of a major technology company, donated to us under the condition of anonymity. There are two fact tables, with roughly 2B and 3B rows respectively, and three smaller dimension tables. We use a real workload provided by the company.
(2) **Stack Overflow** is a single-table dataset with 500M records, each of which represents a post on Stack Overflow.
(3) **TPC-H** is a standard analytics benchmark. We use scale factor 100 to generate the data. Since SageDB currently cannot support

Table 4: Additional space usage for the tuned/optimized versions of System X and SageDB, as a percent of the uncompressed size of the original data.

	Gaming	Stack Overflow	TPC-H
System X tuned	21.1%	0.0005%	22.2%
SageDB opt.	0.0002%	0.0003%	0.00008%
SageDB opt. w/ extra space	31.7%	0.0003%	21%

more complex query structures like subqueries, we created our own simplified version of the TPC-H workload (see [2]).

All systems (SageDB, System X, and Umbra) will by default load the tables of each dataset with a default sort order. For Gaming, we sort all tables by primary key, except for the largest fact table, for which we sort by the foreign key to the other fact table (`attrib_f_id`). For Stack Overflow, we sort the table by the time of post. For TPC-H, we use the standard sort keys: `l_shipdate` for `lineitem`, `o_orderdate` for `orders`, and primary key for other tables. System X also supports distribution keys, which we set to be the same as the sort key.

When loading, SageDB automatically compresses columns as described in Section 3.1. System X also automatically selects the best encoding and compression method for each column.

6.2 Overall Results

We first compare SageDB directly against System X and Umbra on the three datasets and workloads. We show three different configurations for SageDB: (1) unoptimized, the out-of-the-box version of SageDB before the user has issued the `OPTIMIZE` command, (2) optimized, the state of SageDB after the user has issued the `OPTIMIZE` command with no additional space budget, and (3) optimized with extra space, the state of SageDB after the user has issued the `OPTIMIZE` command with some additional space budget (for the exact amount, see Table 4).

We show two different versions of System X: (1) the out-of-the-box configuration, after the data has been loaded. This uses sort keys and automatic column encoding. (2) A hand-tuned version, which represents a system whose knobs have been manually tuned by a human expert to fit the dataset and workload. The tuned version includes hand-picked materialized views. For Stack Overflow, the tuned version also sorts the table using an interleaved sort key (i.e., Z-order) over the `post_date` and `score` columns, which improves performance because `score` is correlated with many of the commonly filtered columns.

Fig. 4 shows that across the three workloads, SageDB outperforms the other systems on average query runtime by up to 8× and outperforms the unoptimized version of itself by up to 18×. Additionally, even the configuration of SageDB that does not use significant extra space outperforms the other systems across all three workloads. Umbra was unable to complete all queries in the workload for Gaming, which is why we do not include it in the plot.

For each workload, Fig. 4 also shows a per-template breakdown of speedups achieved by SageDB compared to System X tuned. For individual query templates, median speedups range from a slight slowdown (Gaming Q12) to almost 10000× (Stack Overflow Q1). In general, templates without speedups are ones for which the query filter is too complex (e.g., Gaming Q4, see Section 6.3), or ones for which the tuned System X has materialized views that achieve the same purpose as SageDB's instance-optimized components (e.g., Gaming Q12) and System X's raw execution engine is simply more efficient than SageDB's.

Table 4 shows the amount of extra space used by different configurations of SageDB and System X. Note that even when the user does not provide a space budget when issuing the `OPTIMIZE` command, SageDB still uses some extra space for storing pre-materializations and partial aggregates. However, this space is minimal compared to the overall size of the data.

While the performance numbers of SageDB are promising compared to System X and Umbra, it has to be pointed out that SageDB is still a prototype and is not yet feature-complete like System X (e.g., we do not support subqueries). Rather, there are two takeaways: first, SageDB as an out-of-the-box system, ignoring instance-optimized components, has roughly comparable performance to System X and Umbra when evaluated on the same hardware in a single-node setting. Second, and arguably more importantly, optimization allows SageDB to outperform the out-of-the-box version of itself by up to 18×. Next, we dive deeper in which how each instance-optimized component contributes to that performance gain.

6.3 Ablation Study

How much do each of SageDB's individual instance-optimized components contribute to the overall performance? In Table 5, we break down the effect of each instance-optimized components on each template of each workload. Overall, there are several takeaways.

First, different components help more for different types of queries. For example, the Hurricane data layout and partial aggregation caching are especially helpful for queries that either filter on a single table (e.g., Stack Overflow queries) or have inexpensive joins (e.g., Gaming Q1-3). Pre-materializations are helpful whenever they are applicable, and especially if it is a full pre-materialization (e.g., Gaming Q8 and Stack Overflow Q1). Join-based optimizations, i.e., join indexes and partially denormalized tables, are most helpful for queries that have expensive joins over large tables (e.g., TPC-H Q2 and Q4).

Second, SageDB's performance when all components are combined is sometimes better than any individual component on its own. For example, Stack Overflow Q4 and Q11 benefit from some synergy between partial pre-materializations, which answer most of the query, and then using the data layout and PAC to speed up the remainder of the query. Similarly, Gaming Q7 benefits from a partial pre-materialization, and then subsequently using a partially denormalized table to speed up the rest of the query.

Third, almost every query benefits from at least one instance-optimized component. This reflects our design goal for maximizing the *coverage* of instance-optimization techniques over diverse workloads. However, there are some types of queries for which none of our current techniques help. For example, Gaming Q4 does not qualify for pre-materialization according to SageDB's rules, does not have an expensive join, and its filter predicates are too complex to benefit from data layouts and partial aggregation caching.

Fourth, occasionally a single instance-optimized component is able to outperform SageDB. For example, on TPC-H Q2, the data layout alone outperforms SageDB. This is because SageDB's query flow (Section 5.2) is to always use pre-materializations if they exist, but in this particular case, the query itself ran relatively quickly, data layout was already very performant, and the extra optimizer overhead from

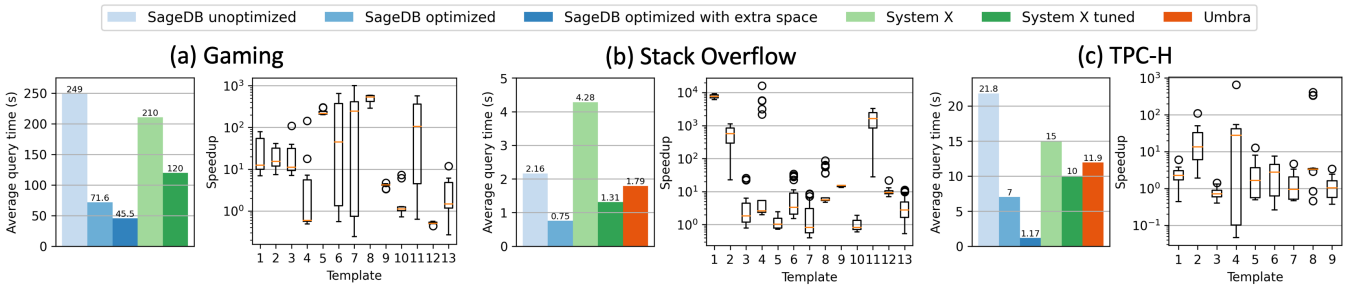

Figure 4: For each dataset, we show average query time on each system for the end-to-end workload, as well as a per-template breakdown of speedups achieved by SageDB compared to System X tuned. SageDB outperforms other systems by up to 8× on end-to-end query workloads and achieves up to almost 10000× speedup for individual templates.

Table 5: Ratio of average query time on the unoptimized SageDB vs. when the specified components is enabled. Higher is better. Highlighted is the component that makes the most impact on each template.

Gaming	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13
Hurricane + PAC	2.17	2.4	2.44	1.02	1.07	0.879	0.828	0.956	0.784	0.992	0.724	0.886	0.98
Pre-mat	0.957	0.989	1.0	1.02	109	3.6	8.74	2140	0.964	0.99	6.74	0.975	0.964
Joins	1.04	1.04	1.06	1.02	1.07	172	208	1.04	1.52	1.25	0.838	1.22	3.4
All	2.44	2.12	2.3	1.02	103	167	313	2100	2.69	3.2	3.16	2.05	3.02
Stack Overflow	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13
Hurricane + PAC	2.06	1.53	1.5	2.73	2.38	5.85	2.23	6.5	1.91	1.88	1.56	3.14	3.08
Pre-mat	11900	1.91	1.1	1.72	1.02	1.49	1.0	1.26	1.02	1.0	1.89	1.32	1.03
All	12700	2.89	1.44	9.01	2.29	5.92	2.2	6.34	1.9	1.84	3.49	3.28	3.16
TPC-H	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9				
Hurricane + PAC	20.1	1.03	5.37	1.02	9.07	1.07	8.04	1.01	1.11				
Pre-mat	25.4	0.99	3.63	1.55	11.7	6.57	11.0	1.07	2.17				
Joins	0.997	24.7	1.0	13.5	0.969	1.05	1.04	2.92	1.01				
All	25.9	28.3	3.67	41.5	11.5	6.72	11.4	2.92	2.12				

considering pre-materializations ate into the performance gains. This points to a direction for future work, which is to automatically determine, for each query, whether a certain instance-optimized component should be disabled.

6.4 Microbenchmarks

6.4.1 Regressions. SageDB improves overall performance, but we also want to ensure that individual queries do not regress. Table 5 showed that on a query template level, performance does not regress. Fig. 5 takes this a step further by breaking down individual query performance for each template, comparing the speedup in query runtime between the optimized and unoptimized configurations of SageDB. In general, regressions are rare, and when regressions do occur, they are minor compared to performance gains. Often, regressions are due to extra query optimization overheads for very short-running queries.

6.4.2 Space Budget. If users issue `OPTIMIZE` with a space budget, SageDB automatically selects the most impactful join indexes and partially denormalized tables within that budget. Fig. 6(A) shows that on the Gaming workload, as users increase the space budget, performance improves. There are sharp jumps in performance because those are the boundaries past which a useful index or denormalized table fits within the space budget.

6.4.3 Hurricane. We now study the impact of our data layout optimizer (Hurricane) and its interaction with partial aggregation caching (PAC). Using the TPC-H `lineitem` table, we generate a workload with conjunctive range predicates on shipdate (`l_shipdate >= ?`; 17 instances), extendedprice (`l_extendedprice >= ?`; 10 instances), or both (170 instances). The shipdate literals are hand picked and are first at year boundaries and then at month boundaries for the final year. The extendedprice literals are uniformly chosen from the column domain. All queries have the same simple aggregation (`sum(l_quantity * l_extendedprice)`). The queries range from being very selective (0.01% selectivity) to being not selective at all (100% selectivity). We train Hurricane on all 197 query instances and compare against a baseline (Sort) that sorts the data by both shipdate and extendedprice. We measure both data layouts with and without PAC. Figure 6(B) shows the query runtimes achieved with the four different storage configurations. For very selective queries there is only a small difference between Hurricane and Sort. The gap between Hurricane and Sort increases with more qualifying rows (0.1% to 1% selectivity). With 100% selectivity (right-most point in the figure), Hurricane is slightly worse than Sort. The reason is that for Hurricane we additionally check whether the per-partition predicates overlap with the query predicate. As one would expect, both Hurricane and Sort have less impact with increasing selectivity

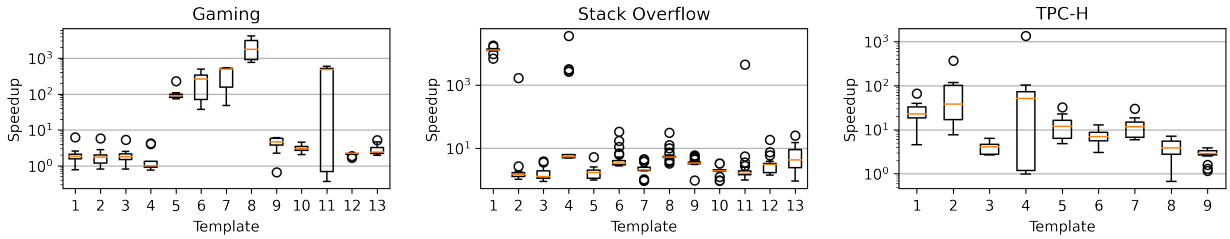

Figure 5: Per-query speedups for SageDB compared to its unoptimized configuration. Regressions are rare. The orange line represents the median; the box represents first and third quartiles; whiskers extend from the box by 1.5× the inter-quartile range; dots are those past the end of the whiskers.

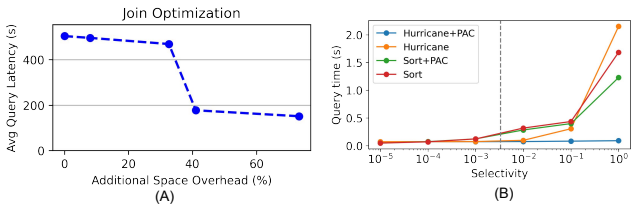

Figure 6: (A) Performance improves as users increase the space budget, as a percentage of the original database size, when issuing the `OPTIMIZE` command. (B) Query runtimes for SageDB with different storage configurations on queries with conjunctive range predicates on TPC-H lineitem. The vertical dashed line denotes the point at which the query result size equals the average size of a horizontal partition (i.e., 2M rows).

Table 6: Optimization Time (in seconds).

	Gaming	Stack Overflow	TPC-H
Pre-materializations	5950	690	923
Joins	1280	N/A	363
Hurricane layout	5240	5500	434
Total	12500	6190	1720

(more qualifying rows). To also improve such queries SageDB employs partial aggregation caching (PAC). As shown in the figure, PAC improves the query runtimes of both Hurricane and Sort. The reason why PAC has a much larger impact on Hurricane than on Sort is that Hurricane optimizes for the number of subsumed partitions. For each subsumed partition, we can reuse cached aggregates created by PAC. For Sort there are fewer subsumed partitions, i.e., queries are not aligned with partition boundaries.

6.4.4 Optimization Time. We expect that users should trigger the `OPTIMIZE` command during a time of low system load, similar to what popular data warehouse products advise their customers to do when following recommended optimizations. Therefore, optimization should not interfere with normal workload execution.

Table 6 breaks down the time that SageDB spends on each step of optimization for each dataset. Overall, optimization finishes in less then 3.5 hours for the largest dataset, which reasonably fits into periods of low system load (e.g., overnight). Even if the optimization is performed while queries are running, this quickly pays off in terms of saved query time. For example, on the Gaming workload, since the benefit from optimization is around 200 seconds per query on average (Fig. 4), it only takes just over 60 queries executed to recoup the time "lost" to optimization. We have not extensively optimized the code for reorganizing the data layout yet (e.g., it is single-threaded), and we believe that with further engineering, data layout optimization can be completed much faster.

7 LESSONS LEARNED AND FUTURE WORK

In this section, we take a step back and consider how the current SageDB design compares to our original design principles (Section 2). We also highlight important directions for future work.

Maximize coverage. Our evaluation shows that SageDB provides benefits for nearly all queries across three diverse query workloads. However, we did identify one query pattern for which none of our components made an impact: queries with complex multi-clause filters that do not perform expensive joins, exemplified by Gaming Q4 (Section 6.3).

Avoid regression. SageDB's instance-optimized components are designed to limit per-query regressions by default. Section 6.4 shows that we avoid regressions on all templates, but individual components do sometimes regress performance (Table 5). To limit these cases further, SageDB can use a learned query optimizer, like Bao [44], to decide whether to disable each instance-optimized component before executing each query.

Currently, all instance-optimized components except for Hurricane's data layout can be turned off for a particular query. For example, the SageDB optimizer could decide to simply not use a pre-materialization. However, since Hurricane changes the data layout of the original table, there is no way to fall back to the original layout. In the future, we plan to change Hurricane so that instead of reorganizing the original data, we only reorganize a copy of the relevant columns of the base table. Therefore, the execution engine can fall back to the original data if desired. One implication of this approach would be that Hurricane layouts contribute to the overall space usage, so we need to incorporate Hurricane into the space budget allocation algorithm (Section 5.1.1). Note that for the purposes of avoiding regressions, creating a Hurricane layout on a denormalized

table does not require creating a copy, since we can always decide not to use the denormalized table.

Simplicity first and avoid negative interference. We did not need to completely revamp the architecture of an analytic database system in order to achieve our performance gains. Instead, we carefully selected how to non-intrusively integrate our instance-optimized components into the overall system—in particular, each query incorporates instance-optimized components into its query flow as part of a simple set of steps through our rule-based optimizer (Section 5.2). As a result, the core storage layer and execution engine, which is a fundamental part of any analytic data system, remained mostly unchanged.

Minimize the burden on the user. SageDB places minimal technical burden on the user: their only responsibility is to issue an `OPTIMIZE` command, with an optional space budget, during times of low system load. However, our longer-term vision is to remove all responsibility altogether by automatically deciding when to perform optimization and which components to re-optimize.

Inserts. Finally, allowing SageDB and other instance-optimized systems to adapt to inserts is a key area of future work. The main challenge behind inserts is that they may invalidate optimizations constructed based on a static snapshot of the data. Different instance-optimization techniques are more or less robust to data changes. In this discussion, we focus on append-like insert patterns.

Instance-optimized techniques that operate on a single table can avoid invalidation through delta buffering. For example, new data is inserted into a special horizontal partition. The existing horizontal partitions remain unchanged, and existing partial aggregations cached for existing partitions remain valid. When scanning, SageDB can still take advantage of data skipping over existing partitions, but may need to always read the new partition. At a later time, when the user again triggers the `OPTIMIZE` command, the buffered data in the new partitions are incorporated into the new data layout.

Techniques which are not maintained at partition granularity, or which span multiple tables, are more difficult to maintain. These include pre-materializations over multi-table query templates, join indexes, and partially denormalized tables. However, as long as data is only changing in one base table (e.g., users append data to a fact table but the dimension tables are stable), these techniques can be maintained using the same delta buffering process described above. Addressing these challenges without performance degradation or increased complexity for the user is a key direction of future work.

8 RELATED WORK

Automatic database tuning. Modern data system have an increasing number of knobs and configuration options to be tuned by database administrators or by (semi-)automatic tools. There have been efforts to automatically tune a DBMSs' configuration since the early 2000s. Much of the previous work on automatic database tuning has focused on optimizing the physical design of the database [9], such as selecting indexes [6, 26], partitioning schemes [7, 11, 52], or materialized views [6]. Based on the method used to find the ideal configuration, the previous work can be divided into two categories: rule-based methods [12, 37] and ML-based methods [19, 41, 42, 51, 60, 63]. Cosine [8] focuses on self-designing key-value stores. Both approach performance optimization differently, with SageDB using learned components while Cosine essentially creates more knobs to tune. NoisePage [50] focuses on designing a self-optimizing database like SageDB by defining an objective function and action space. A centralized service learns to optimize the objective through the actions. NoisePage learns how to take standard actions in the database, such as adding/dropping indices, configuring knobs, and scaling hardware resources. Compared to instance-optimized components or systems, automatic database tuning has fewer degrees of freedom and is typically performed in a black-box manner.

Instance-optimized components. Further research has expanded the breadth and depth of instance-optimized components. More sophisticated learned indexes use multivariate data distributions to create multidimensional indexes [18, 48, 62]. There are now instance-optimized versions of bloom filters [14, 46, 59] and hash tables [56]. New use cases, from caching [30, 40] to query optimization [35, 44, 45] to scheduling [43], have leveraged learning to improve performance. SageDB aims to take this to the next step: where prior work designed components to adapt to the data and workload, SageDB intends to design an entire system with that capability.

Denormalization. Denormalizing multiple tables into a single monolithic table was explored in [38, 58], which determined that denormalized tables can offer substantial speedups in a column store, when combined with compression methods to reduce the extra space required. Unlike SageDB, this work assumed that the user would manually specify which tables had to be denormalized.

Partial pre-materialization. Computation reuse techniques that require knowing the future queries ahead of time include multi-query optimization [55, 57] (which aims to find a globally optimal execution plan for a batch of queries), materialized views [31], data cubes [23], and sub-expression materialization [29].

There have been various works on *opportunistically* caching intermediate results that are already materialized during query execution, and reusing these intermediate results during future query execution [28, 47]. Specific opportunistic caching systems target MapReduce [20] and large multi-node datacenters [24]. Hawc's query optimizer identifies query plans that produce intermediate results that can be cached and used for future queries [53]. Other work has applied deep learning to the task of opportunistic view materialization [39]. The above techniques assume that queries must be fully processed using pre-computed or cached results and are therefore not co-optimized with the data layout.

9 CONCLUSION

In this paper, we presented a progress report on SageDB, a first complete instance-optimized data system, focused on analytics. SageDB incorporates numerous instance-optimized components into one system that exposes a simple interface to the end user. While our prototype system already achieves impressive results, our aspirations for SageDB are far from complete. Our roadmap for future work includes implementing techniques to eliminate performance regressions, gracefully handling data changes, and incorporating further instance-optimized components. We hope that this report leads us a step closer towards making the vision for instance-optimized systems a reality.

REFERENCES

[1] [n.d.]. ML for Systems Papers. http://dsg.csail.mit.edu/mlforsystems/papers/

[2] [n.d.]. SageDB Extended Report. ([n. d.]). https://jialinding.github.io/sagedb.pdf

[3] [n.d.]. PostgreSQL Database, http://www.postgresql.org/. ([n. d.]).

[4] Hussam Abu-Libdeh, Deniz Altinbüken, Alex Beutel, Ed H. Chi, Lyric Doshi, Tim Kraska, Xiaozhou Li, Andy Ly, and Christopher Olston. 2020. Learned Indexes for a Google-scale Disk-based Database. *CoRR* abs/2012.12501 (2020). arXiv:2012.12501 https://arxiv.org/abs/2012.12501

[5] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Jim Chen, Min Chen, Ming Dai, Thanh Do, Haoyu Gao, Haoyan Geng, Raman Grover, Bo Huang, Yanlai Huang, Adam Li, Jianyi Liang, Tao Lin, Li Liu, Yao Liu, Xi Mao, Maya Meng, Prashant Mishra, Jay Patel, Rajesh Sr, Vijayshankar Raman, Sourashis Roy, Mayank Singh Shishodia, Tianhang Sun, Justin Tang, Jun Tatemura, Sagar Trehan, Ramkumar Vadali, Prasanna Venkatasubramanian, Joey Zhang, Kefei Zhang, Yupu Zhang, Zeleng Zhuang, Goetz Graefe, Divy Agrawal, Jeffrey F. Naughton, Sujata Kosalge, and Hakan Hacigümüs. 2021. Napa: Powering Scalable Data Warehousing with Robust Query Performance at Google. *Proc. VLDB Endow.* 14, 12 (2021), 2986–2998. http://www.vldb.org/pvldb/vol14/p2986-sankaranarayanan.pdf

[6] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. 2000. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, Vol. 2000. 496–505.

[7] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 359–370.

[8] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: a cloud-cost optimized self-designing key-value storage engine. *Proceedings of the VLDB Endowment* 15, 1 (2021), 112–126.

[9] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases*. 3–14.

[10] Surajit Chaudhuri and Gerhard Weikum. 2018. Self-Management Technology in Databases. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer. https://doi.org/10.1007/978-1-4614-8265-9_334

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

[12] Benoît Dageville and Mohamed Zait. 2002. SQL memory management in Oracle9i. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 962–973.

[13] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 155–171. https://www.usenix.org/conference/osdi20/presentation/dai

[14] Zhenwei Dai and Anshumali Shrivastava. 2019. Adaptive learned Bloom filter (Ada-BF): Efficient utilization of the classifier. *arXiv preprint arXiv:1910.09131* (2019).

[15] Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. 1998. Caching Multidimensional Queries Using Chunks. *SIGMOD Rec.* 27, 2 (June 1998), 259–270. https://doi.org/10.1145/276305.276328

[16] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. *Instance-Optimized Data Layouts for Cloud Analytics Workloads*. Association for Computing Machinery, New York, NY, USA, 418–431. https://doi.org/10.1145/3448016.3457270

[17] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 969–984. https://doi.org/10.1145/3318464.3389711

[18] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *CoRR* abs/2006.13282 (2020). arXiv:2006.13282 https://arxiv.org/abs/2006.13282

[19] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.

[20] Iman Elghandour and Ashraf Aboulnaga. 2012. ReStore: Reusing Results of MapReduce Jobs. *Proc. VLDB Endow.* 5, 6 (Feb. 2012), 586–597. https://doi.org/10.14778/2168651.2168659

[21] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB* 13, 8 (2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[22] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms (DMTCS Proceedings)*, Philippe Jacquet (Ed.), Vol. DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07). Discrete Mathematics and Theoretical Computer Science, Juan les Pins, France, 137–156. https://hal.inria.fr/hal-00406166

[23] J. Gray, A. Bosworth, A. Lyaman, and H. Pirahesh. 1996. Data cube: a relational aggregation operator generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS. In *Proceedings of the Twelfth International Conference on Data Engineering*. 152–159. https://doi.org/10.1109/ICDE.1996.492099

[24] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/nectar-automatic-management-data-and-computation-datacenters

[25] Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Govind Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal, Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones, Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, and Divyakant Agrawal. 2014. Mesa: Geo-Replicated, near Real-Time, Scalable Data Warehousing. *Proc. VLDB Endow.* 7, 12 (aug 2014), 1259–1270. https://doi.org/10.14778/2732977.2732999

[26] Michael Hammer and Arvola Chan. 1976. Index selection in a self-adaptive data base management system. In *Proceedings of the 1976 ACM SIGMOD International Conference on Management of data*. 1–8.

[27] Piotr Indyk, Yaron Singer, Ali Vakilian, and Sergei Vassilvitskii. [n.d.]. STOC'20 Workshop on Algorithms with Predictions. https://www.mit.edu/~vakilian/stoc-workshop.html

[28] Milena G. Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo A.P. Gonçalves. 2009. An Architecture for Recycling Intermediates in a Column-Store. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 309–320. https://doi.org/10.1145/1559845.1559879

[29] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. *Proc. VLDB Endow.* 11, 7 (March 2018), 800–812. https://doi.org/10.14778/3192965.3192971

[30] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K. Sitaraman. 2020. RL-Cache: Learning-Based Cache Admission for Content Delivery. *IEEE Journal on Selected Areas in Communications* 38, 10 (2020), 2372–2385. https://doi.org/10.1109/JSAC.2020.3000415

[31] Yannis Kotidis and Nick Roussopoulos. 1999. DynaMat: A Dynamic View Management System for Data Warehouses. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (Philadelphia, Pennsylvania, USA) *(SIGMOD '99)*. Association for Computing Machinery, New York, NY, USA, 371–382. https://doi.org/10.1145/304182.304215

[32] Tim Kraska. 2021. Towards instance-optimized data systems. *Proc. VLDB Endow.* 14, 12 (2021), 3222–3232. http://www.vldb.org/pvldb/vol14/p3222-kraska.pdf

[33] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *CIDR*.

[34] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909

[35] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR* abs/1808.03196 (2018). arXiv:1808.03196 http://arxiv.org/abs/1808.03196

[36] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. 2020. The Case for a Learned Sorting Algorithm. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1001–1016. https://doi.org/10.1145/3318464.3389752

[37] Eva Kwan, Sam Lightstone, Adam Storm, and Leanne Wu. 2002. Automatic configuration for IBM DB2 universal database. *Proc. of IBM Perf Technical Report* (2002).

[38] Yinan Li and Jignesh M Patel. 2014. Widetable: An accelerator for analytical data processing. *Proceedings of the VLDB Endowment* 7, 10 (2014), 907–918.

[39] Xi Liang, Aaron J. Elmore, and Sanjay Krishnan. 2019. Opportunistic View Materialization with Deep Reinforcement Learning. arXiv:1903.01363 [cs.DB]

[40] Thodoris Lykouris and Sergei Vassilvitskii. 2021. Competitive Caching with Machine Learned Advice. *J. ACM* 68, 4, Article 24 (jul 2021), 25 pages. https://doi.org/10.1145/3447579

[41] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 631–645. https://doi.org/10.1145/3183713.3196908

[42] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. 2021. MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems. In *Proceedings of*

the 2021 International Conference on Management of Data (SIGMOD/PODS '21). 1248–1261. https://doi.org/10.1145/3448016.3457276

[43] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (Atlanta, GA, USA) *(HotNets '16)*. Association for Computing Machinery, New York, NY, USA, 50–56. https://doi.org/10.1145/3005745.3005750

[44] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. *Bao: Making Learned Query Optimization Practical.* Association for Computing Machinery, New York, NY, USA, 1275–1288. https://doi.org/10.1145/3448016.3452838

[45] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *CoRR* abs/1904.03711 (2019). arXiv:1904.03711 http://arxiv.org/abs/1904.03711

[46] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Optimizing by Sandwiching. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/0f49c89d1e7298bb9930789c8ed59d48-Paper.pdf

[47] Fabian Nagel, Peter Boncz, and Stratis D. Viglas. 2013. Recycling in pipelined query evaluation. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 338–349. https://doi.org/10.1109/ICDE.2013.6544837

[48] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 985–1000. https://doi.org/10.1145/3318464.3380579

[49] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*.

[50] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. 2019. External vs. Internal: An Essay on Machine Learning Agents for Autonomous Database Management Systems. *IEEE Data Engineering Bulletin* (June 2019), 32–46. https://db.cs.cmu.edu/papers/2019/pavlo-icde-bulletin2019.pdf

[51] Andrew Pavlo, Matthew Butrovich, Lin Ma, Wan Shen Lim, Prashanth Menon, Dana Van Aken, and William Zhang. 2021. Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation. *Proc. VLDB Endow.* 14, 12 (2021), 3211–3221. https://db.cs.cmu.edu/papers/2021/p3211-pavlo.pdf

[52] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 61–72.

[53] Luis L. Perez and Christopher M. Jermaine. 2014. History-aware query optimization with materialized intermediate views. In *2014 IEEE 30th International Conference on Data Engineering*. 520–531. https://doi.org/10.1109/ICDE.2014.6816678

[54] Moni Naor Ronald Fagin, Amnon Lotem. 2003. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences* 66, 4 (2003), 614–656.

[55] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Dallas, Texas, USA) *(SIGMOD '00)*. Association for Computing Machinery, New York, NY, USA, 249–260. https://doi.org/10.1145/342009.335419

[56] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, and Tim Kraska. 2021. When Are Learned Models Better Than Hash Functions? *CoRR* abs/2107.01464 (2021). arXiv:2107.01464 https://arxiv.org/abs/2107.01464

[57] Timos K. Sellis. 1988. Multiple-Query Optimization. *ACM Trans. Database Syst.* 13, 1 (March 1988), 23–52. https://doi.org/10.1145/42201.42203

[58] Seung Kyoon Shin and G Lawrence Sanders. 2006. Denormalization strategies for data retrieval from data warehouses. *Decision Support Systems* 42, 1 (2006), 267–282.

[59] Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. 2020. Partitioned Learned Bloom Filter. *CoRR* abs/2006.03176 (2020). arXiv:2006.03176 https://arxiv.org/abs/2006.03176

[60] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.

[61] Weipeng P. Yan and Per-Åke Larson. 1995. Eager Aggregation and Lazy Aggregation. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 345–357.

[62] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning Data Layouts for Big Data Analytics. *CoRR* abs/2004.10898 (2020). arXiv:2004.10898 https://arxiv.org/abs/2004.10898

[63] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2020. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering* (2020).

A DATA SCHEMAS AND WORKLOADS

Here, we define the schemas for the three datasets by displaying their `CREATE TABLE` commands. We also define the three workloads by displaying the prepared statements. All of these use SageDB's SQL dialect, which is similar to but not entirely the same as any commercial SQL dialect.

A.1 Gaming

A.1.1 Schema.

```
create table dim1 (
   d1_label text,
   d1_id int64 UNIQUE
);
create table dim2 (
   d2_type text,
   d2_duration int64,
   d2_label text,
   d2_d1_id int64,
   d2_id int64 UNIQUE
);
create table dim3 (
   d3_label text,
   d3_joined int64,
   d3_loc text,
   d3_p1 float64,
   d3_p2 float64,
   d3_p3 float64,
   d3_p4 float64,
   d3_p5 float64,
   d3_id int64 UNIQUE
);
create table fact (
   f_time INT64,
   f_d3_id INT64,
   f_d1_id INT64,
   f_amt INT64,
   f_type TEXT,
   f_p1 INT64,
   f_p2 INT64,
   f_p3 INT64,
   f_p4 INT64,
   f_p5 INT64,
   f_p6 INT64,
   f_p7 INT64,
   f_p8 float64,
   f_p9 float64,
   f_p10 float64,
   f_p11 float64,
   f_p12 float64,
   f_id int64 UNIQUE
);
create table attrib (
   attrib_f_id int64,
   attrib_d2_id int64,
   attrib_share float64
);
```

A.1.2 Workload. Q1: `SELECT d1_label, COUNT(*) as cnt FROM fact, dim1 WHERE d1_id = f_d1_id AND f_p1 < ?:INT64 AND f_p8 < ?:FLOAT64 GROUP BY d1_label ORDER BY cnt;`

Q2: `SELECT d1_label, COUNT(*) as cnt FROM fact, dim1 WHERE d1_id = f_d1_id AND f_p2 < ?:INT64 AND f_p9 < ?:FLOAT64 GROUP BY d1_label ORDER BY cnt;`

Q3: `SELECT d1_label, COUNT(*) as cnt FROM fact, dim1 WHERE d1_id = f_d1_id AND f_p3 < ?:INT64 AND f_p10 < ?:FLOAT64 GROUP BY d1_label ORDER BY cnt;`

Q4: `SELECT d1_label, COUNT(*) as cnt FROM fact, dim1 WHERE d1_id = f_d1_id AND (f_p4 < ?:INT64 OR f_p5 < ?:INT64) AND (f_p6 < ?:INT64 OR f_p7 < ?:INT64) AND (f_p11 < ?:FLOAT64 OR f_p12 < ?:FLOAT64) GROUP BY d1_label ORDER BY cnt;`

Q5: `select d3_loc, sum(f_amt) as total from fact, dim3 where d3_id = f_d3_id and f_type=?:TEXT group by d3_loc order by total desc limit 20;`

Q6: `Select d2_label, sum(attrib_share * f_amt) as total from attrib, fact, dim2 where d2_id = attrib_d2_id and f_id = attrib_f_id and f_amt > ?:INT64 group by d2_label order by total desc;`

Q7: `Select d2_type, sum(attrib_share * f_amt) as total from fact, attrib, dim2 where d2_id = attrib_d2_id and f_id = attrib_f_id and f_amt > ?:INT64 group by d2_type order by total desc;`

Q8: `Select d2_label, d2_type, sum(attrib_share * f_amt) as total from fact, attrib, dim2, dim3 where d2_id = attrib_d2_id and f_id = attrib_f_id and f_d3_id = d3_id and d3_loc IN (?:TEXT, ?:TEXT, ?:TEXT, ?:TEXT, ?:TEXT, ?:TEXT) group by d2_label, d2_type order by total desc;`

Q9: `Select d3_loc, sum(f_amt) as total from fact, dim3 where f_d3_id = d3_id and (d3_p1 > ?:FLOAT64 or d3_p2 > ?:FLOAT64) group by d3_loc order by total desc;`

Q10: `Select d3_loc, sum(f_amt) as total from fact, dim3 where f_d3_id = d3_id and (d3_p3 > ?:FLOAT64 or d3_p4 > ?:FLOAT64) and d3_p5 > ?:FLOAT64 group by d3_loc order by total desc;`

Q11: `Select d2_type, sum(attrib_share * f_amt) as total from fact, attrib, dim2 where d2_id = attrib_d2_id and f_id = attrib_f_id and f_amt > ?:INT64 and attrib_share > 0.10 and f_p4 - 5500 > f_p7 group by d2_type order by total desc;`

Q12: `Select d3_loc, sum(f_p9) from fact, dim3 where f_d3_id = d3_id and (f_p2 = ?:INT64 or f_p4 = ?:INT64) group by d3_loc order by d3_loc;`

Q13: `Select d3_loc, sum(f_p9) from fact, dim3 where f_d3_id = d3_id and f_p2 > ?:INT64 and f_p2 < ?:INT64 and f_p4 > ?:INT64 and f_p4 < ?:INT64 group by d3_loc order by d3_loc;`

A.2 Stack Overflow

A.2.1 Schema.

```
create table stack_overflow (
  id UINT64,
  site_name TEXT,
  post_date FLOAT64,
  poster_name TEXT,
  poster_reputation INT32,
  poster_join_date FLOAT64,
  score INT32,
  view_count UINT64,
  favorite_count UINT64,
  answered UINT8,
  highest_score_answer INT32,
  comment_count UINT32,
  comment_max_score INT32,
  tag_count UINT32,
  tag_top25 UINT8,
  tag_top20 UINT8,
  tag_top15 UINT8,
  tag_top10 UINT8,
  tag_top5 UINT8,
  tag_rust UINT8,
  tag_cpp UINT8,
  tag_gpu UINT8,
  post_year INT32
)
```

A.2.2 Workload. Q1: `SELECT post_year, COUNT(*) FROM denorm_so WHERE answered = 1 AND comment_count <= ?:UINT32 GROUP BY post_year ORDER BY post_year;`

Q2: `SELECT post_year, COUNT(*) FROM denorm_so WHERE answered = 1 AND score >= ?:INT32 GROUP BY post_year ORDER BY post_year;`

Q3: `SELECT post_year, COUNT(*) FROM denorm_so WHERE highest_score_answer >= score AND view_count >= ?:UINT64 AND comment_max_score >= ?:INT32 AND answered = 1 AND comment_count >= 0 GROUP BY post_year ORDER BY post_year;`

Q4: `SELECT post_year, COUNT(*) FROM denorm_so WHERE answered = 0 AND comment_max_score >= ?:INT32 GROUP BY post_year ORDER BY post_year;`

Q5: `SELECT post_year, COUNT(*) FROM denorm_so WHERE view_count >= ?:UINT64 AND comment_count >= ?:UINT32 GROUP BY post_year;`

Q6: `SELECT poster_name, COUNT(*) FROM denorm_so WHERE tag_rust = 1 AND poster_join_date <= ?:FLOAT64 AND view_count >= ?:UINT64 GROUP BY poster_name;`

Q7: `SELECT post_year, COUNT(*) FROM denorm_so WHERE favorite_count <= ?:UINT64 AND post_date >= ?:FLOAT64 GROUP BY post_year ORDER BY post_year;`

Q8: `SELECT COUNT(*) FROM denorm_so WHERE poster_reputation >= ?:INT32 AND score >= ?:INT32 AND tag_top5 = 1;`

Q9: `SELECT post_year, COUNT(*) FROM denorm_so WHERE score >= ?:INT32 AND favorite_count >= ?:UINT64 GROUP BY post_year ORDER BY post_year;`

Q10: `SELECT post_year, COUNT(*) FROM denorm_so WHERE score <= ?:INT32 AND comment_count <= ?:UINT32 GROUP BY post_year;`

Q11: `SELECT post_year, COUNT(*) FROM denorm_so WHERE answered = 0 AND score >= ?:INT32 AND tag_count >= 4 GROUP BY post_year ORDER BY post_year;`

Q12: `SELECT COUNT(*) FROM denorm_so WHERE answered = 1 AND post_date >= ?:FLOAT64;`

Q13: `SELECT COUNT(*) FROM denorm_so WHERE view_count >= ?:UINT64 AND (tag_rust = ?:UINT8 OR tag_cpp = ?:UINT8 OR tag_gpu = ?:UINT8);`

A.3 TPC-H

A.3.1 Schema. We use the same TPC-H schema in the official specfication, except that we convert all datetime columns into integers, by remove the dashes from the datetime strings (e.g., "1995-01-01" becomes "19950101"). This is because SageDB does not yet have robust support for operations on datetime types.

A.3.2 Workload. Q1: `select l_returnflag, l_linestatus, count(*) from lineitem where l_shipdate <= ?:UINT64 group by l_returnflag, l_linestatus order by l_returnflag, l_linestatus;`

Q2: `select o_shippriority, sum(l_extendedprice) from lineitem, orders where l_orderkey = o_orderkey and o_orderdate < ?:UINT64 and l_shipdate > ?:UINT64 group by o_shippriority order by o_shippriority;`

Q3: `select o_shippriority, count(*) from orders where o_orderdate >= ?:UINT64 and o_orderdate < ?:UINT64 group by o_shippriority order by o_shippriority;`

Q4: `select n_name, sum(l_extendedprice) from lineitem, orders, customer, supplier, nation where c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey = n_nationkey and s_nationkey = n_nationkey and o_orderdate >= ?:UINT64 and o_orderdate < ?:UINT64 group by n_name;`

Q5: `select l_returnflag, sum(l_extendedprice * l_discount) from lineitem where l_shipdate >= ?:UINT64 and l_shipdate < ?:UINT64 group by l_returnflag order by l_returnflag;`

Q6: `select l_shipmode, count(*) from lineitem where l_commitdate < l_receiptdate and l_shipdate < l_commitdate and l_receiptdate >= ?:UINT64 and l_receiptdate < ?:UINT64 group by l_shipmode order by l_shipmode;`

Q7: `select l_returnflag, count(*) from lineitem where l_shipdate >= ?:UINT64 and l_shipdate < ?:UINT64 group by l_returnflag order by l_returnflag;`

Q8: `select sum(l_extendedprice) from lineitem, part where l_partkey = p_partkey and p_brand = ?:TEXT and l_quantity < ?:FLOAT64;`

Q9: `select p_mfgr, count(*) from part where p_size >= ?:UINT32 and p_size <= ?:UINT32 group by p_mfgr order by p_mfgr;`