Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems

Lukas Vogel
TU München
lukas.vogel@in.tum.de

Alexander van Renen
TU München
renen@in.tum.de

Satoshi Imamura
Fujitsu Laboratories Ltd.
s-imamura@fujitsu.com

Viktor Leis
Uni Jena
viktor.leis@uni-jena.de

Thomas Neumann
TU München
neumann@in.tum.de

Alfons Kemper
TU München
kemper@in.tum.de

ABSTRACT

Relational database systems are purpose-built for a specific storage device class (e.g., HDD, SSD, or DRAM). They do not cope well with the multitude of storage devices that are competitive at their price 'sweet spots'. To make use of different storage device classes, users have to resort to workarounds, such as storing data in different tablespaces. A lot of research has been done on heterogeneous storage frameworks for distributed big data query engines. These engines scale well for big data sets but are often CPU- or network-bound. Both approaches only maximize performance for previously purchased storage devices.

We present Mosaic, a storage engine for scan-heavy workloads on RDBMS that manages devices in a tierless pool and provides device purchase recommendations for a specified workload and budget. In contrast to existing systems, Mosaic generates a performance/budget curve that is Pareto-optimal, along which the user can choose. Our approach uses device models and linear optimization to find a data placement solution that maximizes I/O throughput for the workload. Our evaluation shows that Mosaic provides a higher throughput at the same budget or a similar throughput at a lower budget than the state-of-the-art approaches of big data query engines and RDBMS.

PVLDB Reference Format:
Lukas Vogel, Alexander van Renen, Satoshi Imamura, Viktor Leis, Thomas Neumann, and Alfons Kemper. Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems. *PVLDB*, 13(11): 2662-2675, 2020.
DOI: https://doi.org/10.14778/3407790.3407852

1. INTRODUCTION

For analytical queries on large data sets, I/O is often the bottleneck of query execution. The simplest solution is to store the data set on fast storage devices, such as PCIe SSDs. While it is prohibitively expensive to store all data on such devices, systems can leverage the inherent hot/cold clustering of data. Workloads often have a small working set, and storing the cold data on fast, but expensive devices wastes money. It would be better for a storage engine to store it on a cheap HDD instead, as no performance penalty is incurred.

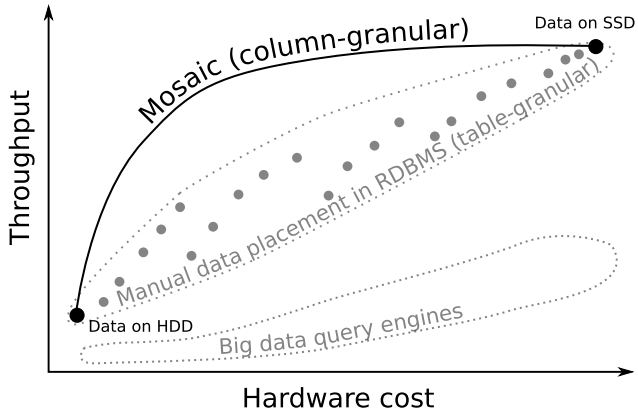


Figure 1: Estimated performance spectrum of Mosaic compared to big data query engines like Spark and manual data placement.

Traditional relational database systems (RDBMS) are unsuitable for this task. Most are optimized for a specific class of storage devices and assume that all data will be stored on a device of the given class. Traditional RDBMS, such as PostgreSQL or MySQL, are optimized for HDD and only maintain a small DRAM cache. Modern systems like HyPer [23], SAP HANA [10], or Microsoft Hekaton [8] are built for DRAM. Our database system, Umbra [28], is optimized for SSD. Some allow system administrators the freedom to choose where to place data, even if they are not designed for multiple types of storage devices, for instance, via tablespaces. Here, the administrator can choose the storage location (and thus the storage device) for each table. However, moving an entire table either wastes fast storage space or negatively impacts on performance, as a table's cold columns are always moved together with its hot columns. Therefore, enabling *column-granular* placement allows for a much more *cost-efficient* storage allocation.

This problem is well-known in the big data world. Big data query engines like Spark [41] are therefore optimized for column-major storage formats like Parquet [18]. These file formats support the splitting of tables and their columns

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407852

into multiple files, so that they can be distributed between multiple nodes. Heterogeneous, tiered storage frameworks, such as OctopusFS [21], hatS [30], or CAST [5], distribute these chunks over multiple devices. They are very good at eking out every ounce of performance from the storage devices. Their downside is that they cannot judge if the provisioned storage devices are a good fit for the workload, as they are installed after the system has already been purchased and provisioned. Big data query engines using such storage frameworks are often distributed systems optimized for cluster operation. While this enables scaling to very large data sets, it incurs significant CPU and networking overheads. Queries are therefore more frequently CPU- or network-bound than is the case with traditional RDBMS.

It would thus be beneficial to have column-granular table placement on RDBMS. While table-granular placement is not optimal for the reasons mentioned above, a user can at least manually determine a sensible placement on the basis of their experience. For column-granular placement, however, it is considerably harder to find a good solution by manual means, as the number of possible placements grows exponentially in the number of columns and storage devices.

Big data engines and RDBMS with heterogeneous storage frameworks have another shortcoming. A modern server can have storage devices of multiple classes: DRAM, Persistent Memory, NVMe SSDs, SATA SSDs, and HDDs in different RAID configurations, and all are competitive at their price point. A system administrator buying a new database server cannot determine the optimum configuration that will achieve the required throughput at the lowest cost.

We therefore propose Mosaic, a storage engine for RDBMS that is optimized for scan-heavy workloads and covers the entire deployment process of a database system: (1) hardware selection, (2) data placement on purchased hardware, and (3) adaption to changing workloads. Mosaic uses purchased storage devices to their full potential with column-granular placement, ensuring an optimum throughput/performance ratio at *all* budgets. So as not to restrict the user in the purchase process, Mosaic does not categorize storage devices into tiers but organizes all devices in a tierless pool. A conventional storage engine, in contrast, has distinct tiers (e.g. HDD, SSD, and DRAM). Mosaic's tierless design allows the user to mix device classes (e.g. adding an NVMe SSD to a system already equipped with a SATA SSD, where a tiered approach would only have an SSD tier).

Figure 1 compares Mosaic to existing approaches. The x-axis shows the cost of the installed storage devices, the y-axis the throughput of the system. Big data engines do not scale well with the price of the storage devices used as they are rarely I/O-bound, even for smaller workloads. RDBMS scale well but offer no automated mechanism for data placement and are restrained to table-granular placement. Manual data placement does not guarantee that the choice is Pareto-optimal or fits the data set. Mosaic's automatic column-granularity placement not only increases throughput for scan-heavy workloads at all price points, but it also empowers the user to find the best configuration within a given budget or subject to specific performance requirements.

Mosaic is an improvement over existing solutions during all stages of the deployment process: (1) Before hardware is purchased: given a typical set of queries and a list of devices available for purchase, Mosaic gives purchase recommendations for arbitrary budgets. Each recommendation is guaranteed to be Pareto-optimal, i.e., no other device configuration is faster while also being cheaper. (2) After purchase: given the trace of a typical set of queries and a set of storage devices, Mosaic places data optimally to maximize throughput. Mosaic can work with any set of storage devices, not only those that have been bought on the basis of its recommendations. (3) During operation: Mosaic acts as a pluggable storage engine for an RDBMS.

In summary, our key contributions are:

1. We present Mosaic, a column-based storage engine for RDBMS, optimized for scan-heavy OLAP workloads and using a device pool without fixed tiers. In contrast to existing approaches, it places data with column granularity.
2. We design a placement algorithm based on linear optimization that finds optimum data placement for a workload.
3. We design a prediction component for Mosaic that gives purchasing recommendations along a Pareto-optimal price/performance curve.
4. We integrate Mosaic into our DBMS, Umbra [28], and point out its benefits over state-of-the-art approaches.

2. BACKGROUND AND RELATED WORK

While, to the best of our knowledge, Mosaic has no direct competitor, all of its design goals have been achieved in other systems individually, but never together. These systems are therefore not able to leverage the synergies of implementing *all* of Mosaic's design goals.

2.1 Heterogeneous Storage Frameworks for Big Data Query Engines

Data set sizes have over time outgrown the storage capacity of single systems, which is why big data engines were introduced. Most query engines, such as MapReduce [6] or Spark [41], support the Hadoop file system (HDFS) [33]. This splits files into blocks, which it replicates across nodes. Until recently, nodes were unaware of the characteristics of their storage device and therefore could not place data in a workload-aware fashion.

Multiple extensions to HDFS have been developed to rectify this issue. Kakoulli et al. implement OctopusFS [21], a tiered distributed file system based on HDFS. OctopusFS uses a model to infer a data placement for a fixed set of tiers (DRAM, SSD, and HDD) that maximizes throughput. The authors later built on OctopusFS and developed an automated tiered storage manager [14] that uses machine learning to decide on which blocks to up- or downgrade.

CAST [5] recognizes that cost models and tiering mechanisms used for operating systems do not solve problems of OLAP style workloads, as they rely on access characteristics that are atypical for an OS, i.e., large, sequential table scans. Multiple other works have introduced a heterogeneity-aware layer on top of HDFS using fixed tiers [17, 30, 31, 32]. Snowflake does not rely on HDFS but uses its own tiered distributed storage system, optimized for cloud operation [37].

What all solutions building on HDFS have in common is that they focus on opaque HDFS blocks as atomic units of storage. As they do not know what is stored inside those blocks, they cannot make domain-specific optimizations. Mosaic knows its domain (retrieving columns for table scans in an OLAP context), and its placement strategies can take this into account. Mosaic deliberately decides against

a tiered architecture common in HDFS approaches, as new hardware does not always cleanly map onto existing tiers.

The approaches referred to in this section do not offer any purchase recommendations. In contrast to Mosaic, they have to manage replication and data locality, as they run on clusters. While replication is orthogonal to Mosaic (i.e. one could extend Mosaic to replicate data), we focus on a single machine for now, to simplify the data model.

2.2 Heterogeneous Storage in RDBMS

RDBMS hide the throughput gap between DRAM and background storage with a buffer pool. In the last decade, when SSDs became affordable, a lot of work was done to exploit their improved throughput and random access characteristics. For example, Umbra, our database system, is optimized for SSDs [28]. It provides main memory-like speed when the working set fits into the DRAM buffer pool and gracefully degrades to SSD-speed with larger working sets, an idea first implemented in LeanStore [25].

Novel algorithms have been proposed for caching data from SSD into DRAM [19, 20, 35] or using SSDs as a cache for HDDs [15]. MaSM [3], for example, uses an SSD cache for out-of-place updates. DeBrabant et al. [7] introduce anti-caching, where DRAM is the primary storage device, and cold data is evicted to HDD. Stoica and Ailamaki [34] reorganize cold data so that the OS can efficiently page it out. Another approach is to build buffer pools with multiple tiers [9, 22]. Here, SSD is a caching layer for slower devices like HDDs. These approaches, however, waste valuable storage space on SSDs, as data is replicated across multiple tiers. While caching is necessary for volatile devices like DRAM, it is not needed for persistent storage.

Hybrid storage managers circumvent this issue by placing the data on different device classes without caching [4, 26, 27, 42]. hStorage-DB [27], like Mosaic, uses information from the query engine to place data on HDDs and SSDs.

These approaches focus on HDD and SSD. With new storage technologies, such as persistent memory, the whole cycle of research begins anew as RDBMS now have to integrate a new layer of storage [2, 36]. Mosaic finally breaks this cycle by being device-agnostic. Every device is instead parameterized by the user and added to a tierless pool. The user can add or remove new device classes without having to re-engineer Mosaic.

General-purpose data placement systems [16, 29, 39] are not restricted to relational data and therefore, unlike Mosaic, cannot exploit domain knowledge.

2.3 Prediction and Storage Recommendation

Mosaic's prediction component also builds on prior work. Wang et al. built a MapReduce simulator [38] to investigate the impacts of different design decisions, such as data placement or device parameters, on performance. In contrast to Mosaic, their tool only plans new setups and does not act as a storage engine. Herodotou et al. designed a 'what if' [13] engine capable of comparing different configurations and giving recommendations for MapReduce jobs. Cheng et al. went a step further and designed CAST [5], a tiered storage framework for MapReduce jobs. It gives data placement recommendations for a cloud context that minimize cost while maintaining performance guarantees. However, they also limit themselves to a predefined set of device classes.

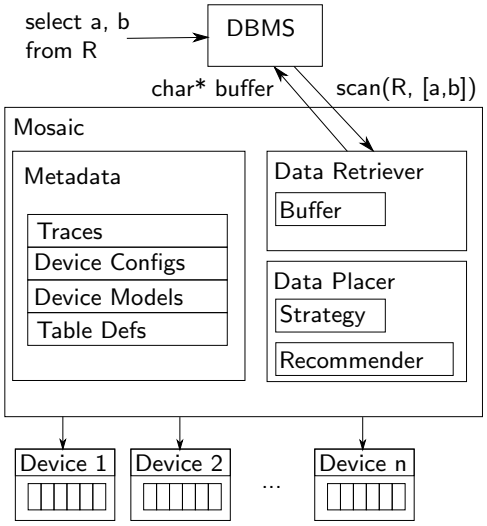


Figure 2: The components of Mosaic and its interface with the RDBMS.

Guerra et al. developed a general-purpose framework for dynamic tiering [11]. Like Mosaic, it has an advisor that gives purchase recommendations and a runtime component that retrieves data. In contrast to Mosaic, however, it operates on opaque data chunks instead of tables. While this approach is more generalized, it has the downside that it cannot make domain-specific optimizations, as explained above.

Wu et al. developed a general-purpose hybrid storage system with an approach similar to that of Mosaic [40]. It forgoes tiering and places data so that the bandwidth of all devices is fully utilized. Unlike Mosaic, however, it only supports a mixture of identical HDDs and SSDs (i.e. not multiple HDDs or SSDs at different speeds).

3. MOSAIC SYSTEM DESIGN

Mosaic comprises four components, as shown in Figure 2. Mosaic collects metadata from users before they purchase devices and while running as a storage engine. Information about attached devices, their measured performance, and recorded traces is kept in a metadata store (Section 3.1). Mosaic stores its managed data in a storage format that is optimized for the access characteristics of its devices (Section 3.2). The data retriever is an interface between the storage layer and the DBMS (Section 3.3). The data placement component distributes data between attached storage devices so that the data retriever can maximize the average throughput for a workload (Section 3.4). It is also responsible for predictions and purchase recommendations.

The components are interdependent: An inappropriate storage format (e.g. one that uses an excessively slow compression algorithm for a storage device with high throughput) reduces overall throughput, even if the data placer finds an optimal placement. The same goes for the data retriever: If data placement is not optimal or Mosaic chooses the wrong compression type for the data, performance will suffer, even if it can exploit the throughput of a storage device.

3.1 Metadata

Figure 3 summarizes the metadata gathered by Mosaic. The only information supplied by the user is a list of connected devices (3a). This contains each device's capacity,

```
[[device]]
    id = 0
    mnt = "/mnt/nvme"
    name = "NVMe SSD"
    capacity = 60 GB
    threads = 8
    compression = "ZSTD"
    cost_per_gb = 60 ct
```

(a) Device configuration entry

```
[[trace]]
    A:x,y
    B:z
    A:x
    C:u,v
    A:y
    C:v,w
    ...
```

(b) Excerpt of a trace

```
A:  x -> int,
    y -> varchar(200)
B:  z -> int
C:  u -> int64,
    v -> varchar(10),
    w -> int
```

(c) Table definitions

```
[[model]]
    device_id = 0
    none = 2.1 GB/s
    LZ4 = 1.6 GB/s
    zstd = 1.2 GB/s
    seek = 0
```

(d) Device model

Figure 3: Metadata recorded, maintained, and stored by Mosaic.

the optimum number of concurrent reader threads, cost[1], and the preferred compression algorithm. Since Mosaic does not depend on fixed device tiering, the user can add and remove devices to the pool during runtime by editing the device configuration file. Mosaic records table scans of queries being run since the last time the user triggered data placement in a trace file (3b). If the user has not yet purchased any storage devices, but wishes to receive a purchase recommendation, Mosaic can generate a trace file without a data set being present. Mosaic then extracts the table scans from each query and inserts them into the trace file. The trace file allows Mosaic to match the accessed data chunks to columns, and the columns to table scans, and shows which columns the DBMS has frequently accessed together. The data placer uses the trace file to infer the optimum data placement for the recorded workload. This is an advantage over established big data file systems. HDFS, for example, only keeps access statistics per file, with no insight into what a file consists of. Mosaic can derive data interdependencies from the trace file (i.e. it can determine which columns are often queried together).

Mosaic furthermore extracts table definitions from the data set (3c). It also periodically measures the throughput of all attached devices for the current workload and stores it in a device model file (3d). The predictor uses this file to predict how the data retriever would perform with hypothetical data placements.

3.2 Storage Format

We adapted Apache's established Parquet file format to form the Mosaic data storage format. It is a columnar data storage format, and it has many properties beneficial to Mosaic. (1) Parquet stores data in a column-major format with columns further subdivided into chunks comprising pages. Mosaic extricates column chunks out of existing Parquet files and distributes them between storage devices. (2) Parquet can compress pages individually with a variety of compression algorithms. Mosaic can thus compress column chunks depending on their storage device and recompress them with a different algorithm during migration. (3) Parquet's internal data format has built-in support for partitioning a relation on multiple files. We extend this so that Mosaic can place any column on any storage device. (4) Parquet stores one metadata block per column chunk and separates data and metadata. Mosaic can thus easily move them independently of each other. Instead of storing the column chunk metadata with the data itself, we reserve some storage space on a metadata device chosen by the user. This ensures that reading metadata does not affect concurrent reads on devices not suited to random reads (i.e. HDDs).

Mosaic allows the user to choose a compression algorithm for each device. Compressing the stored data has multiple advantages: (1) Mosaic can store more data while staying within budget, as compressed data takes up less space. (2) When the decompression throughput of the CPU is higher than a device's throughput, data compression increases the effective throughput. (3) When data on faster devices is compressed, Mosaic can move a greater percentage of the working set to those devices, thus increasing overall throughput. Column-major storage and compression make random accesses and updates more difficult. This, however, is no issue for Mosaic, as it focuses on scan-heavy workloads.

3.3 Data Retrieval

Mosaic's data retriever component retrieves stored data and converts it into a format that the RDBMS is able to read. The smallest unit of storage it can retrieve on request is a column chunk, which, by default, comprises 5 million values. At a higher level, the RDBMS can also request entire table scans. When the RDBMS triggers a table scan, Mosaic asynchronously fetches chunks of the requested columns in ascending order, until the buffer is full.

Mosaic can keep this buffer small: It assumes that queries are I/O-bound, and the RDBMS is thus limited by the speed at wich Mosaic fills the buffer. The buffer only holds the set of chunks that the RDBMS is actively processing and the set of chunks being concurrently prefetched. The size of the buffer thus depends on the number of columns being scanned and their data type. In our experiments, it never exceeded 1 GiB. Whenever the data retriever has buffered a set of chunks, it notifies the RDBMS of the new data via a callback. As soon as the RDBMS has processed a chunk, Mosaic evicts it from the buffer. Prefetching is straightforward, as Mosaic only needs to support linear table scans.

As the data placer can store columns of a table scan on different devices, Mosaic must read from multiple devices in parallel. Devices such as NVMe SSDs only reach their maximum throughput when multiple threads read concurrently. Mosaic's data retriever maintains a thread pool with reader threads. It assigns each requested column chunk to a reader thread. As most table scans access multiple columns, the data retriever reads the chunks of all requested columns in parallel. This is important, as the slowest reader determines overall throughput. The placement strategy must ensure that the chunks are placed in a way that maximizes the data retriever's throughput. The placer thus has to make sure that a column on a slow device does not stall a table scan whose other columns are on fast devices. Each reader forces the OS to sequentially populate its page cache with

[1]We use €-cents (ct) as the unit of currency as € is the currency in which we bought our evaluation system. If the absolute cost is unknown or subject to change, it is possible to define the cost relative to the cheapest device. For example, if an SSD costs three times as much as an HDD, enter 3 and 1 respectively.

the relevant data. Without this step, random access by the RDBMS or the decompressor could reduce the throughput.

While SSDs need concurrent access to maximize throughput, HDDs have a large drop in throughput if accessed concurrently, as sequential access will degenerate into random access when multiple threads contend for the device. Mosaic thus lets the user set a per-device thread limit. A semaphore guards each device to ensure that the number of threads reading from a device in parallel never exceeds the optimum.

Before returning to the *reader* thread pool, reader threads add the chunk's data, which the OS page cache now buffers, to a queue. This queue is ordered by the chunk request time. Mosaic now decompresses the queued chunks. Since the throughput of a storage device could be higher than that of a single thread that is decompressing data, Mosaic maintains a *decompression* thread pool. Whenever a decompression thread is idle, it fetches the first chunk in the queue, decompresses it, and makes the resulting values available to the RDBMS via its callback.

3.4 Data Placement

Mosaic places data offline and only reorders data when prompted to do so by the user. Mosaic starts a new trace file after each placement or on manual prompt by the user. It stores information about the columns accessed by the RDBMS in the trace file of that epoch. It enters a record for each table scan of every query executed. Each record stores the table, and the columns requested by the table scan.

As explained in Section 1, Mosaic's placement module has two modes. Before devices are purchased, Mosaic is in *budget* mode. When they are installed, Mosaic switches to *capacity* mode. In *budget* mode, Mosaic calculates a recommended placement on the basis of a given budget. In *capacity* mode, it distributes the data between the connected devices up to their capacity as specified in the device configuration metadata. When in *budget* mode, Mosaic considers all devices of the device configuration metadata as targets regardless of whether they are present. This mode does not restrict the device's capacities, but it does restrict their cost. Here, Mosaic's recommender provides the user with the recommended hypothetical placement along with a set of devices and their capacity for installation. Mosaic ensures that the total device cost does not exceed the user-defined maximum budget. In both modes, Mosaic uses swappable placement strategies to calculate a data placement. The next section summarizes the strategies employed.

4. DATA PLACEMENT STRATEGIES

For performance predictions, Mosaic not only needs to place data optimally, i.e., to find the best placement solution *qualitatively*, it also has to predict performance *quantitatively*. Mosaic consequently needs a model that can predict how data placement impacts query runtime (Section 4.1).

Mosaic supports pluggable data placement strategies (Section 4.2). The following three sections present three different placement strategies. The first two of these (Section 4.3 and Section 4.4) are used by multiple state-of-the-art systems. They were designed for a tiered storage engine, i.e., they assume that 'slow' and 'fast' layers exist, between which they can move the data. However, as Mosaic is a tierless engine, they are not a good fit. We therefore use them as a baseline against which we compare our contribution, which is the third strategy, called LOPT, and is explained in Section 4.5.

4.1 A Model for Predicting Table Scan Time

Since Mosaic not only offers data placement for installed devices but also predicts performance for hypothetical configurations, it needs a model on which to base its predictions. To keep complexity down, we make three assumptions:

Columns are atomic. We assume a column is stored contiguously on a single device. Mosaic can split columns at the parquet column chunk level and distribute the chunks on multiple devices. The prediction component, however, considers columns to be atomic. This speeds up placement calculation, as only whole columns have to be placed, which reduces the complexity of the model. It also has the added benefit that placement calculation is independent of data set size, as the number of columns and therefore possible placement permutations is constant in the number of tuples. Distributing chunks on multiple devices only benefits runtime performance if some chunks of a column are read disproportionally frequently and therefore profit from being on faster devices. This is only the case if data is either sorted (which is only possible for one column per table) or the query is so selective that chunks can be skipped. This is unrealistic with large chunk sizes. While possible with smaller chunk sizes, Mosaic cannot shrink chunks too far as the placement calculation would become too expensive.

Queries are I/O dominated. To keep the model agnostic of the query execution engine, we ignore computation times, such as aggregation, joins, or predicate evaluation and we only model table scans. Each query comprises one or more table scans, each of which reads one or more columns. Columns on different devices can — and should — be scanned in parallel. While this assumption might reduce absolute prediction accuracy for CPU-bound workloads, predictions will still be correct in relation to each other, as the computation overhead is constant. The overhead only depends on the contents and size of its tables, not on data placement and only adds a constant error to all predictions, assuming the computation overhead is not shadowed by I/O.

The throughput of a device is independent of the number of columns being read in parallel. We assume that Mosaic can saturate a device's I/O bandwidth regardless of how many columns it reads in parallel. This is true for SSDs, which benefit from multi-threaded reads. It is wrong for HDDs, whose throughput decreases when reading columns in parallel, because of their seek time. Since we solve this problem on the architecture side by reading columns a chunk at a time and using per-device semaphores (see Section 3.3) that ensure that only one thread at a time can read from a HDD, we need not model it.

The model we built is based on these assumptions. It predicts the total execution time t_{total} of a set of table scans TS given a set of devices D and a set of columns C.

For each column $c \in C$, the function $size$ returns its size:

$$size : C \to \mathbb{N} : \quad \text{size of column}$$

For uncompressed data, $size$ is the product of the number of tuples in the column and the size of the column's data type. For compressed data, Mosaic looks up its size in the metadata of each column chunk.

Each table scan $T \in TS$ is a subset of C, and each device $d \in D$ is modeled as a 5-tuple

$$d = \langle t_{seek}, cr, t, capacity, cost \rangle \tag{1}$$

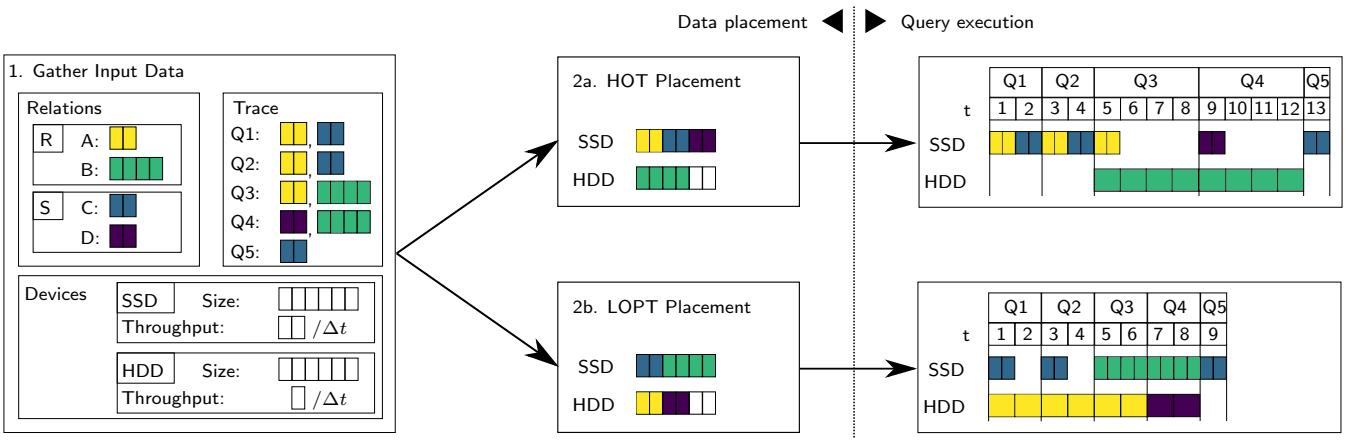


Figure 4: Modus operandi of Mosaic, and two exemplary placement strategies. The HOT algorithm indiscriminately moves the most frequently accessed columns to the SSD. The LOPT algorithm finds the data placement with the least storage device idle time and thus speeds up sequential execution of the 4 sample queries by 30%. For demonstration purposes, we assume that the SSD has twice the throughput of the HDD.

with the following values:

$$\begin{aligned} t_{seek}: &\quad \text{seek time} \\ cr: &\quad \text{compression ratio} \\ t: &\quad \text{throughput} \\ capacity: &\quad \text{capacity} \\ cost: &\quad \text{cost per unit of storage} \end{aligned}$$

These values are stored in the user-provided device configuration entry (see Section 3.1), with the exception of t, the continuously measured throughput stored in the device model metadata.

Equation (2) expresses the time $t_{d,c}$ required to scan a column $c \in C$ stored on a device $d \in D$:

$$t_{d,c} = t_{seek} + \frac{size(c)}{cr(d) \cdot t(d)} \tag{2}$$

The fraction $\frac{size(c)}{cr(d)}$ is an estimation of the compressed size of c on d. If the column has already been stored on d or another device with the same compression algorithm, Mosaic looks up the actual size instead of estimating it.

When the placer stores two or more columns relevant to a table scan on different devices, the retriever can read them in parallel. The runtime of each table scan $T \in TS$, t_T is thus only determined by the device taking the longest, as seen in Equation (3).

$$t_T = \max\{\sum_{c \in T} I_{d,c} \cdot t_{d,c} \mid d \in D\} \tag{3}$$

I is an indicator function:

$$I_{d,c} = \begin{cases} 1 & \text{if column } c \text{ is stored on device } d \\ 0 & \text{otherwise} \end{cases}$$

The total time required to run the set of table scans TS is the sum of the runtime of each table scan:

$$t_{\text{total}} = \sum_{T \in TS} t_T \tag{4}$$

The model allows the approximate cost of a real or hypothetical data placement to be calculated:

$$cost_{\text{total}} = \sum_{c \in C} \sum_{d \in D} I_{d,c} \cdot cost(d) \cdot \frac{size(c)}{cr(d)} \tag{5}$$

Mosaic's data placer, given I, moves all columns to the device specified by I.

I is an abstraction over specific placement strategies and their implementations. A strategy can either determine I algorithmically (Sections 4.3 and 4.4) or with a constraint solver (Section 4.5). Mosaic's prediction and placement component is therefore independent of the placement algorithm.

4.2 Responsibilities of a Strategy

As seen in Figure 4, the data placer supplies each strategy with a number of inputs. These are (1) the size of each relation's columns, (2) the throughput, size, price per gigabyte, and optimal number of parallel readers for each attached device, and (3) a trace with the table scans since the start of the current epoch.

A strategy places columns on storage devices in such a way that the average throughput of a workload similar to the trace is maximized. Figure 4 shows two such strategies. Strategy (a), called HOT, places the columns read the most often (i.e., the 'hottest' columns) on faster devices. Strategy (b), called LOPT, finds the optimum placement using linear optimization. As can be seen on the right-hand side, the choice of strategy impacts the overall throughput. The HOT strategy cannot exploit the fact that Mosaic reads data from multiple devices in parallel. The LOPT strategy, in contrast, uses both devices, concurrently decreasing the overall table scan time in the example by $\approx 30\%$.

4.3 HOT Strategy at Table Granularity

The table granular HOT strategy (HOT table) treats each table as an atomic entity that can only ever live on one single device at a time. The strategy places tables according to their 'hotness'. It assumes that a table that the RDBMS scans often (being 'hot') benefits from being on a fast device. Improving a table scan that runs more often has an overall

higher positive impact on average throughput. It places tables descending in order of their number of accesses on the fastest device with enough space for the whole table.

This strategy is an approximation of the toolset available to administrators of many established RDBMS such as PostgreSQL or Oracle. These systems allow database administrators to create different tablespaces on different devices and assign each table to a specific tablespace. While these systems do not allow automatic data placement like Mosaic does, we assume that a system administrator using tablespaces will decide in the same way as the HOT strategy: they will move tables appearing disproportionally often in observed queries to faster devices.

4.4 HOT Strategy at Column Granularity

The column-granular HOT strategy (HOT column) is an improvement over the table granular version. As before, data accessed more frequently is considered 'hot' and so is placed on devices with higher throughput. But this time, tables are no longer treated as atomic. Instead, HOT column migrates single columns of tables. This is a huge improvement over HOT table, as even the hottest tables often have multiple columns that are only rarely queried. HOT column will rightfully prioritize warmer columns of cold tables over cold columns of hot tables.

While the HOT approach has been proven to be workable by many existing tiered storage engines, it has multiple weaknesses. For instance, (1) HOT relies on a tiered architecture in which data is moved up or down one tier at a time. With HOT, Mosaic can emulate such a hierarchy with two or three devices that have large performance gaps (say, an HDD and an SSD). If we, however, add multiple devices whose throughputs are close (i.e. multiple HDDs, or a SATA SSD and a RAID 5 of multiple HDDs) the HOT strategy can no longer cleanly bin those devices into distinct tiers. (2) As can be seen in Figure 4, HOT does not place data such that a table scan can be parallelized. If the RDBMS often scans two hot columns together, they would benefit from being on different devices so that Mosaic could read from both devices in parallel. HOT would try to place both on the fastest device available, leaving optimization potential on the table. (3) Mosaic can only apply the HOT placement strategy if it knows the device capacities beforehand. If Mosaic is in *budget* mode, it is not obvious how to choose device sizes to maximize throughput.

4.5 Linear Optimization Strategy

Rather than using a heuristic to place data, the linear optimization strategy (LOPT) uses the model defined in Section 4.1 to find an optimal solution. LOPT deems a solution optimal if it minimizes the time spent scanning tables for a set of queries. It uses a constraint solver to define the indicator function I in such a way that t_{total} of Equation (4) is minimized.

LOPT subjects Equation (4) to the following constraints for each column:

$$\forall c \in C : \sum_{d \in D} I_{d,c} = 1 \tag{6a}$$

$$\forall c \in C : \forall d \in D : I_{d,c} \in \{0, 1\} \tag{6b}$$

A column has to be stored exactly once (6a) and is completely stored on a device or not at all (6b). LOPT enforces one of two additional constraints, depending on the mode.

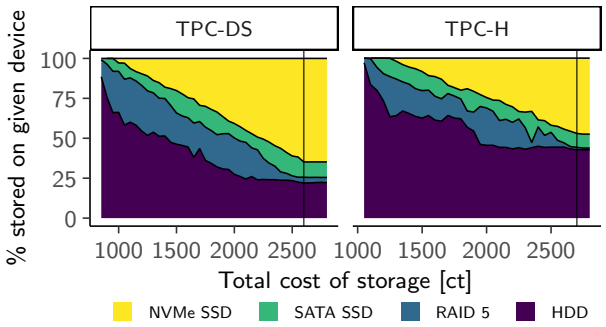


Figure 5: LOPT data placement in *budget* mode for TPC-H and TPC-DS (SF 100) at different budgets. The vertical lines indicate from when an increased budget does not increase performance.

In *capacity* mode, the strategy infers optimal placement for previously purchased hardware. A valid placement must therefore not exceed the storage capacity of any installed device. Mosaic thus subjects Equation (4) to the following additional constraint for each device:

$$\forall d \in D : \left(\sum_{c \in C} I_{d,c} \cdot \frac{size(c)}{cr(d)} \right) \leq capacity(d) \tag{7}$$

In *budget* mode, the strategy predicts the optimum placement for a budget $cost_{max}$. Since no hardware has been bought yet, Mosaic can ignore all the capacity limitations but has to stay below budget. Mosaic subjects Equation (4) to the following additional constraint:

$$\left(\sum_{d \in D} \sum_{c \in C} I_{d,c} \cdot cost(d) \cdot \frac{size(c)}{cr(d)} \right) \leq cost_{max} \tag{8}$$

Mosaic uses Gurobi [12] to solve this optimization problem. Gurobi is a constraint solver with support for mixed-integer programming (MIP).

LOPT strategies' advantage over HOT variants is that it exploits all the information encoded into the model. (1) As Figure 4 shows, HOT 'leaves bandwidth on the table'. It underutilizes slower devices, which — while having less throughput than their faster counterparts — could still contribute to overall throughput. This is because HOT tries to concentrate hot data on a few, fast devices. LOPT is free to place hot data on slower devices if a larger column is the bottleneck of the table scan. (2) LOPT is aware that it is optimizing table scan performance and makes domain-specific optimizations through its modeling. It does not waste precious space on faster storage devices for columns that are hot but are often queried together with colder columns. (3) The user can easily extend LOPT. A user might, for example, want to model a limited amount of expansion slots, a maximum/minimum size of each storage device, or a power budget constraint. With LOPT, they can just add new dimensions to the device model and add additional constraints for those dimensions to the solver. The solver will then find the best solution given the additional constraints. No further changes to Mosaic are needed.

Figure 5 shows the advantages of LOPT and its *budget* mode for an exemplary storage configuration. It comprises a fast NVMe SSD, a slower SATA SSD, an even slower HDD,

Table 1: Storage devices of the evaluation system.

Device	Price per GB	Throughput
NVMe PCIe SSD	125 ct	2.10 GB/s
SATA SSD	60 ct	0.41 GB/s
RAID 5 of HDDs	45 ct	0.32 GB/s
HDD	30 ct	0.23 GB/s

and a RAID 5 of three HDDs. At lower budgets, LOPT in *budget* mode does not spend all the available money on a fast NVMe drive. It instead distributes data between the four devices, maximizing overall throughput. Only with an increasing budget does LOPT gradually place data on the fast NVMe SSD. Even at high budgets, it still keeps parts of the data on SATA SSD. To save costs, it keeps never-touched data (25% for TPC-DS, 50% for TPC-H) on HDD. LOPT can thus determine when adding additional hardware is just a waste of money. In the figure, this threshold is marked by a vertical line.

While LOPT is more sophisticated than HOT, it is also much harder to compute. Constraint (6b) that permits only integers is particularly constricting, as it forces us to employ MIP, which is NP-hard. But it is important to note that run time only depends on the device count and the number of distinct table scans. It is independent of the number of tuples (as we treat columns as atomic units) and queries. If multiple queries 're-use' the same table scans or the user runs a query multiple times, the model does not become more complex. LOPT just multiplies its modeled runtime for that query by the number of reuses, and the optimizer does not need to consider more variables. Section 5.5 evaluates placement computation cost in detail.

5. EVALUATION

Table 1 shows the storage configuration of the evaluation system. The system comprises four different storage device classes, each competitive at its respective price point. Besides two SSDs of different speeds, we equip the server with four enterprise grade server HDDs at 10k RPM. We configure three HDDs as a RAID 5 and keep the fourth as a standalone disk. The server is equipped with 192 GB of DRAM and a single socket Intel Xeon Gold 6212U CPU with 24 physical cores @ 2.4 GHz (with SMT: 48 cores).

As explained in Section 3, choosing a fitting compression algorithm for each storage device increases throughput. While using no compression incurs no added CPU overhead, it requires the most space. LZ4 has a low CPU overhead with an acceptable compression ratio. Zstandard (ZSTD) has the highest compression ratio with a still acceptable CPU overhead. As expected, the synthetic TPC-H SF30 data set compresses quite well, requiring 44.11 GB uncompressed, 16.51 GB if compressed with LZ4, and only 10.03 GB with ZSTD. ZSTD still yields a compression ratio of about 3 on real-world data sets (2 for LZ4) [1]. Table 2 shows the relative speedup of the TPC-H benchmark over the baseline for different compression algorithms. ZSTD compressed data takes up less space and increases overall performance compared to the cheaper LZ4 algorithm, even on PCIe SSD. For this setup, we therefore configure Mosaic to always compress data with ZSTD.

We run all benchmarks with Umbra as the database engine and Mosaic as its storage engine. We choose Umbra as it provides best-of-class speed and thus rules out CPU bottlenecks, unlike big data query engines. While RDBMS like MySQL also expose an interface for storage engines, they cannot easily be adapted to columnar data storage. A more detailed reasoning as to why we evaluate Mosaic only in conjunction with Umbra can be found in Section 5.9.

Table 2: TPC-H benchmark speedup (SF 30) of SSD and HDD for different compression algorithms.

	Speedup over HDD		
Device	None	LZ4	ZSTD
HDD	1	—	2.92
NVMe PCIe SSD	6.2	11.18	12.66

5.1 Benchmarks

For our evaluation, we use two OLAP benchmarks: TPC-H and TPC-DS. TPC-H comprises 22 queries and 8 tables. The largest table, *lineitem*, accounts for 70% of the data set size, while the smallest 5 tables together only make up 3%. Choosing the best placement for the columns of the *lineitem* table thus gives Mosaic a large optimization potential. TPC-DS is a much more complex OLAP benchmark. It comprises 99 queries and 24 tables. Since Umbra does not yet support all features required by TCP-DS, such as window functions, we discard unsupported queries. We thus run a subset of TPC-DS comprising 67 queries. We run both benchmarks at scale factor 30 and 100.

For both benchmarks, we define one run as a measurement of the runtime of each query executed once sequentially, with the execution times then added. To accurately measure a query's runtime, we execute it five times and take the mean. Before each query, we clear the OS cache to force Mosaic to read all data from the underlying storage devices. Running all queries sequentially just once is not a realistic benchmark. In reality, a workload is usually heavily skewed towards just a few queries. It is, however, the worst case for Mosaic and thus a good benchmark. The more distinct queries we run, the harder it is for Mosaic to find an optimal placement. The working set is also larger. Mosaic thus benefits less from expensive storage on faster devices.

5.2 Mosaic vs. Traditional RDBMS

In this section, we evaluate how Mosaic compares against the toolkit of a traditional relational database system. We compare Mosaic's column-granular LOPT placement strategy against table-granular placement. Table-granular placement is the status quo and the best option in an RDBMS such as Oracle or PostgreSQL.

We first import a trace of the TPC-H benchmark (executing queries 1 to 22 once in sequence). We then trigger Mosaic's LOPT placement strategy for different budgets. After data placement, we repeat the benchmark and record the runtime. As a baseline, we benchmark all table placement permutations for the four largest TPC-H tables that make up 98% of the total data set size. The remaining four smallest tables are always stored on NVMe SSD.

Figure 6 shows all unique table-granular placement configurations (▲) for HDD, SATA SSD, and NVMe SSD. Each configuration could have been chosen by a system administrator of a traditional RDBMS with tablespaces. We mark the three configurations in which Mosaic stores all five tables

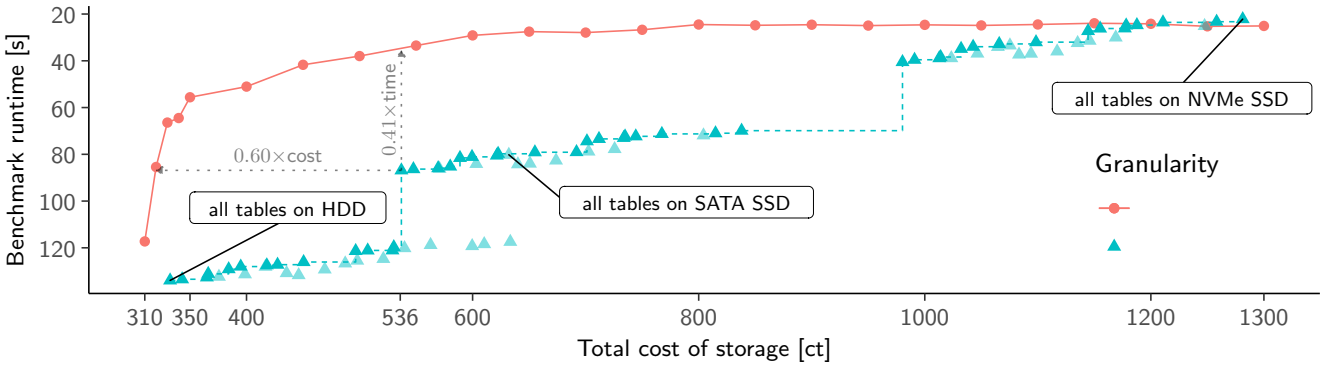


Figure 6: Benchmark runtime for TPC-H (SF 30) with column-granular placement using the LOPT strategy compared to all placement permutations of the four largest tables at table granularity. The dashed line indicates the Pareto optimum for table placement. The dotted arrows show that Mosaic using LOPT placement offers the same performance at a lower budget or faster runtime at the same budget.

on the same device. The three distinct clusters correspond to the storage location of the *lineitem* table. At 6.8 GB, it contributes 70% of the total data set size, and its placement thus has the greatest effect on the total cost of storage.

The Pareto-optimal line (---) shows the best case for table-granular placement, i.e., there is no cheaper placement that also reduces benchmark runtime. A system administrator can, therefore, hope at best to hit this line. For most budgets, Mosaic's LOPT placement strategy (—●—) dominates and offers the choice of having the same performance at less cost or more performance at the same cost. As indicated in the figure, at a budget of 536 ct, LOPT offers the same throughput as the Pareto-optimal table placement at 60% of the cost, or 41% of the runtime at the same budget. Table-granular data placement is only competitive if Mosaic places all data on the cheapest or most expensive devices.[2]

This result also shows that when Mosaic just stores a small part of the working set on fast storage, this already drastically increases overall throughput. The cost of this increase is very low if placing data at column granularity. A budget increase of 12% (from 310 ct to 350 ct) speeds up the benchmark by over 100% (from 117 s to 55.6 s). At higher costs, where most data fits on the fastest device, Mosaic cannot gain much advantage from distributing data between devices (as seen in Figure 5). It thus has equal or — if the model's throughput estimates are inaccurate — slightly worse performance than if the user placed all data on the fastest device.

Mosaic also visualizes a law of diminishing returns. With a budget of 600 ct, Mosaic is already within 14% of the best performance that requires twice the budget, i.e., 1300 ct. The optimal table granular placement at 600 ct results in a benchmark that takes 3.7 times as long as at maximum budget.

5.3 Comparison of Placement Strategies

In this experiment, we compare Mosaic's three placement strategies, LOPT, HOT table, and HOT column, against each other. How much the placement strategies differ in performance depends on the storage configuration. If, for example, only one storage device is available, all strategies place data identically. To obtain a representative comparison of the strategies, we compare performance across a range of device configurations. For each of the four devices in Table 1, we fix its proportion of the total storage to a value between 0% and 100% of the data set size, in 20% steps. We then recursively fix the values of the remaining three devices in the same way. We only consider configurations whose storage adds up to 100% of the data set size. Mosaic thus runs the benchmark for 56 configurations for each strategy. We then add the runtimes of those benchmarks.

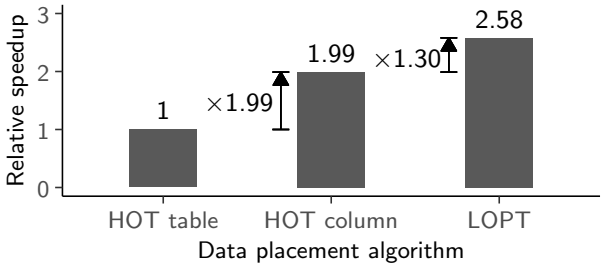
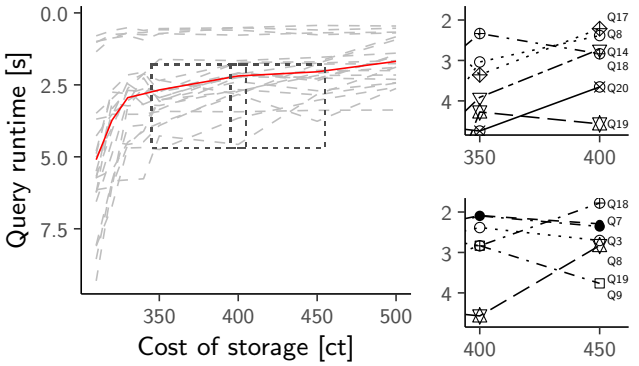


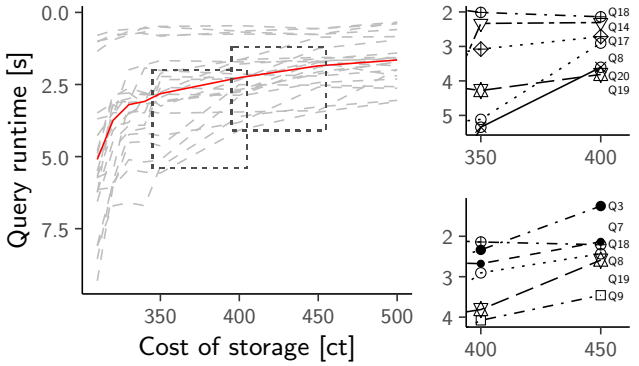
Figure 7: Comparison of placement algorithms normalized to HOT table. Each bar is the sum of 56 runs of the TPC-H benchmark (SF 30). Each run uses a distinct device configuration.

Figure 7 shows the results for the TPC-H benchmark. It shows the speedup of the placement strategies over the baseline, HOT table. HOT table is worse than the other two strategies, as the TPC-H data set has many large but cold columns on otherwise hot tables. At table granularity, these columns waste valuable storage space that hot columns of different tables could have used. Data placement at column granularity provides a 99% speedup, confirming our findings in Section 5.2. LOPT is ≈ 25% faster still than HOT column, showing the advantage of a tierless device pool over a tiered architecture even with just four devices. Because throughput gaps between SATA SSD, RAID 5, and HDD are small, LOPT can distribute columns often accessed together between those devices. HOT column places as much

[2]To keep the number of variants for table granularity measurements manageable, the four smallest tables always reside on NVMe SSD. The cheapest measurement at column granularity is therefore cheaper than the cheapest measurement at table granularity.



(a) Default LOPT. The zoomed-in sections show the biggest winner and loser queries at budgets of 400 and 450 cents.



(b) Modified LOPT. Placement is constrained so that no query may become slower. The zoomed-in sections show the same queries as (a).

Figure 8: Runtime per query for two different LOPT variants (TPC-H SF 30). The solid red line shows average runtime, the dashed lines show runtime of each of TPC-H's 22 queries.

data as possible on SATA SSD, preferring it over HDD and RAID 5, leaving optimization potential on the table. We, therefore, chose LOPT as Mosaic's default strategy.

5.4 Per-Query Analysis of LOPT

While average query performance increases monotonically with budget, there are 'loser queries' that either do not become faster or even degrade with increasing budget, since LOPT's only goal is to minimize the sum of all query runtimes. Figure 8a shows per-query performance at varying budgets. The two zoomed-in sections show the biggest 'winner' and 'loser' queries at 400 and 450 cents.

At 400 ct (upper cutout), Q18 and Q19 are slower than at 350 ct, as LOPT moves the columns of `lineitem` read by both queries from RAID back to HDD. This makes space for four columns read by the other queries, reducing overall runtime. When the budget increases to 500 ct (lower cutout), the pattern reverses: LOPT moves `lineitem`'s primary key back to RAID from SATA SSD. This slightly slows down most queries reading it but allows LOPT to move Q18's and Q19's previously demoted columns back to SATA SSD.

The user may deem such regressions unacceptable, i.e., they require guarantees that some specific subset — or all queries — do not slow down after a system upgrade. In this case, Mosaic supports the addition of a new constraint to LOPT, setting a query's (or all queries') current execution time as an upper bound. Figure 8b shows LOPT's performance with this constraint. While the throughput is 10.1% worse on average, there are no more unpredictable performance regressions.

5.5 Placement Calculation Cost of LOPT

As stated in Section 4.5, LOPT is NP-hard. Heuristics of modern MIP solvers, however, keep computation time at a reasonable level even for larger problems. We first evaluate LOPT's placement calculation time for smaller sized workloads. The results are shown in Table 3. The JOB workload by Leis et al. [24] benchmarks cardinality estimators with queries that join many tables. Many of its table scans only touch primary and foreign keys. It thus has only a few more *distinct* table scans than TPC-H. In both cases, LOPT finds the optimal solution effectively instantaneous for arbitrary data set sizes. TPC-DS has over 3 times as many distinct table scans as TPC-H. LOPT's performance with TPC-DS

Table 3: LOPT search time for a placement solution for four devices with three different workloads. It shows the time to find a solution that is within 5% or 1% of the theoretical optimum, or is optimal.

		table scans		time [s]		
	queries	total	dist.	$< 5\%$	$< 1\%$	opt
TPC-H	22	86	58	< 1	< 1	< 1
JOB	113	977	62	< 1	< 1	< 1
TPC-DS	67	492	193	< 1	≈ 4	≈ 64

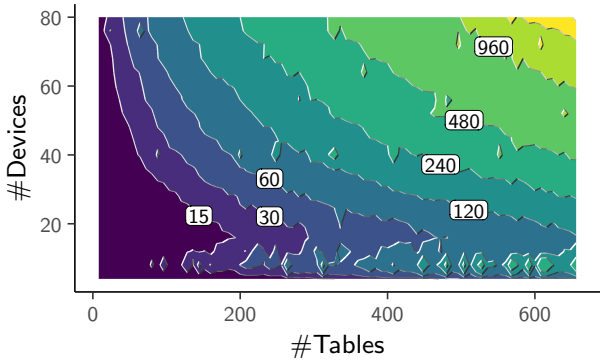


Figure 9: Computation time in seconds for a solution within 5% of the lower bound. The z-axis is $\log_2$ scale, i.e., time doubles with each contour step.

is acceptable, but at ≈ 1 minute for the optimal solution it is considerably worse.

We now move on to progressively larger workloads, to see how LOPT scales with more devices, tables, and queries. We load multiple independent instances of TPC-H, multiplying the number of tables and queries by up to 80 times (resulting in up to 1760 queries on 640 tables with 4880 columns) and simulate each device up to 20 times (up to 80 in total). Note that this is an adverse workload, since as each column is accessed by 22 queries at most, there is no fast way for Gurobi to prune the solution space, i.e., all TPC-H instances are 'warm'. Figure 9 shows how long Mosaic takes to calculate a placement within 5% of the theoretical lower bound for all permutations. The worst case is ≈ 52 minutes

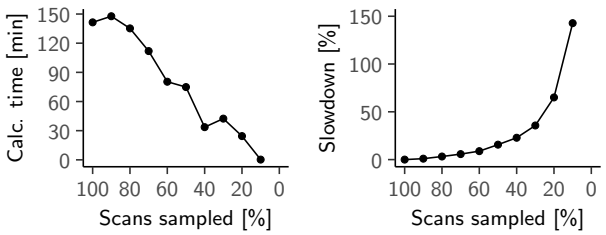


Figure 10: Left: Impact of sampling on placement calculation time. Right: Impact of sampling on predicted runtime performance. 400 TPC-H SF 30 instances, 8 devices.

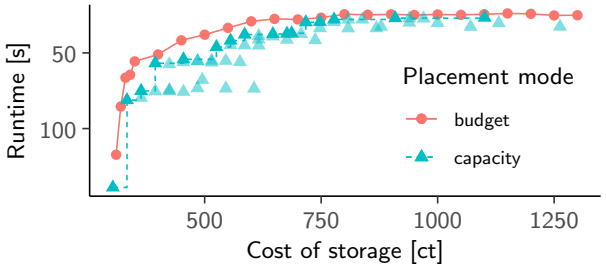


Figure 11: Comparison of placement modes for the TPC-H benchmark (SF 30) using LOPT. In budget mode, Mosaic chooses its storage devices for a budget. In capacity mode, Mosaic places data on 56 predefined device configurations.

for 80 devices and 640 tables. Cases that are realistic for a single node (i.e. $\leq$ 10 devices) take less than 8 minutes.

Being NP-hard, LOPT has its limits. With 1200 TPC-H instances (26400 queries, 10800 tables, 73200 columns) on eight devices, computing a solution within 5% of the lower bound takes $\approx$ 21.5 hours. This can be remedied with sampling, i.e., having LOPT only consider a subset of all table scans. Figure 10 shows the impact of sampling with 400 TPC-H instances (8800 queries, 3200 tables, 24400 columns) on eight devices. Since all columns are always 'warm' in this adverse workload, each discarded table scan removes valuable information. Even here, sampling is still beneficial. If a predicted slowdown of $\approx$ 9% is acceptable, it is possible to sample 60% of the table scans, thus reducing the placement calculation time by 44%, from 141 to 80 minutes.

5.6 Capacity Mode vs. Budget Mode

Figure 11 compares Mosaic's *capacity* mode (-▲-) against its *budget* mode (—●—). For *budget* mode, we repeat the measurement of Section 5.2. For *capacity* mode, we use the method of the experiment in Section 5.3 to generate 56 device configurations and use the LOPT strategy for placement. Each configuration (▲) could have been chosen by a system administrator using educated guesses. Because Mosaic uses the LOPT strategy for both placement modes, we can now quantify the advantage of having Mosaic assisting in the purchase decision (—●—) over pre-purchasing hardware and only then letting Mosaic place data (▲).

16 out of 56 capacity configurations are Pareto-optimal (- - -). For TPC-H, there is a probability of $\approx$ 29% of a system administrator picking a desirable storage device configuration by guessing which could be the best. But even if they pick a Pareto-optimal configuration, its corresponding

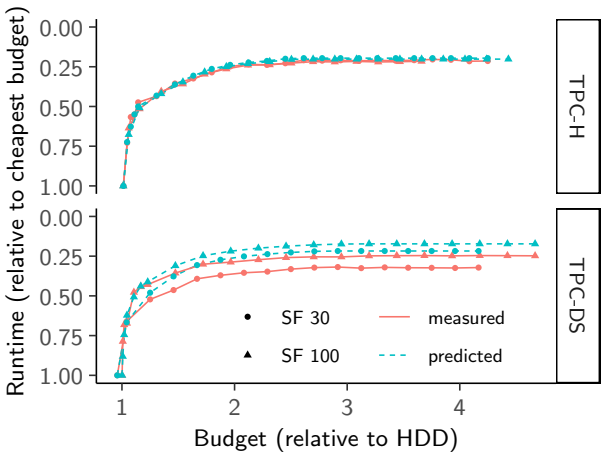


Figure 12: Predicted vs. actual performance for the TPC-H and TPC-DS benchmarks (SF 30 and 100).

budget counterpart dominates it. On a price-point-per-price-point comparison, the *budget* approach is $\approx$ 26% faster than the Pareto optimum of the *capacity* mode.

5.7 Prediction Accuracy

In this section, we evaluate whether predictions made by Mosaic's table scan model are accurate. For this benchmark, we use the LOPT placement strategy in budget mode. Mosaic predicts the runtime for a range of maximum budgets, both for TPC-H and TPC-DS, at scale factors of 30 and 100. It then places data according to the budget constraint and runs the benchmark. We then compare Mosaic's predicted benchmark runtime with the actual runtime.

Figure 12 shows the predicted runtime for a budget (- - -) and the measured time after Mosaic placed the data (—). For TPC-H, the absolute mean error between predicted and measured time across all scale factors and budgets is only 4.1%. For TPC-DS, it is 19.0%, with higher budgets having a higher error than lower budgets. The reason is that TPC-DS, is CPU-bound on the evaluation system, when Mosaic stores most of the data on the NVMe SSD. At lower budgets, the slower but cheaper devices hide the CPU overhead.

While the prediction is accurate when running an I/O-dominated workload or using slow devices, the prediction becomes inaccurate when the workload becomes CPU-bound. This is because Mosaic cannot predict the throughput of the DBMS's execution engine. While slower devices shadow the execution overhead, faster devices expose it. The experiment, however, shows that Mosaic is useful, even in CPU-heavy workloads for the following reasons: (1) $\approx$ 20% error is still acceptable when the status quo is having no prediction; (2) Mosaic brings the most benefit when users have limited budget and thus having most of the data on fast devices is not an option. Here, Mosaic is quite accurate, even for TPC-DS; (3) Mosaic correctly predicts the shape of the graph, showing where a small investment makes a huge return and when diminishing returns kick in. Mosaic's purchase recommendations are still valid, and it finds the fastest configuration for the given cost. It just does not take the bottleneck of the execution engine into account. We thus argue that even for CPU-bound benchmarks, Mosaic still offers great benefits over storage engines without predictive capabilities.

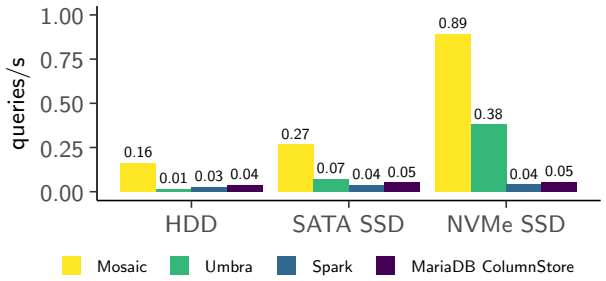


Figure 14: Mosaic's TPC-H throughput (SF 30) compared to Umbra and two Big Data query engines, all 22 queries distributed uniformly.

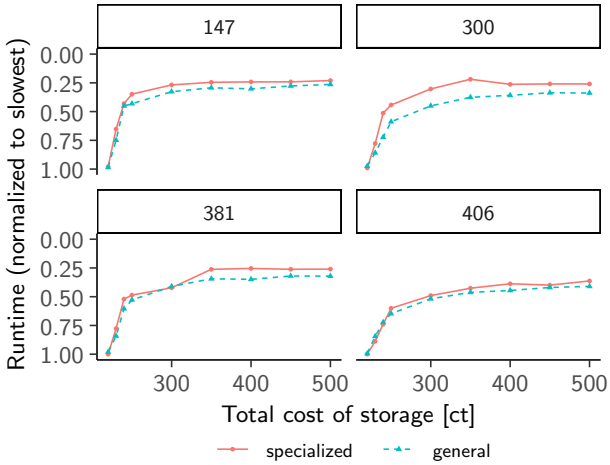


Figure 13: Performance of 4 out of 1000 TPC-DS workloads at different budgets for data placed specifically for the workload and for data placed for the TPC-DS benchmark in general.

5.8 Impact of Workload

To evaluate how Mosaic adapts to different workloads, we generate 1000 workloads with 10 random TPC-DS queries each. We pick four of those workloads that deviate the most from the shape in Figure 6 and compare their performance at different budgets. We have chosen them for a number of characteristics, for instance workload 147 profits above average at low budgets while workloads 300 and 406 profit at higher budgets. Workload 381 has a big performance jump at medium budgets. The performance of all 1000 workloads increases by more than 100% at 500 ct. Figure 13 shows Mosaic's performance for the four chosen workloads with data placed specifically for the workload (—●—) and data placed for the original TPC-DS workload (- -▲- -), which is a superset of the four workloads.

The workloads profit from a placement specifically tailored to them. Since the working set is smaller, Mosaic can move a larger percentage to devices with higher throughput. Each workload, however, also sees improvements with the generic TPC-DS placement. This experiment shows that — while it is beneficial to give Mosaic a trace that represents the actual workload as closely as possible — performance is still acceptable if the trace is a superset. Our earlier evaluations show that Mosaic finds a placement quickly even for large traces. A superset can be chosen (e.g. all queries run in the last month) without hurting performance too much.

5.9 Mosaic vs. Big Data Query Engines

In this section, we compare the performance of Mosaic against Spark and MariaDB ColumnStore as representatives of big data query engines. These OLAP systems are optimized to read data in column-major format. Both claim to be competitive on a single node. We also compare Mosaic against vanilla Umbra as a representative of conventional RDBMS. Umbra buffers data into main memory when first accessed. Consequently, Umbra is an order of magnitude faster when data is already buffered. To benchmark I/O speed, we clear Umbra's buffer between queries.

Figure 14 shows the throughput for TPC-H. For all three configurations, we store the data set on just one device. Umbra is optimized for in-memory data sets and SSD. Its performance degrades on devices not suited for random I/O, but it is slower than Mosaic even on NVMe SSD, as Mosaic's compression results in a higher effective throughput. Umbra's table scans furthermore read all columns while Mosaic only reads queried columns. Spark and MariaDB ColumnStore are slower by an order of magnitude. While Umbra and Mosaic speed up when moving the data set to an NVMe SSD, Spark and MariaDB only become marginally faster.

When reading from disk, Spark has a similar throughput to Umbra with Mosaic. It is optimized for distributed workloads and introduces abstraction layers required to make it compatible to its many supported file formats. This, however, results in the computation time shadowing the I/O time when running on a single node. Query 7, for example, takes 42 seconds at SF 30, even with all data on NVMe SSD. Even if we ignore four seconds of startup time, Umbra with Mosaic is ≈ 30 times faster at 1.2 seconds. At SF 100, it is still 10 times faster than Spark at SF 30. Spark spends more time on garbage collection (≈ 2 seconds) than Umbra takes for the whole query. On a single node, there is therefore not much to be gained by integrating Mosaic's smart data placement into big data query engines. Mosaic therefore has an important use case for single node systems with big data sets.

6. CONCLUSION

We present Mosaic, a storage engine optimized for scan-heavy workloads on RDBMS. It manages columnar data in a tierless device pool and supports pluggable data placement strategies. We evaluate three such strategies, including our linear programming placement strategy (LOPT), based on a model for predicting the throughput of table scans. In *capacity* mode, LOPT places data on previously purchased devices. In *budget* mode, LOPT predicts performance for a budget and makes purchase recommendations.

We evaluate Mosaic on two data sets to show the advantage of Mosaic's column-granular data placement over existing approaches of RDBMS and big data query engines. Mosaic outperforms them by an order of magnitude and beats Umbra in OLAP queries when the working set does not fit into DRAM. We show the accuracy of Mosaic's prediction, which closely follows the Pareto-optimal price/performance curve. It is accurate for I/O-bound benchmarks.

We recognize that it is difficult to confirm the results of this paper without reimplementing Mosaic. We therefore plan to open-source Mosaic so that our findings can be verified with third-party RDBMS.

7. REFERENCES

[1] https://facebook.github.io/zstd/, 2020. [accessed February 27, 2020].

[2] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum. Cache, cache everywhere, flushing all hits down the sink: On exclusivity in multilevel, hybrid caches. In *MSST*, pages 1–14. IEEE, 2013.

[3] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. MaSM: efficient online updates in data warehouses. In *SIGMOD*, pages 865–876. ACM, 2011.

[4] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD bufferpool extensions for database systems. *PVLDB*, 3(2):1435–1446, 2010.

[5] Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt. CAST: tiering storage for data analytics in the cloud. In *HPDC*, pages 45–56. ACM, 2015.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[7] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013.

[8] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*, pages 1243–1254. ACM, 2013.

[9] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging DBMS buffer pool using SSDs. In *SIGMOD*, pages 1113–1124. ACM, 2011.

[10] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[11] J. Guerra, H. Pucha, J. S. Glider, W. Belluomini, and R. Rangaswami. Cost effective storage using extent based dynamic tiering. In *FAST*, pages 273–286. USENIX, 2011.

[12] Gurobi Optimization LLC. Gurobi optimizer reference manual, 2019.

[13] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011.

[14] H. Herodotou and E. Kakoulli. Automating distributed tiered storage management in cluster computing. *PVLDB*, 13(1):43–56, 2019.

[15] S. Huang, Q. Wei, D. Feng, J. Chen, and C. Chen. Improving flash-based disk cache with lazy adaptive replacement. *TOS*, 12(2):8:1–8:24, 2016.

[16] I. Iliadis, J. Jelitto, Y. Kim, S. Sarafijanovic, and V. Venkatesan. ExaPlan: Efficient queueing-based data placement, provisioning, and load balancing for large tiered storage systems. *TOS*, 13(2):17:1–17:41, 2017.

[17] N. S. Islam, X. Lu, M. Wasi-ur-Rahman, D. Shankar, and D. K. Panda. Triple-h: A hybrid approach to accelerate HDFS on HPC clusters with heterogeneous storage architecture. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 101–110. IEEE, 2015.

[18] T. Ivanov and M. Pergolesi. The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and parquet. *Concurrency and Computation: Practice and Experience*, 32(5), 2020.

[19] Z. Jiang, Y. Zhang, J. Wang, and C. Xing. A cost-aware buffer management policy for flash-based storage devices. In *DASFAA*, volume 9049 of *Lecture Notes in Computer Science*, pages 175–190. Springer, 2015.

[20] P. Jin, Y. Ou, T. Härder, and Z. Li. AD-LRU: an efficient buffer replacement algorithm for flash-based databases. *Data Knowl. Eng.*, 72:83–102, 2012.

[21] E. Kakoulli and H. Herodotou. OctopusFS: A distributed file system with tiered storage management. In *SIGMOD*, pages 65–78. ACM, 2017.

[22] W. Kang, S. Lee, and B. Moon. Flash-based extended cache for higher throughput and faster recovery. *PVLDB*, 5(11):1615–1626, 2012.

[23] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206. IEEE, 2011.

[24] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[25] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. LeanStore: In-memory data management beyond main memory. In *ICDE*, pages 185–196. IEEE, 2018.

[26] X. Liu and K. Salem. Hybrid storage management for database systems. *PVLDB*, 6(8):541–552, 2013.

[27] T. Luo, R. Lee, M. P. Mesnier, F. Chen, and X. Zhang. hStorage-DB: Heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *PVLDB*, 5(10):1076–1087, 2012.

[28] T. Neumann and M. J. Freitag. Umbra: A disk-based system with in-memory performance. In *CIDR*. www.cidrdb.org, 2020.

[29] K. Oe and K. Okamura. A hybrid storage system composed of on-the-fly automated storage tiering (OTF-AST) and caching. In *CANDAR*, pages 406–411. IEEE, 2014.

[30] K. K. R., A. Anwar, and A. R. Butt. hatS: A heterogeneity-aware tiered storage for hadoop. In *CCGRID*, pages 502–511. IEEE, 2014.

[31] K. K. R., M. S. Iqbal, and A. R. Butt. VENU: orchestrating SSDs in hadoop storage. In *BigData*, pages 207–212. IEEE, 2014.

[32] K. K. R., B. Wadhwa, M. S. Iqbal, M. M. Rafique, and A. R. Butt. On efficient hierarchical storage for big data processing. In *CCGrid*, pages 403–408. IEEE, 2016.

[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *MSST*, pages 1–10. IEEE, 2010.

[34] R. Stoica and A. Ailamaki. Enabling efficient OS paging for main-memory OLTP databases. In *DaMoN*, page 7. ACM, 2013.

[35] C. Ungureanu, B. Debnath, S. Rago, and A. Aranya. TBF: A memory-efficient replacement policy for flash-based caches. In *ICDE*, pages 1117–1128. IEEE, 2013.

[36] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. Managing non-volatile memory in database systems. In *SIGMOD*, pages 1541–1555. ACM, 2018.

[37] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. Building an elastic query engine on disaggregated storage. In *NSDI*, pages 449–462. USENIX Association, 2020.

[38] G. Wang, A. R. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in MapReduce setups. In *MASCOTS*, pages 1–11. IEEE, 2009.

[39] H. Wang and P. J. Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *FAST*, pages 229–242. USENIX, 2014.

[40] X. Wu and A. L. N. Reddy. Exploiting concurrency to improve latency and throughput in a hybrid storage system. In *MASCOTS*, pages 14–23. IEEE, 2010.

[41] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28. USENIX, 2012.

[42] G. Zhang, L. Chiu, C. Dickey, L. Liu, P. Muench, and S. Seshadri. Automated lookahead data migration in SSD-enabled multi-tiered storage systems. In *MSST*, pages 1–6. IEEE.