SageDB: An Instance-Optimized Data Analytics System

Jialin Ding, Ryan Marcus*, Andreas Kipf,
Vikram Nathan, Aniruddha Nrusimha, Kapil Vaidya, Alexander van Renen†, Tim Kraska
Massachusetts Institute of Technology *University of Pennsylvania †Friedrich-Alexander-Universität Erlangen-Nürnberg

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 user. Our prototype outperforms a commercial cloud-based analytics system by up to 3× on end-to-end query workloads and up to 250× on individual queries. SageDB is an ongoing research effort, and we highlight our lessons learned and key directions for future work.

PVLDB Reference Format:
Jialin Ding, Ryan Marcus, Andreas Kipf, Vikram Nathan, Aniruddha Nrusimha, Kapil Vaidya, Alexander van Renen, Tim Kraska. SageDB: An Instance-Optimized Data Analytics System. PVLDB, 15(13): 4062 - 4078, 2022.
doi:10.14778/3565838.3565857

1 INTRODUCTION

Most modern data management systems fall on a spectrum between general-purpose and application-specific. For example, PostgreSQL [8] 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 [30] and Napa [10] 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 [15]; 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, which are able to automatically adjust to a specific use case and workload (see [5] for an overview). For example, learned index structures [24, 37] 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 [63] 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 [19] replaces block indexes in an LSM-tree with learned indexes and demonstrates latency improvements. Google integrated learned indexes into BigTable [9] 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 [36]. In this paper, we present our first prototype of SageDB, and show how two carefully selected components can work together in practice. These instance-optimized components are (1) (multi-dimensional) data layouts and data replication and (2) *partial* materialized views. These techniques minimize I/O when scanning data from disk and maximize computation reuse through intelligent pre-materialization of partial results. While the ultimate goal is to automatically trigger self-optimization whenever necessary, for the

Proceedings of the VLDB Endowment, Vol. 15, No. 13 ISSN 2150-8097.
doi:10.14778/3565838.3565857

current prototype we decided to expose a single easy-to-use command to the user — `OPTIMIZE` — with a user-given 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 introduce two new instance-optimized techniques: partial materialized views (PMVs), which is a generalization of traditional materialized views with more degrees of freedom, and replicated data layouts, which combines the idea of instance-optimized data layouts with partial table replication.
(2) We introduce a global optimization algorithm that jointly and automatically configures partial materialized views and replicated data layouts given the user's data and workload. As a result, a user only needs to decide when to issue the `OPTIMIZE` command, and SageDB will automatically decide how to simultaneously configure all 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 3× on end-to-end query workloads and up to 250× 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.

Background. Instance-optimization (a term inspired by the definition of instance-optimal algorithms [55]) refers to specializing a system based on the dataset and workload to achieve performance close to specialized solutions [36]. 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 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 [32]. For example, learned indexes [37] model the user's data to accelerate searches on that dataset. Instance-optimized data layout techniques [48, 63] 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.

Motivation and Design Principles. 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 [37, 44, 63], we focus on analytic workloads as well. We leave investigation of instance-optimization for transactional workloads to future work.

We used several general principles to guide our design:

(1) **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.
(2) **Minimize the burden on the user.** Configuring the components 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.
(3) **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. We must carefully consider how each component affects the others.

3 DESIGN OVERVIEW

In this section, we provide a high-level overview of SageDB as a system and its instance-optimized components. Section 4 describe the instance-optimized components in more detail, and Section 5 covers the global optimization procedure. Fig. 1 provides an overview.

3.1 System

3.1.1 Storage Layer. SageDB stores data and performs query execution on a single node. 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; each column of each 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

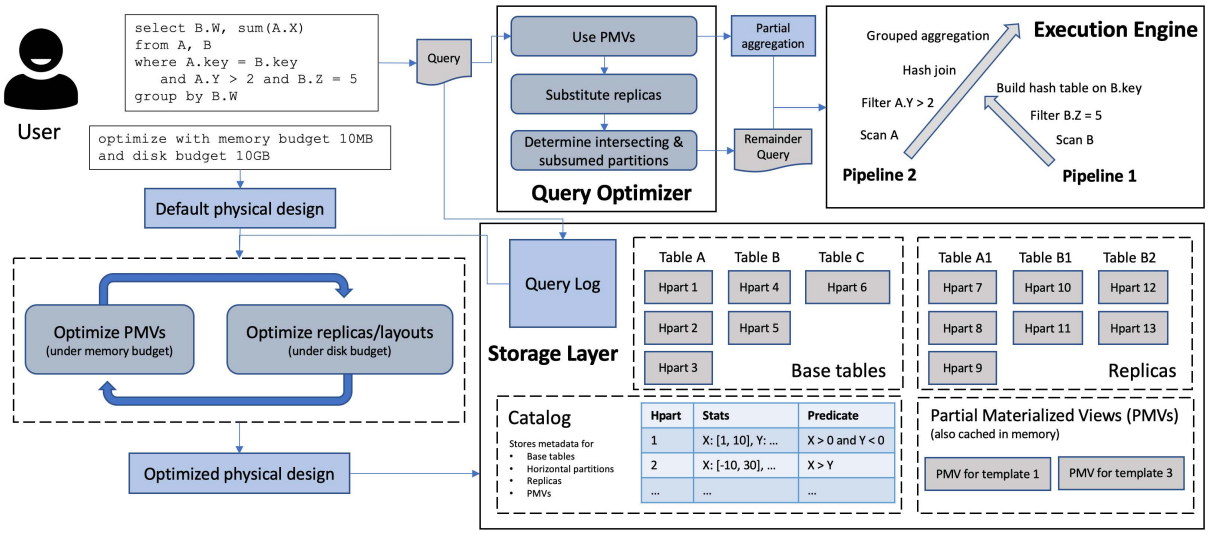

Figure 1: A user query passes through the rule-based optimizer, which determines if and how to use SageDB's instance-optimized components, then runs on SageDB's vectorized execution engine. When users issue an `OPTIMIZE` command, SageDB automatically configures its instance-optimized components to maximize performance based on the user's query history.

Section 4.2 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.

3.1.2 Query Optimizer and 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 (see Fig. 1 for an example). 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. SageDB uses multi-threaded parallel execution of pipelines.

SageDB has a rule-based query optimizer that determines the minimal set of columns that need to be read from each table, determines which horizontal partitions to scan from each table by using per-partition statistics and predicates to skip irrelevant partitions, orders tables for hash joins so that the largest table is the probe side, and constructs the execution pipelines.

3.1.3 Usage and SQL Support. We assume that queries issued by the user contain meaningful patterns and are not completely ad-hoc. More formally, we assume that user queries can be categorized into *templates* (also referred to as prepared statements), which are queries whose filters contain changeable parameters. For example, the template in Fig. 2 has parameters which are represented in the SQL text by ?. SageDB gives users the ability to explicitly create these templates and issue queries by specifying the template ID and the parameter values, as shown in Fig. 2.

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 select-project-join-aggregate queries that can contain `GROUP BY`, `ORDER BY`, `HAVING`, `LIMIT`, `DISTINCT`, and analytic functions. Supported aggregation functions include `COUNT`, `COUNT DISTINCT`, `COUNT APPROX DISTINCT` (using HyperLogLog [27]), `SUM`, `AVG`, `MIN`, and `MAX`, including multi-attribute aggregations. SageDB supports nested queries through through unnesting [50] and treats CTEs as temporary tables. SageDB only supports inner equijoins, implemented as hash joins. SageDB also supports `INSERT`, but it is not a focus of the current design.

3.2 Instance-Optimization

What distinguishes SageDB from traditional systems is the degree to which it is able to customize its design for a specific use case. Many of the techniques that traditional analytic systems use to optimize for a given dataset and workload fall in two categories. First, users are allowed to create materialized views, which are used at query time to substitute a subquery or the entire query itself. This can result in serious performance improvements—some systems' performance relies almost entirely on aggressive use of materialized views [10]—and significant commercial effort has been put on automating materialized view selection [2, 6], maintenance [4], and matching [28].

Second, users are allowed to specify how the records of a table should be sorted. Classically, each table can be sorted by a specified column (i.e., the sort key), and some systems aim to automate sort key selection [1], but newer systems now also support multi-column sort orders such as the Z-order [3, 64]. Contiguous chunks of the sorted records are grouped into blocks, and systems traditionally store per-block metadata, such as the minimum and maximum value

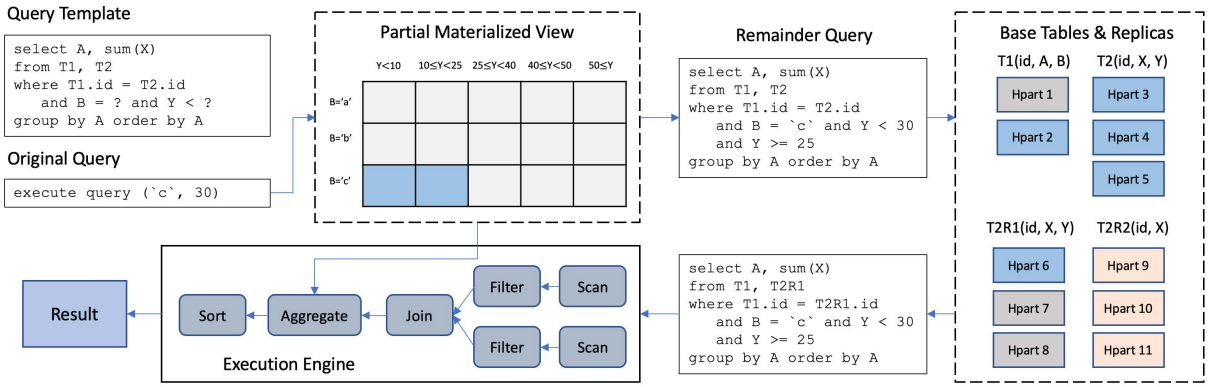

Figure 2: The example query takes advantage of the partial materialized view (PMV) to produce a remainder query with a more selective filter. It then reads from a replica instead of the base table in order to reduce scan cost.

for each column [16, 21, 46, 51], which are used to skip irrelevant blocks during query processing.

Materialized views and data layouts can have a significant impact on performance. However, in traditional systems they are used independently of each other, and furthermore, they are limited in complexity, which can limit their effectiveness. SageDB takes both components, expands their scope to go far beyond the capabilities of traditional systems, and combines them under a single global optimization objective. In particular, SageDB introduces the concept of *partial* materialized views, and SageDB uses instance-optimized block-based data layouts in combination with data replication. We explain these components in depth in Section 4, but we first briefly provide high-level intuition by presenting an example of their usage (Fig. 2).

3.2.1 An Illustrative Example. Assume there are two tables: a small table `T1` with columns `(id, A, B)`; and a large table `T2` with columns `(id, X, Y)`. Assume that the user creates a query template:

```
select A, sum(X) from T1, T2
where T1.id = T2.id and B = ? and Y < ?
group by A order by A
```

Assume that `A` and `B` are a low-cardinality categorical columns, while `Y` is a high-cardinality column whose values are unique floating-point numbers. A *traditional* materialized view for answering queries following this template would look like:

```
select A, B, Y, sum(X) as sumX from T1, T2
where T1.id = T2.id
group by A, B, Y
```

When the user issues a query using this template by specifying values for the parameters, the engine would answer the query directly from this materialized view instead of scanning the base tables, `T1` and `T2`, with a query such as:

```
select A, sum(sumX) from MaterializedView
where B = ? and Y < ?
```

However, since column `Y` has unique values, the materialized view has as many rows as the base table `T2`. This makes the materialized view expensive to store and also greatly reduces its performance benefits. In fact, executing using the materialized view might be slower than scanning the base tables. In this example, the engine would need to scan four columns from the materialized view and apply filters to two of those columns, whereas the original query would only need to read three columns from `T2` and apply filters to one column (the cost of reading and filtering the smaller `T1` are negligible) and perform a potentially inexpensive join.

To avoid the limitations of traditional materialized views, SageDB introduces the concept of *partial* materialized views (PMVs). A PMV is associated with a specific query template. Each cell in the grid (Fig. 2) represents a filtered subset of the joint data distribution of the base tables. For example, the top-left cell represents the data of `T1` and `T2` (joined by `id`) that satisfies the predicate `B='a' and Y<10`. Note that a PMV's grid is specific to a certain join pattern, namely, the join pattern observed in the template.

Each cell stores the result of the template's aggregation over only the data that it represents. For example, the top-left cell would store a relation that is equivalent to the result of executing

```
select A, sum(X) from T1, T2
where T1.id = T2.id and B='a' and Y<30
group by A
```

When executing a query following this template (Fig. 2), the SageDB query optimizer will find the cells that are *subsumed* by (i.e., entirely contained within) the query's filter, which for the example query in the figure are the two cells highlighted. We then produce the *remainder query*, which is the query whose filter has removed the parts that are already subsumed by the PMV (details in Section 4.1) and is therefore much more selective.

Furthermore, unlike traditional systems which allow users to specify a sort order for each table, SageDB has the ability to create multiple partial replicas for each base table (i.e., a replica containing a subset of the columns but all of the records of the base table), each with their own instance-optimized data layout. Before executing the remainder query, SageDB's optimizer considers whether to scan from the base table or a replica. Fig. 2 shows that there are two replicas of `T2`, namely `T2R1` with columns `(id, X, Y)` and `T2R2` with columns `(id, X)`. Imagine that the data layout for `T2R1` has specifically been optimized for the template (Section 4.2 presents more details), so that we would only need to read one horizontal partition from `T2R1` (highlighted in blue), whereas we would need to read all horizontal partitions from `T2`. Therefore the SageDB optimizer would substitute `T2R1` into the query. Note that we cannot use `T2R2` because it does not contain all the necessary columns.

Finally, the modified remainder query is fed to the execution engine, and the partial aggregations from the two subsumed PMV grid cells are merged during the aggregation step.

This example shows how SageDB's instance-optimized components work to reduce query execution cost: first, the PMV eliminates part of the query filter, which reduces the cost of joins (because the join inputs are smaller) and aggregation (because we aggregate fewer records). Second, substitution of base tables with replicas reduces scan cost by reducing the number of horizontal partitions scanned. The former technique is not easily supported in traditional systems; the latter is supported in traditional systems but is limited to simple data layouts (e.g., sort keys) and requires the user to manually specify replicas and layouts, whereas SageDB uses automatically-configured instance-optimized data layouts.

An important part of our contribution is not only *supporting* these techniques in SageDB, but also *automatically optimizing* their configuration. In particular, the performance of each component is dependent on the other. In Section 5, we describe our algorithm for co-optimizing these components given the user's data and workload.

4 INSTANCE-OPTIMIZED COMPONENTS

In this section, we more formally introduce SageDB's instance-optimized components, which we gave intuition for in Section 3.2.1. First, partial materialized views (PMVs) are a novel technique for generalizing traditional materialized views with more degrees of freedom. Second, although the idea of combining instance-optimized data layouts with data replication has been proposed in [60], SageDB's replicated data layouts applies them to the novel context of disk-resident datasets composed of multiple tables, and we introduce a novel optimization algorithm (Section 5.4).

4.1 Partial Materialized Views

A partial materialized view (PMV) is associated with a specific query template. For a given query template (see Section 3.1.3), a *templated column* is a column that is directly involved in a filter predicate that includes a parameter. For the template in Fig. 2, the two templated columns are `B` and `Y`. A partial materialized view (PMV) for a given query template is logically defined as a grid over the templated columns. If a templated column is used twice in the same template (e.g., the filter includes `Y > ? AND Y < ?`), the column is only used once in the grid. For each grid cell, the PMV stores the result of executing the query template over only the data represented by the grid cell.

In concept, several templates can share the same PMV. For example, two templates that have the same filter and group-by clauses but have different aggregations (e.g., template 1 computes `SUM(A)` but template 2 computes `MIN(B)`) can share the same grid. However, this reduces our flexibility to adjust the amount of resources (i.e., memory budget, see Section 5.3) allocated to each template. For example, it is inefficient for a infrequently-queried low-cost template and a frequently-queried high-cost template to share the same grid; instead, the former should have a coarse-grained grid with fewer cells that uses low memory and the latter should have a fine-grained grid with more cells that uses more memory. It is therefore unlikely that two templates have the same optimal PMV grid. Therefore, we decide in SageDB to limit each PMV to a single template.

4.1.1 Construction. Given a PMV grid definition, we construct PMV in a single pass over the data. In fact, the construction can be posed as a SQL query. For example, the PMV in Fig. 2 is constructed as:

```
select A, [CASE WHEN B='a' AND Y<10 THEN 1 ELSE WHEN...], sum(X)
from T1, T2 where T1.id = T2.id
group by A, [CASE WHEN B='a' AND Y<10 THEN 1 ELSE WHEN...]
```

where the `CASE` expression will output a cell number based on the record's value in the templated columns[1]. Note that in the construction query, we remove the parameterized filter predicates, but leave remaining filter predicates as-is (e.g., if there were an additional predicate `AND X>0` in the template).

4.1.2 Usage. To use PMVs at query time, we first logically determine which cells are subsumed by the query filter. We then exclude those regions of the data space from the filter. To determine which cells are subsumed, we break down the filter into its atomic components by splitting apart `AND`s and `OR`s. The example query in Fig. 2 has one `AND`, and therefore two atomic components. Any atomic component that only references a single templated column can be checked against the corresponding grid dimension. For the query in Fig. 2, the atomic component `B='c'` is checked against the partitions of grid dimension `B`, and we see that only one partition is subsumed, and the atomic component `Y<30` subsumes the two partitions that, when combined, represent `Y<25`. An expression describing the subsumed cells can then be constructed by re-combining the atomic components, e.g., `B='c' AND Y < 25`.

To modify the query filter, we add a `NOT` of an expression describing the subsumed regions. For the example query in Fig. 2, the remainder query is

```
select A, sum(X) from T1, T2
where T1.id = T2.id and B='a' and Y<30 and not (B='a' and Y<25)
group by A
```

Note that the expression in parentheses describes the subsumed cells. This may result in an overly complicated filter, but the SageDB optimizer uses an SMT solver [22] to simplify filters into conjunctive normal form (CNF) before passing it to the execution engine.

SageDB caches partial materialized views in memory but they are also persisted to disk and cloud storage.

4.1.3 Strengths and Limitations. The scope of PMVs is quite broad. PMVs can be used for nearly any query template with parameterized filters, since the usage technique is very generic. This idea extends to multiple templated columns, and also to queries with joins (such as the one in Fig. 2), for arbitrary filter predicates (containing both `AND`s and `OR`s).

However, there are some scenarios in which PMVs are unlikely to help (see Section 6.3 for experiments). For templates that produce large aggregations (i.e., group by high-cardinality columns), the PMV becomes expensive to store and has limited benefits[2], similar to the limitation of traditional materialized views presented in Section 3.2.1. Also, if there are many templated columns, the high-dimensional PMV grid is less effective at isolating subsumed cells due to the curse of dimensionality (e.g., Gaming Q4, Section 6.3).

[1] Instead of having a case for every cell, an optimization is have a `CASE` expression for each grid dimension individually, and then combine them to form a unique cell number.
[2] Typically, these types of queries include a `LIMIT` clause (e.g., TPC-H Q10). Unfortunately, we cannot simply take take a `LIMIT` within each cell of the PMV grid, because a global top-K is not equivalent to merging the top-K of each cell.

4.2 Replicated Data Layouts

Prior work on instance-optimized data layouts [23, 25, 48, 63] has already shown that more complex data layouts perform better than traditional single-column or multi-column sort orders. However, these prior instance-optimized techniques assume that they modify the original copy of the data. In SageDB, we want to avoid this because it violates our design principle of avoiding regressions, because an unexpected future query may execute slower on the "optimized" layout than the original layout.

In SageDB, we do not modify the layout of the original copy, which we refer to as *base tables*. Instead, we use a user-provided additional disk space budget to create partial replicas of tables. A partial replica contains a subset of the columns from the base table (which may be the full set). The data layout for each replica is independent. For each query, the query optimizer chooses to read from the replica (or the base table) that minimizes scan cost. The challenge is to determine which subset of the workload to optimize each replica for in order to achieve the best performance, since it does not make sense to optimize multiple replicas for the same query/template if the execution engine only uses one replica at execution time—we examine this optimization problem in Section 5.4.

4.2.1 Construction. For a given replica (i.e., a subset of columns from a base table) and a set of queries to optimize the replica's data layout for, we use the same algorithm as in [63] to create a set of horizontal partitions. SageDB defines a target number of records per horizontal partition, which by default is set to 2M rows based on the latencies we observed for Amazon S3. Each block is associated with a predicate, with the property that all records in the block satisfy the predicate, and also all records that satisfy the predicate are in the block, i.e., blocks do not "overlap." For brevity, we omit the details of the algorithm, which can be found in [63] and is summarized in Section 2.1 of [23].

4.2.2 Usage. For each table referenced in the query, the SageDB optimizer iterates over the replicas, first checks whether the replica contains all the necessary columns, and checks the per-horizontal partition metadata to determine the number of files and rows that need to be read from each, and picks the replica with the lowest cost (see Section 5.1). This procedure is done for each table *independently*, because substituting replicas purely improves scan cost. Downstream operators that introduce dependencies between tables, just as joins, are not affected.

4.2.3 Strengths and Limitations. Replicated layouts have the greatest impact on reducing cost for scan-heavy queries with selective filters. However, replicated data layouts only help reduce scan cost (by skipping irrelevant data blocks) but cannot reduce the cost of other parts of query execution, such as joins, and are therefore less effective for queries where joins dominate execution time (see Section 6.3 for examples). Furthermore, if the query filter is extremely complex (e.g., composed of many conjunctions and disjunctions over many columns), then even instance-optimized data layouts may not be able to meaningfully outperform a full table scan, due to the curse of dimensionality.

5 THE `OPTIMIZE` COMMAND

The user can issue the `OPTIMIZE` command to trigger automatic configuration of SageDB's instance-optimized components. The command has two arguments, a budget for the amount of memory space that SageDB can use to store PMVs, and a budget for the amount of disk space that SageDB can use to store replicated data layouts. The user is allowed to set either budget to zero, though this would of course limit the effectiveness of the optimization.

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.

When the user triggers the `OPTIMIZE` command, SageDB needs to automatically configure its instance-optimized components simultaneously. Why not simply optimize PMVs and replicated layouts independently, each on the full query workload? The choice of PMVs affects the optimal replicated layouts, because PMVs produce remainder queries and in some cases answer the entire query, so the layout should only be optimized for the remainder queries. The choice of replicated layouts also affects the optimal PMVs; depending on how effective the layouts are at processing a template's remainder queries, we may want to allocate more or less memory budget for that template's PMV (e.g., a PMV is useless if the remainder query would anyway require scanning all of the data because of a poor data layout).

SageDB uses an iterative algorithm that optimizes PMVs and layouts, dependent on the other, in a loop until convergence. We now describe the cost model which forms the optimization objective, then the global optimization procedure.

5.1 Cost Model

SageDB uses an analytic cost model. The cost of a query is the sum of scan cost, join cost, and aggregation cost:

$$\begin{aligned}\text{ScanCost} &= w_0(\#\text{ horizontal partitions scanned}) \\ &\quad + w_1(\#\text{ scanned records})(\#\text{ columns read}) \\ \text{JoinCost} &= w_2(\#\text{ build side records}) + w_3(\#\text{ probe side records}) \\ &\quad + w_4(\#\text{ output records})(\#\text{ output columns}) \\ \text{AggCost} &= w_5(\#\text{ aggregated records})\end{aligned}$$

Scan cost and (hash) join cost are evaluated for each table/join, while aggregation cost is computed for the post-join relation. The weights w_i are tuned based on the hardware. To estimate the features, we use a simple cardinality estimator which assumes independence between columns and uniform data distributions of the values in each column. We could use a more complex cost model, or even a learned cost model, but that is orthogonal to the core optimization technique.

5.2 Global Optimization

The optimization objective is to minimize total workload cost, i.e., the sum of costs, according to the cost model, for all queries in the workload. The algorithm is as follows:

(1) The catalog stores a log of all past user queries. We 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. 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.

(2) Starting from the default physical configuration, which only contains the base tables in their original layout and has no PMVs and no replicas, perform the following steps in a loop, until the relative cost decrease from the previous iteration of the loop is less than a certain threshold, by default 1%:
   (a) Optimize the PMVs, using an objective function that takes the current replicated layout configuration into account (see Section 5.3).
   (b) Feed all queries through the optimized PMVs to construct a workload consisting only of remainder queries.
   (c) For each remainder query with joins, push down all single-table predicates to their respective tables and create a single-table query for each table.
   (d) Optimize the replicas and data layouts on the single-table remainder queries (see Section 5.4). Each table is optimized only for the queries that filter on that table.

The intuition behind the loop is to incrementally optimize each component given the currently-optimized configuration of the other component. For example, the first time that PMVs are created, some PMVs may be rejected because the remainder queries would anyways be very expensive on the default layout. However, after the layouts are optimized once, it is likely that those PMVs are selected in the next iteration, because the layout is now optimized. This way, the dependencies between PMVs and replicated layouts are captured.

There are no regressions from one iteration to the next (i.e., cost can only decrease) because the algorithm can always select to choose the same PMVs or replicated layouts as the previous iteration.

5.3 Optimizing Partial Materialized Views

Given a memory budget and a set of templates T, we need an algorithm to decide how much memory to allocate to building a PMV for each template, and also what the PMV grid should be given that memory allocation. We first describe the latter, since it is used as a subroutine in the former.

5.3.1 Optimizing the PMV grid. If a query template t has n templated columns, then a PMV for t is a grid with n dimensions, one representing each templated column (see Fig. 2 for an example). The domain of each dimension's values are logically divided into equally-sized buckets, much like an equi-depth histogram: for each dimension $i \in [0,n)$, we create b_i buckets by setting the boundary values between buckets in such a way that $1/b_i$ of all records fall in each bucket. For templated columns involved in equality filters (i.e., `=`, `!=`, `IN`), we ensure that each bucket only contains one unique value of that column, since a bucket is only useful if the filter is able to subsume it; if there are more unique values than buckets, we only use the b_i most frequent values.

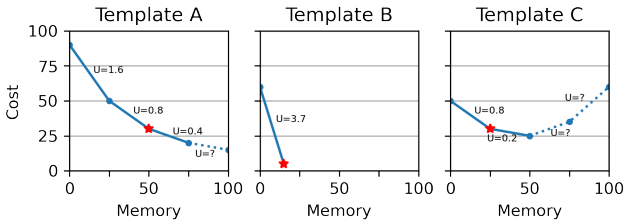

Figure 3: Optimizing PMVs for three templates with a total memory budget of 100 and step size of 25. Utility is visualized as the slope of the lines. Red stars represent the selected configurations. Dotted lines would not actually be considered in the optimization.

Given a query template t and a space budget s, we want to configure a PMV grid, i.e., set b_i for $i \in [0,n)$, that minimizes the total cost, according to our cost model, of executing all queries in the workload that come from t. (Note that scan cost of the remainder query *depends* on the current replicated layout configuration, in the context of the global optimization algorithm in Section 5.2.) We use Bayesian optimization to determine the b_i for each dimension that minimizes cost, under the space budget constraint s. Throughout this process, we make use of SageDB's ability to simulate a PMV without physically creating it, by only storing the PMV metadata (i.e., the grid definition), which is used to generate remainder queries as if the PMV actually existed and to estimate the memory usage of the PMV.

5.3.2 Allocating memory to templates. Given a total memory budget B and a set of templates $T = \{t_1,\ldots,t_n\}$, our algorithm aims to minimize $\sum_i C_i(s_i)$, under the constraint that $\sum_i s_i \le B$, where $C_i(s)$ is the total cost of executing the queries from template t_i if we could create a PMV for t_i with space s, as described above. While solving this optimization problem, we would like to minimize the number of times we compute $C_i(s)$, since PMV simulation, while cheaper than physically creating the PMV, is still expensive.

We make the following observation: allocating more space to a template's PMV generally has diminishing marginal returns. That is, for a template $t_i \in T$, the cost function $C_i(s)$ is convex. For intuition, consider a simple template, `SELECT SUM(A) FROM T WHERE B < ?`, where column B contains numeric values. A PMV for this template would essentially divide the domain of column B into n equally-sized cells. By using the PMV, a query from this template would only ever need to scan/aggregate the data corresponding to one cell, because all cells to the "left" would be subsumed. Each cell contains around $1/n$ of the data, so the cost as a function of the number of cell is approximately $C(n) = 1/n$, which is convex. This intuition also roughly extends to higher-dimensional grids.

Therefore, the intuition behind the optimization algorithm is that instead of allocating the total memory budget across the different templates in one shot, we take an iterative approach where we incrementally allocate more space to the template with the highest impact on cost. Essentially, we do not know the cost functions $C_i(s)$

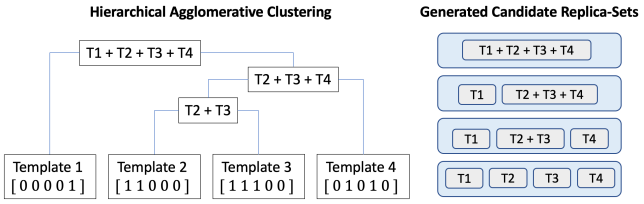

Figure 4: Hierarchical clustering of four query templates on a table with five columns, which produces four candidate replica-sets (blue) over seven distinct replicas (gray).

upfront, so we incrementally explore these cost functions, starting from $s=0$. We now formalize this algorithm.

Our algorithm works by incrementally allocating b memory budget at a time, where $b < B$. We refer to b as the "step size." If a template t_i currently has a PMV that uses memory space s, we define the (marginal) utility $U_i(s,b)$ of allocating another b space as as $(C_i(s) - C_i(s+b))/b$. Throughout the optimization algorithm, we maintain a memo that stores, for each template t_i, three pieces of data: (1) the amount of space currently allocated to that template s_i, (2) the cost $C_i(s_i)$, and (3) the utility $U_i(s_i,b)$ of allocating b more space to the template. The algorithm proceeds as follows (Fig. 3 shows an example):

(1) The memo is initially empty, i.e., zero space is allocated to each template. We begin by computing the utility $U_i(0,b)$ for each template. This requires computing $C_i(b)$ for each template t_i.
(2) Pick the template with the highest utility: $\text{argmax}_i\, U_i(s_i, b)$. Change that template's entry in the memo. That is, if the entry was previously $(s_i, C_i(s_i), U_i(s_i,b))$, we now replace it with $(s_i+b, C_i(s_i+b), U_i(s_i+b,b))$. This requires computing $C_i(s_i+b)$. In case the PMV has reached its maximum size (i.e., we have done the equivalent of a traditional materialized view that groups by the templated columns), we do not consider this template further.
(3) Repeat step 2 until the space budget is filled.

In general, a smaller step size b means a closer-to-optimal solution. By default, we set $b = B/n$, where n is the number of templates. This is small enough to guarantee that every template can get a PMV, if that is indeed optimal. Furthermore, this means that the complexity of the algorithm (i.e., the number of invocations of a cost function) is $O(n)$: we perform $O(n)$ cost function calls to initialize the memo, and we perform $O(B/b) = O(n)$ additional cost function calls before the budget is filled.

5.4 Optimizing Replicated Layouts

Assume the dataset is composed of m tables, $T_1,\ldots,T_m$. Given a total disk budget B and a set of single-table remainder queries Q_i for each table T_i, our algorithm aims to find the set of replicas, along with the data layout for each replica, that minimizes the total scan cost of all remainder queries. Note that we do not need to consider join cost or aggregation cost, since the amount of data scanned does not affect the inputs to downstream operators like joins and aggregations. Our algorithm has two steps: finding a collection of candidate *replica-sets* for each table, then selecting the optimal set of replica-sets.

5.4.1 Generating candidate replica-sets. This step is repeated for each table T_i. Given $|Q_i|$ remainder queries, there are $2^{|Q_i|}$ possible replicas we could create, i.e., we could create a replica whose data layout is optimized for any subset of the queries. For simplicity, we consider each query template as one atomic unit, but nonetheless, given n templates, there are an exponential number of possible replicas, so a brute force search is infeasible. Instead, we generate a collection of promising *replica-sets*, i.e., a set of replicas along with their optimized data layouts.

Our insight is that we should only consider replicas whose layouts are optimized for a set of similar query templates. More concretely, we generate a embedding for each query template, in two different ways that represent two different notions of similarity:

(1) A binary embedding (i.e., composed only of 0's and 1's) of columns that appear in the query filter (in both parameterized and constant predicates). Templates with similar embeddings will benefit from similar layouts. As an extreme example, three templates that all only filter on `colA` will both benefit from a replica that sorts records by `colA`, whereas a replica optimized for three templates that filter on three different columns would not do a great job for any template.
(2) A binary embedding of columns that appear anywhere in the query template, i.e., the columns that the execution engine needs to read when processing this query. By placing templates with similar embeddings in the same cluster, we minimize the number of columns we would need to include in a replica for that cluster (recall that a replica does need to include all columns from the base table, only the columns necessary for execution).

There is a fundamental trade-off between space and cost: having a different replica for every different template will achieve the lowest scan cost, but will take the most disk storage space, while optimizing a single replica for all the templates takes the least space but will not reduce scan cost as much. The candidate replica-sets that we generate should form a Pareto frontier that spans this space-cost tradeoff.

Specifically, use hierarchical agglomerative clustering [59] over the embedded space to separate the n templates into anywhere from 1 cluster to n clusters (see Fig. 4 for an example). For each cluster, we generate a replica whose data layout is optimized (using the algorithm from [63]) for only the templates in its cluster, containing only the columns from the base table needed to execute the templates in its cluster. This results in n replica-sets for each type of embedding, and since we use two types of embeddings, we have up to $2n-2$ unique replica sets (since the replica-sets corresponding to 1 cluster and n clusters will be the same for both embeddings). Due to the nature of hierarchical clustering, these replica-sets are built from up to $3n-3$ unique replicas. Therefore, the time complexity of this step is $O(n)$. For each replica-set, we compute the scan cost of executing the queries Q_i, according to the cost model.

5.4.2 Selecting replica-sets. After generating a collection of candidate replica-sets for each table, we need to select a global configuration of replicas, i.e., select zero or one replica-set for each table, that minimizes total scan cost under the space budget B. This optimization problem is almost identical to the 0-1 knapsack problem, so we use the standard dynamic programming solution to find the optimal collection of replica-sets. The only difference is that if we

Table 1: 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	15

select multiple replica-sets corresponding to the same table, we only use the one that reduces scan cost the most.

If the query workload has n templates and m tables, we generate up to nm candidate replica-sets, so the standard dynamic programming algorithm for the 0-1 knapsack problem takes $O((nm)^2)$ time. In practice, this is very fast, for several reasons: first, the time-intensive optimization steps (i.e., simulating PMVs and replicated layouts and feeding estimated statistics through the cost model) have already been done. Second, n is typically small (e.g., TPC-H has 22 templates). Third, even if there are many tables in the dataset, most of these tables are small; we do not even need to consider creating replicas for tables that only have enough records for one horizontal partition.

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 3× on end-to-end query workloads and up to almost 250× 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`), with data on an attached EBS volume with 4000 IOPS. 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 compare against Umbra [49], a high-performance on-disk analytics research prototype which incorporates many state-of-the-art techniques such as just-in-time code compilation, though it currently doesn't support indexes and cannot be tuned for a particular workload.

We evaluate using three datasets and workloads (Table 1). We include full dataset schema and workload specifications in Appendix A of our extended report [7].

(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.

All experiments that involve running a query workload will first deterministically shuffle the order of queries (i.e., we want to avoid caching effects of running all queries of the same template sequentially). We then run the workload three times and report the median time for each query.

6.2 Overall Results

We first compare SageDB directly against System X and Umbra on the three datasets and workloads. We show two 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 a memory budget of 1GB (which is a small fraction of overall memory) and a disk budget equal to the size of the original dataset (which we believe to be reasonable since datasets are often fully replicated for fault tolerance anyway, especially on the cloud).

We show two different versions of System X: (1) the out-of-the-box configuration, after the data has been loaded. (2) A tuned version, in which we enable System X's ability to automatically select a sort key for each table, as well as automatically select materialized views. We believe that these capabilities represent the state-of-the-art in automated physical design in a large-scale commercial analytic system. To ensure we maximize System X's performance, we performed additional hand-tuning: we included hand-picked materialized views for each dataset, and 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. In summary, the tuned System X reflects the combination of automatic tuning and hand tuning. The disk storage cost of our manually-tuned materialized views are 40%, 2%, and 100% of the size of the original dataset for the Gaming, Stack Overflow, and TPC-H datasets respectively, which is higher than SageDB's 1GB budget for PMVs but smaller than its budget for replicated data layouts; System X does not allow users to access automatically-created materialized views, so the total storage cost of all materialized views is likely higher. Umbra does not use any extra storage space because it does not support indexes.

Fig. 5 shows that across the three workloads, SageDB outperforms the other systems on average query runtime by up to 3×. As evidence of the effectiveness of SageDB's instance-optimized components, SageDB optimized outperforms the unoptimized version of itself by between 3–6×, whereas System X tuned, which uses a combination of manual tuning and state-of-the-art automatic tuning, achieves between 25% and 3× performance gain over the default version of itself. Umbra performs best when the working set fits in memory; otherwise it is bottlenecked by disk since it doesn't use any indexes or layouts, which is why it performs poorly on TPC-H. 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. 5 also shows a per-template breakdown of speedups achieved by the optimized version of SageDB compared to the tuned version of System X. For individual query templates, median speedups are as high as 250× (Gaming Q8). In general, templates for which SageDB performs worse than System X are ones for

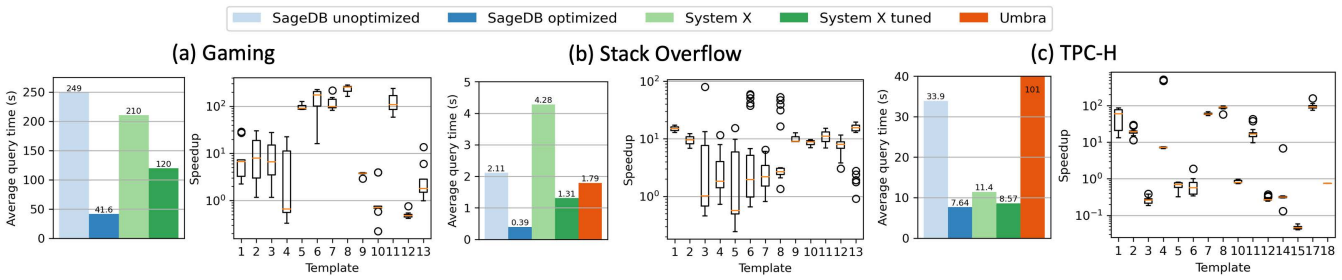

Figure 5: For each dataset, we show average query time on each system for the end-to-end workload, as well as a per-template breakdown of speedups achieved by SageDB optimized compared to System X tuned. SageDB outperforms other systems by up to 3× on end-to-end query workloads and achieves up to almost 250× speedup for individual templates.

which SageDB's instance-optimized components do not make an impact (e.g., TPC-H Q18, see Section 6.3), ones for which the tuned System X has sort keys and materialized views that achieve the same purpose as SageDB's instance-optimized components (e.g., Gaming Q12), or ones for which System X's raw execution engine is simply more efficient than SageDB's (e.g., some TPC-H templates). The variability in speedups is simply due to the fact that the effectiveness of SageDB's instance-optimization depends not only on the query template, but also the specific parameter values of the template; for example, a parameter value that results in a non-selective filter may not benefit as much from replicated layouts as a selective filter.

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 outer joins). 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 6×. 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 2, 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, replicated data layouts are especially helpful for queries that either filter on a single table (e.g., Stack Overflow queries, TPC-H Q1) or have inexpensive joins (e.g., TPC-H Q14). PMVs are helpful whenever they are applicable, and especially if it fully answers the query so that the remainder query is empty (e.g., Gaming Q8 and Stack Overflow Q1).

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 PMVs, which answer most of the query, and then using the replicated data layouts to speed up the remainder query.

Third, there are some types of queries for which PMVs or replicated data layouts make no impact, as we alluded to in Sections 4.1.3 and 4.2.3. For example, TPC-H Q18 produces extremely large aggregations (since it groups by the primary key of a table with 150M rows before applying a `LIMIT`), so a PMV is unhelpful and would exceed the memory budget anyway.

Fourth, occasionally using instance-optimized component is worse than not using it. For example, on Stack Overflow Q8, using PMVs decreases performance compared to the default. This is because SageDB always uses PMVs if they exist, but in this particular case, the query itself ran relatively quickly, and the extra optimizer overhead from computing subsumed cells in the PMV 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 2 showed that on a query template level, performance does not regress. Fig. 6 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. Fig. 5 showed SageDB's performance when optimized with 1GB memory budget and disk budget equal to the size of the original dataset. To show how performance would change if the budgets were set differently, we hold one budget constant while varying the other budget, on the Gaming dataset. Fig. 7 shows the overall workload cost as each budget varies, compared to the cost of having zero budget (i.e., if the corresponding component were disabled). As more space is given, cost decreases and performance improves. Note that the cost curve is convex for PMV optimization (note the log scale for the x-axis), confirming our intuition from Section 5.3. For replicated layout optimization, there is a significant decrease in cost at 70% disk space because that is the boundaries past which an especially important replica fits within the space budget.

Table 2: 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
PMVs only	1.7	1.68	1.72	1.0	51.9	99.7	93.8	1.28e+03	2.43	1.12	77.5	1.01	1.36
Replicated layouts only	1.25	1.32	1.33	1.65	1.14	1.05	1.05	1.06	1.0	1.1	1.12	1.0	1.21
All	3.1	3.76	3.11	1.65	51.9	1.17e+02	1.05e+02	1.33e+03	2.96	1.17	85.5	1.02	1.81

Stack Overflow	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13
PMVs only	10.5	1.36	1.55	2.16	1.9	1.39	7.75	0.648	1.39	6.3	1.2	2.78	2.04
Replicated layouts only	7.0	1.53	1.95	3.19	3.38	4.33	9.95	2.43	3.61	13.6	1.09	2.21	15.3
All	17.1	1.6	2.24	4.78	3.66	4.33	10.9	2.44	3.82	14.2	1.71	2.78	15.6

TPC-H	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q10	Q11	Q12	Q14	Q15	Q17	Q18
PMVs only	3.17	25.9	1.0	1.18	2.04	1.0	2.47e+02	63.3	2.46	1.78e+02	1.08	1.09	1.02	27.3	1.0
Replicated layouts only	56.7	1.03	1.02	1.05	1.03	5.18	1.07	1.04	1.05	1.07	6.42	1.69	3.12	1.04	1.05
All	80.1	27.3	1.03	1.26	2.13	6.02	2.54e+02	67.2	3.32	2.02e+02	7.97	2.12	3.15	29.1	1.05

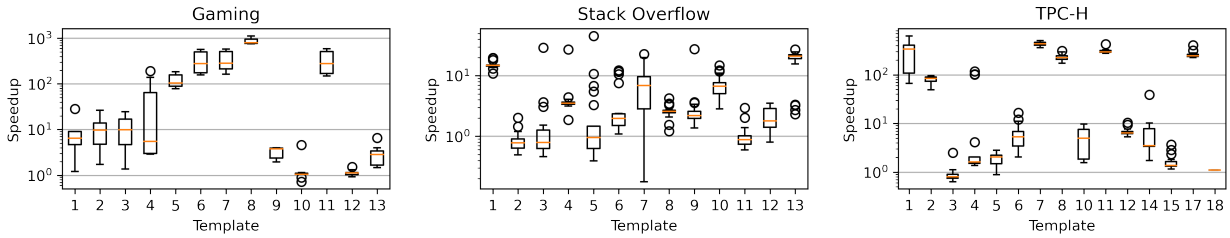

Figure 6: Per-query speedups for SageDB compared to its unoptimized configuration. Regressions are rare. The orange line represents the median; the box represents first and third quartiles; whiskers extend from the box by 1.5× the inter-quartile range; dots are those past the end of the whiskers.

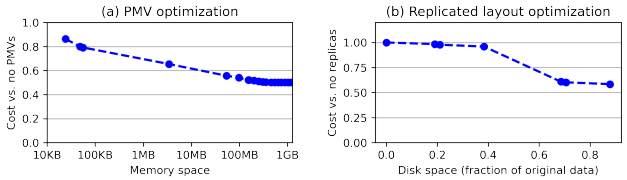

Figure 7: Gaming dataset: cost decreases as more space is provided to the `OPTIMIZE` command.

Table 3: Optimization Time (in seconds).

	Gaming	**Stack Overflow**	**TPC-H**
Optimization algorithm	107	117	143
PMV construction	4340	265	4150
Replicated layout construction	16100	1280	2400
Total	20500	1660	6690

6.4.3 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 3 breaks down the time that SageDB spends on each step of optimization for each dataset. Overall, optimization finishes in less than 6 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 (Fig. 5), we recoup the time "lost" to optimization after executing only around 100 queries.

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.

Avoid regression. Due to SageDB's design, for any particular query we can always fall back to the default out-of-the-box configuration without instance-optimization. For example, the optimizer can always choose to read from the base table instead of from the replicas. Indeed, we show in Section 6.4 that we avoid regressions on all templates.

The implication is that the burden of avoiding regressions (e.g., deciding to not use the PMV for a certain query) falls on the query optimizer, which may make mistakes due to inaccurate cost models or cardinality estimates. Besides integrating a more sophisticated query optimizer, one way to guard more aggressively against regression

is to allow the optimizer to use the instance-optimized components only when cost reduction is greater than a certain threshold.

Minimize the burden on the user. SageDB places minimal technical burden on the user: their only responsibility is to issue an `OPTIMIZE` command, with a 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. This will require detecting when the data or the workload have shifted enough to merit a re-optimization, and can incorporate ideas from [23]. It will also require considering the cost of re-optimization itself, as well as forecasting the future workload—even if the workload has shifted, we may not want to optimize if the workload will shift again soon anyway.

Inserts. 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. Replicas with instance-optimized data layouts can avoid invalidation through delta buffering. For example, new data is inserted into a special horizontal partition. Existing horizontal partitions remain unchanged, and when scanning, SageDB can still take advantage of data skipping over existing partitions, but may need to always read the new partition. When the user again triggers the `OPTIMIZE` command, the buffered data in the new partitions are incorporated into the new data layout.

Likewise, PMVs are not necessarily invalidated when data is inserted into a new horizontal partition, especially if data is only changing in one base table (e.g., users append data to a fact table but the dimension tables are stable); we essentially execute two separate queries, one over the data over which the PMV was constructed and another over the new/buffered data, and merge the results. At a later time, we can perform incremental maintenance on the PMV, by essentially building a new PMV over the new/buffered data, with the same grid definition as the existing PMV, and then merging each cell with its counterpart in the existing PMV.

Expanding Components. Since SageDB so far has focused primarily on physical design, good candidates for components to add next are ones that improve the logical side, e.g., a learned query optimizer or a learned cost model. We believe the main challenge will be to keep these components "in sync." For example, the physical design optimization depends on the query optimizer (especially its ability to simulate PMVs and layouts) and cost model. If the optimizer or cost model changes due to retraining, then some pieces of the physical design might no longer be selected by the optimizer.

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 [14], such as selecting indexes [11, 31], partitioning schemes [12, 17, 54], or materialized views [11]. Based on the method used to find the ideal configuration, the previous work can be divided into two categories: rule-based methods [18, 39] and ML-based methods [26, 41, 42, 53, 62, 65]. Cosine [13] 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 [52] 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 [25, 48, 63]. There are now instance-optimized versions of bloom filters [20, 47, 61] and hash tables [57]. New use cases, from caching [34, 40] to query optimization [38, 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.

Computation Reuse. PMVs can be considered a form of computation reuse, in which we pre-materialize certain results that will be used multiple times in the future. Other forms of computation reuse are multi-query optimization [56, 58] (which aims to find a globally optimal execution plan for a batch of queries), materialized views [35], data cubes [29], and sub-expression materialization [33]. These techniques all assume that (sub)queries must be fully processed using pre-computed or cached results, whereas PMVs have the flexibility of *partially* answering the query, while the cheaper remainder query scans the base data.

Replication and Data Layouts While there have many works on instance-optimized data layouts [23, 25, 48, 63], the only other work to consider the combination of partial replication with instance-optimized data layouts is CopyRight [60]. However, their optimization algorithm makes assumptions that are specialized for grid-based data layouts for in-memory data over a single table, while SageDB handles multi-table disk-based datasets.

9 CONCLUSION

In this paper, we presented a progress report on SageDB, a first instance-optimized data system, focused on analytics. SageDB incorporates two instance-optimized components into one system that exposes a simple interface to the 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.

ACKNOWLEDGMENTS

This research is supported by the MIT Data Systems and AI Lab (DSAIL), NSF IIS 1900933, and a Meta Research PhD Fellowship.

REFERENCES

[1] [n.d.]. Amazon Redshift Automatic Table Optimization. https://docs.aws.amazon.com/redshift/latest/dg/t_Creating_tables.html

[2] [n.d.]. Amazon Redshift AutoMV. https://docs.aws.amazon.com/redshift/latest/dg/materialized-view-auto-mv.html

[3] [n.d.]. Databricks Delta Lake Z-Ordering. https://docs.databricks.com/delta/optimizations/file-mgmt.html#z-ordering-multi-dimensional-clustering

[4] [n.d.]. Materialize. https://materialize.com/

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

[6] [n.d.]. Oracle Automatic Materialized Views. https://docs.oracle.com/en/database/oracle/oracle-database/21/tgdba/auto_material_views.html

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

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

[9] 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

[10] 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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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

[16] Zach Christopherson. 2016. Amazon Redshift Engineering's Advanced Table Design Playbook: Compound and Interleaved Sort Keys. https://aws.amazon.com/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-compound-and-interleaved-sort-keys/

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

[18] 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.

[19] 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

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

[21] Databricks. 2020. Data skipping index. https://docs.databricks.com/spark/latest/spark-sql/dataskipping-index.html

[22] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[23] 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

[24] 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

[25] 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

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

[27] 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

[28] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. *SIGMOD Rec.* 30, 2 (may 2001), 331–342. https://doi.org/10.1145/376284.375706

[29] 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

[30] 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

[31] 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.

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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*.

[37] 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

[38] 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

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

[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] Microsoft. 2019. Columnstore indexes - Query performance. https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-query-performance

[47] 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

[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] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings (LNI)*, Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath (Eds.), Vol. P-241. GI, 383–402. https://dl.gi.de/20.500.12116/2418

[51] Oracle. 2020. Database Data Warehousing Guide: Using Zone Maps. https://docs.oracle.com/database/121/DWHSG/zone_maps.htm

[52] 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

[53] 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

[54] 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.

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

[56] 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

[57] 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

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

[59] R. Sibson. 1973. SLINK: An optimally efficient algorithm for the single-link cluster method. *Comput. J.* 16, 1 (01 1973), 30–34. https://doi.org/10.1093/comjnl/16.1.30 arXiv:https://academic.oup.com/comjnl/article-pdf/16/1/30/1196082/160030.pdf

[60] Sivaprasad Sudhir, Michael Cafarella, and Samuel Madden. 2021. Replicated Layout for In-Memory Database Systems. *Proc. VLDB Endow.* 15, 4 (dec 2021), 984–997. https://doi.org/10.14778/3503585.3503606

[61] 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

[62] 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.

[63] 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

[64] Zack Slayton. 2017. Z-Order Indexing for Multifaceted Queries in Amazon DynamoDB. https://aws.amazon.com/blogs/database/z-order-indexing-for-multifaceted-queries-in-amazon-dynamodb-part-1/.

[65] 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 DATE,
  poster_name TEXT,
  poster_reputation INT32,
  poster_join_date DATE,
  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
)
```

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

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

Q3: `SELECT EXTRACT(YEAR FROM post_date) AS 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 EXTRACT(YEAR FROM post_date) ORDER BY post_year;`

Q4: `SELECT EXTRACT(YEAR FROM post_date) AS post_year, COUNT(*) FROM denorm_so WHERE answered = 0 AND comment_max_score`

```
>= ?:INT32 GROUP BY EXTRACT(YEAR FROM post_date) ORDER
BY post_year;

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

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 EXTRACT(YEAR FROM post_date) AS post_year,
COUNT(*) FROM denorm_so WHERE
favorite_count <= ?:UINT64 AND post_date >= ?:FLOAT64
GROUP BY EXTRACT(YEAR FROM post_date) ORDER BY post_year;

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

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

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

Q11: SELECT EXTRACT(YEAR FROM post_date) AS post_year,
COUNT(*) FROM denorm_so WHERE answered = 0 AND score >=
?:INT32 AND tag_count >= 4 GROUP BY EXTRACT(YEAR FROM
post_date) 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 specification.

A.3.2 Workload.

```
Q1: select l_returnflag, l_linestatus, sum(l_quantity)
as sum_qty, sum(l_extendedprice) as sum_base_price, sum(l_extendedprice*(1-l_discount))
as sum_disc_price, sum(l_extendedprice*(1-l_discount)*(1+l_tax))
as sum_charge, avg(l_quantity) as avg_qty, avg(l_extendedprice)
as avg_price, avg(l_discount) as avg_disc, count(*) as
count_order from lineitem where l_shipdate <= ?:DATE
group by l_returnflag, l_linestatus order by l_returnflag,
l_linestatus;

Q2: select s_name, sum(s_acctbal) as balance from part,
supplier, partsupp, nation, region where part.p_partkey
= ps_partkey and s_suppkey = ps_suppkey and s_nationkey
= n_nationkey and n_regionkey = r_regionkey and p_size
= ?:INT32 and region.r_name = ?:TEXT and ps_supplycost
= ( select min(ps_supplycost) from partsupp, supplier,
nation, region where p_partkey = ps_partkey and s_suppkey
= ps_suppkey and s_nationkey = n_nationkey and n_regionkey
= r_regionkey and region.r_name = ?:TEXT ) group by s_name
order by balance limit 100;

Q3: select sum(l_extendedprice*(1-l_discount)) as revenue,
o_orderdate, o_shippriority from lineitem, orders, customer
where c_custkey = o_custkey and l_orderkey = o_orderkey
and c_mktsegment = ?:TEXT and o_orderdate < ?:DATE and
l_shipdate > ?:DATE group by o_orderdate, o_shippriority
order by revenue, o_orderdate limit 10;

Q4: select o_orderpriority, count(*) from orders where
o_orderdate >= ?:DATE and o_orderdate < ?:DATE and exists
( select * from lineitem where l_orderkey = o_orderkey and
l_commitdate < l_receiptdate ) group by o_orderpriority
order by o_orderpriority;

Q5: select n_name, sum(l_extendedprice * (1 - l_discount))
as revenue from lineitem, orders, customer, supplier,
nation, region where c_custkey = o_custkey and l_orderkey
= o_orderkey and l_suppkey = s_suppkey and c_nationkey
= s_nationkey and s_nationkey = n_nationkey and n_regionkey
= r_regionkey and r_name = ?:TEXT and o_orderdate >=
?:DATE and o_orderdate < ?:DATE group by n_name order by
revenue desc;

Q6: select sum(l_extendedprice*l_discount) from lineitem
where l_shipdate >= ?:DATE and l_shipdate < ?:DATE and
l_discount >= ?:FLOAT64 and l_discount <= ?:FLOAT64 and
l_quantity < ?:FLOAT64;

Q7: select n1.n_name as supp_nation, n2.n_name as cust_nation,
sum(l_extendedprice * (1 - l_discount)) from lineitem,
orders, supplier, customer, nation n1, nation n2 where
s_suppkey = l_suppkey and o_orderkey = l_orderkey and
c_custkey = o_custkey and s_nationkey = n1.n_nationkey
and c_nationkey = n2.n_nationkey and ((n1.n_name = ?:TEXT
and n2.n_name = ?:TEXT) or (n1.n_name = ?:TEXT and n2.n_name
= ?:TEXT)) and l_shipdate >= 19950101 and l_shipdate
<= 19961231 group by n1.n_name, n2.n_name order by supp_nation,
cust_nation;

Q8: select o_year, sum(case when nation = ?:TEXT then volume
else 0 end) / sum(volume) from ( select extract(year from
o_orderdate) as o_year, l_extendedprice * (1-l_discount) as
volume, n2.n_name as nation from part, supplier, lineitem,
orders, customer, nation n1, nation n2, region where p_partkey
= l_partkey and s_suppkey = l_suppkey and l_orderkey
= o_orderkey and o_custkey = c_custkey and c_nationkey
= n1.n_nationkey and n1.n_regionkey = r_regionkey and
r_name = ?:TEXT and s_nationkey = n2.n_nationkey and
o_orderdate >= 19950101 and o_orderdate <= 19961231 and
p_type = ?:TEXT group by n2.n_name;

Q10: select c_custkey, n_name, sum(l_extendedprice * (1 -
```

```
l_discount)) as revenue from lineitem, orders, customer,
nation where c_custkey = o_custkey and l_orderkey =
o_orderkey and o_orderdate >= ?:DATE and o_orderdate
< ?:DATE and l_returnflag = 'R' and c_nationkey = n_nationkey
group by c_custkey, n_name order by revenue desc limit
20;

Q11: select ps_partkey, sum(ps_supplycost * ps_availqty)
as value from partsupp, supplier, nation where ps_suppkey
= s_suppkey and s_nationkey = nation.n_nationkey and
n_name = ?:TEXT group by ps_partkey order by value limit
10;

Q12: select l_shipmode, sum(case when o_orderpriority =
'1-URGENT' or o_orderpriority = '2-HIGH' then 1 else 0
end) as high_line_count, sum(case when o_orderpriority
!= '1-URGENT' and o_orderpriority != '2-HIGH' then 1
else 0 end) as low_line_count from orders, lineitem
where o_orderkey = l_orderkey and l_shipmode = ?:TEXT
and l_commitdate < l_receiptdate and l_shipdate < l_commitdate
and l_receiptdate >= ?:DATE and l_receiptdate < ?:DATE
group by l_shipmode order by l_shipmode;

Q14: select sum(case when p_size <= 5 then l_extendedprice
* (1 - l_discount) else 0.0 end), sum(l_extendedprice *
(1 - l_discount)) from lineitem, part where l_partkey
= p_partkey and l_shipdate >= ?:DATE and l_shipdate <
?:DATE;

Q15: select l_suppkey, sum(l_extendedprice * (1 - l_discount))
as total_revenue from lineitem where l_shipdate >= ?:DATE
and l_shipdate < ?:DATE group by l_suppkey order by
total_revenue desc limit 10;

Q17: select sum(0.7 * l_extendedprice) from lineitem,
part where p_partkey = lineitem.l_partkey and p_brand =
?:TEXT and p_container = ?:TEXT and l_quantity < ( select
0.2 * avg(l_quantity) from lineitem where l_partkey =
p_partkey );

Q18: select c_name, c_custkey, o_orderkey, o_orderdate,
o_totalprice, sum(l_quantity) from customer, orders, lineitem
where c_custkey = o_custkey and o_orderkey = l_orderkey
and o_orderkey in ( select l_orderkey from lineitem
group by l_orderkey having sum(l_quantity) > ?:FLOAT64
) group by c_name, c_custkey, o_orderkey, o_orderdate,
o_totalprice order by o_totalprice, o_orderdate limit
100;
```