MIT Libraries | DSpace@MIT

MIT Open Access Articles

SageDB: A learned database system

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation:

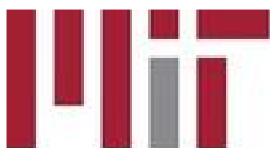
Published Version: http://cidrdb.org/cidr2019/program.html

Publisher:

Permanent Link: https://hdl.handle.net/1721.1/132282

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: http://creativecommons.org/licenses/by-nc-sa/4.0/



Massachusetts Institute of Technology

SageDB: A Learned Database System

Tim Kraska[1,2] Mohammad Alizadeh[1] Alex Beutel[2] Ed H. Chi[2] Jialin Ding[1]
Ani Kristo[3] Guillaume Leclerc[1] Samuel Madden[1] Hongzi Mao[1] Vikram Nathan[1]

[1]Massachusetts Institute of Technology [2]Google [3]Brown University

ABSTRACT

Modern data processing systems are designed to be general purpose, in that they can handle a wide variety of different schemas, data types, and data distributions, and aim to provide efficient access to that data via the use of optimizers and cost models. This general purpose nature results in systems that do not take advantage of the characteristics of the particular application and data of the user. With SageDB we present a vision towards a new type of a data processing system, one which highly specializes to an application through code synthesis and machine learning. By modeling the data distribution, workload, and hardware, SageDB learns the structure of the data and optimal access methods and query plans. These learned models are deeply embedded, through code synthesis, in essentially every component of the database. As such, SageDB presents radical departure from the way database systems are currently developed, raising a host of new problems in databases, machine learning and programming systems.

1. INTRODUCTION

Database systems have a long history of automatically selecting efficient algorithms, e.g., a merge vs hash-join, based on data statistics. Yet, existing databases remain general purpose systems and are not engineered on a case-by-case basis for the specific workload and data characteristics of a user, because doing so manually would be hugely time consuming. Yet, specialized solutions can be much more efficient. For example, if the goal is to build a highly-tuned system to store and query ranges of fixed-length records with continuous integer keys (e.g., the keys 1 to 100M), one should not use a conventional index. Using B+Trees for such range queries would make not much sense, since the key itself can be used as an offset, making it a constant $O(1)$ operation to look-up the first key of a range.[1] Indeed, a simple C program that loads 100M integers into an array and performs a summation over a range runs in about 300ms on a modern desktop, but doing the same operations in a modern database (Postgres 9.6) takes about 150 seconds. This represents a 500x overhead for a general purpose design that isn't aware of the specific data distribution.

Similar benefits extend to operations like sorting or joins. For sorting, if we know keys come from a dense integer domain, we can simplify the sorting of incoming records based on the primary key, as the key can be again used as an offset to put the data directly into a sorted array, reducing the complexity of sorting from $O(N \log N)$ to $O(N)$ for this particular instance of the data. Joins over dense integer keys also simplify to lookups, requiring only a direct lookup using the key as an offset. Inserts and concurrency control might also become easier as we do not need to keep index structures in sync.

Perhaps surprisingly, the same optimizations are also possible for other data patterns. For example, if the data contains only even numbers or follows any other deterministic distribution we can apply similar optimizations. In short, we can optimize almost any algorithm or data structure used by the database with knowledge of the exact data distribution. These optimizations can sometimes even change the complexity class of well-known data processing algorithms. Of course, in most real-world use cases the data does not perfectly follow a known pattern, and it is usually not worthwhile to engineer a specialized system for every use case. However, if it were be possible to *learn* the data pattern, correlations, etc. of the data, we argue that we can automatically synthesize index structures, sorting and join algorithms, and even entire query optimizers that leverage these patterns for performance gains. Ideally, even with imperfect knowledge of the patterns it will be possible to change the complexity class of algorithms, as in the above example.

In this paper we present our vision of SageDB, a new class of data management system that specializes itself to exploit the distributions of the data it stores and the queries it serves. Prior work has explored the use of learning to tune knobs [37, 34, 7, 9, 35], choosing indexes [12, 4, 36] partitioning schemes [6, 2, 27, 28], or materialized views (see [5] for a general overview) but our goal is different. We argue that **learned components can fully replace core components of a database** system such as index structures, sorting algorithms, or even the query executor. This may seem counter-intuitive because machine learning cannot provide the semantic guarantees we traditionally associate with these components, and because machine learning models are traditionally thought of as being very expensive to evaluate. This vision paper argues that neither of these apparent obstacles are as problematic as they might seem.

In terms of performance, we observe that more and more devices are equipped with specialized hardware for machine learning. For example, the iPhone has the "Neural Engine", Google's phone have a "Visual Core," Google's Cloud has Cloud TPUs, and Microsoft developed BrainWave. As it is easier to scale the simple (math) operations required for

[1]Note, that we use the big O-notation here over the particular instance of a database, similar to the notation of instance-optimality[10], except that our class of databases is exactly one.

machine learning, these devices already deliver a stunning performance. For instance, Nvidia's TESLA V100 product is capable of performing 120 TeraFlops of neural net operations. It was also stated that GPUs will increase 1000× in performance by 2025, whereas Moore's law for CPUs essentially is dead [1]. By replacing branch-heavy algorithms with neural networks, the DBMS can profit from these hardware trends.

Similarly, it is often surprisingly easy to provide the the same semantic guarantees. For example, a B-Tree can be seen as a model that points to a page containing records with a particular key, requiring a scan of the page to return all records that actually satisfy a query. In this sense, a B-Tree already trades off execution performance for accuracy [19]. Learning-based models will simply have more flexibility to explore that trade-off.

Finally, aggressive use of synthesis and code generation will allow us to automatically create efficient data structures, which combine models with more traditional algorithms. Here our goal is to (1) generate code tailored to the user's data distribution, and (2) synthesize database components with embedded *machine learning models*, which balance the trade-off between memory, latency, and compute for a given use case while providing the same semantic guarantees as the traditional components.

Building SageDB requires a radical departure from the way database systems are currently developed and involves challenges from databases, machine learning and programming systems. SageDB is currently being developed as part of MIT's new Data Systems for AI Lab (DSAIL), which consists of an interdisciplinary team with expertise in all of these areas. Our focus is on building a new analytical (OLAP) engine but we believe the approach also has significant advantages for OLTP or hybrid workloads. The remainder of the paper outlines the overall architecture of SageDB, individual challenges and presents some promising initial results.

2. MODEL-DRIVEN APPROACH

The core idea behind SageDB is to build one or more models about the data and workload distribution and based on them automatically build the best data structures and algorithms for for all components of the database system. This approach, which we call "database synthesis" will allow us to achieve unprecedented performance by specializing the implementation of every database component to the specific database, query workload, and execution environment.

2.1 Types of Customization

The proposed customization goes far beyond the current use of statistics and models about the data, hardware or performance of algorithms, which can be roughly classified in the following levels:

Customization through Configuration: The most basic form of customization is configuring the systems, aka knob tuning. Most systems and heuristics have a huge number of settings (e.g., page-size, buffer-pool size, etc.). Traditionally, database administrators tune those knobs to configure the general purpose database to a particular use case. In that sense the creation of indexes, finding the right partitioning scheme, or the creation of materialized views for performance can also be considered as finding the best configuration of the systems. It comes also at no surprise, that there has been a lot of work on automatically tuning those configurations [37, 34, 7, 9, 35] based on the workload and data characteristics.

Customization through Algorithm Picking: While configuring the system is largely static, databases have a long history of using query optimizers to dynamically "customize" the execution strategy for a particular query based on statistics about the data and the configuration (e.g., available indexes) of the system. That is, the query optimizer decides on the best execution order (e.g., predicate push-downs, join-ordering, etc.) and picks the best implementation from a set of available algorithms (e.g., nested-loop join vs hash-join). This declarative approach, which allows the user to specify on a high-level the query, while the system figures out how to best achieve it, is one of the most significant contributions of the database community.

Customization through Self-Design: Self-designed systems rely on the notion of mapping the possible space of critical design decisions in a system and automatically generating a design that best fits the workload and hardware [15]. Here the space of possible designs is defined by all combinations and tunings of first principle components, such as fence pointers, links, temporal partitioning, etc., which together form a "periodic table of data structures" [14]. This goes far beyond algorithm picking or configuring a system because new combinations of these primitives might yield previously unknown algorithms/data structures and can lead to significant performance gains [15].

Customization through Learning: In contrast to self-design, learned systems replace core data systems components through learned models. For example, in [19] we show how indexes can be replaced by models, whereas [21] shows how to learn workload-specific scheduling strategies. Models make it possible to capture data and workload properties traditional data structures and algorithms have a hard time supporting well. As a result, under certain conditions these data structures can provide the best-case complexity, e.g., $O(N)$ instead of $O(N \log N)$, and yield even higher performance gains than customization through self-design. Furthermore, they change the type of computation from traditional control-flow heavy computation to data-dependency-focused computation, which often can be more efficiently execute on CPUs and the upcoming ML accelerators.

These different types of customization can be composed. Especially, customization through self-design and customization through learning go hand in hand as the learned models often have to be combined with more traditional algorithms and data structures in order to provide the same semantic guarantees. More interestingly, models can potentially be shared among different components of a database system. In that sense, we argue in this paper that customization through learning is the most powerful form of customization and outline how SageDB deeply embeds models into all algorithms and data structures, making the models the *brain* of the database (see Figure 2).

2.2 The Power of Distributions

To illustrate the power of learning and synthesizing algorithms and data structures with models, consider the following thought experiment: Suppose we have a **perfect** model for the empirical cumulative distribution function (CDF) over a table R_1 with attributes $X_1, ..., X_m$ (i.e., there is no prediction error):

$$M_{CDF} = F_{X_1,...X_m}(x_1, ...x_m) = P(X_1 \leq x_1, ..., X_m \leq x_m)$$

For simplicity, we use CDF to refer to the empirical CDF of the data actually stored in the database (i.e., the instance of the data), not the theoretical CDF underlying the generative function for that data. For analytics, we mainly care about the empirical CDF, however, for update/insert-heavy workloads the underlying CDF plays a more important role. Even though this paper mainly focuses on analytics, we will touch upon the topic at the end.

Optimizer: First, and most obviously, such a model significantly simplifies query optimization as cardinality estimation will be essentially free (i.e., cardinality estimation is nothing more than probability mass estimation). This makes decisions about the best join ordering much easier, although, as the long history of literature on query optimization has shown, computing and maintaining compact and accurate multi-dimensional distributions is a key challenge, which we address in more detail later.

Indexes: As we showed in [19], such a model can also be used to implement point or range-indexes. Suppose we have an index-organized table on the primary key X_1, which stores the data sorted by X_1 in a continuous region on disk and fixed-size records of length l. In this case, we can calculate the offset of every record by calculating the probability mass less than a given primary key k and multiplying it with the total number of records N and the size of each record:

$$F_{X_1,\dots X_m}(key, \infty_2 \dots, \infty_m) * N * l$$

In this case, we also say that M_{CDF} *predicts* the position of key k. Note, that this index supports range requests as it returns the first key equal or larger than the lookup key. Assuming we can perform a lookup in the CDF in constant time (an assumption we address below), this makes the lookup of any key an $O(1)$ operation while traditional tree structures require $O(\log N)$ operations.

Interestingly, the CDF can also be used for secondary indexes or multi-dimensional indexes. However, in those cases we need not only to build a CDF model for the index but also optimize the storage layout (see Section 3 for a more detailed discussion).

Compression: Additionally, the CDF can also be used to compress the data. For simplicity, lets assume we have a sorted column, i.e., <*attr*,*r-id*>-pairs, and a CDF model only over the column *attr*. If we can reverse the CDF – i.e., compute, for a position in the column, the value at the column – then this inverse CDF model effectively allows us to avoid storing the values of *attr* at all. This scheme can be best regarded as a higher-level entropy compression. However, the beauty in the context of a DBMS is, that the same model for indexing, query optimization, etc. could potentially be reused for compression.

Execution: Similarly, the CDF model can help with partial or full-table joins or sorts. For example for sorting, Bayes theorem can be used to "rewrite" M_{CDF} for any given subset of a table to achieve a new model $\hat{M}_{CDF}$ which can be used to predict the position of every record within the sorted array, turning sorting into an $O(N)$ operation. For joins, we need an M_{CDF} model for every table, and afterwards can use the model to "predict" the position of any join-key, similar to the indexing case. Furthermore, certain aggregations like counting also become free as the system can use the model directly to answer them without even going to the data.

Advanced Analytics: A CDF can also be used for many data cube operations and for approximate query answers. For example, most OLAP queries are concerned with summing and counting. As a result in many cases we can use CDF model to directly (approximately) answer those queries. Furthermore, such a model could be directly used as part of machine learning models or help to build predictive models faster; in the end most machine learning models are about distribution. However, there is one caveat: machine learning is usually about generalizability, and not the empirical CDF.

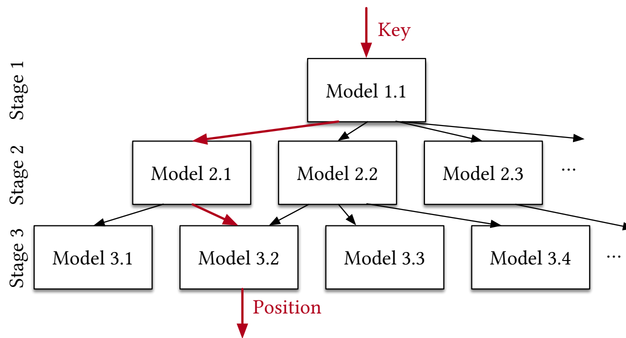


Figure 1: RMI Model

Discussion: This thought experiment using a CDF model shows how deep such a model could be embedded into a database system and what benefits it could provide. But it is not just about the CDF. For example, a query optimizer would also need a cost model to estimate the execution time for a given plan, which can either be a traditional hand-tuned model or a learned model. Moreover, in some cases it makes sense to learn the entire algorithm. For example, even with perfect information about the input designing a near-optimal scheduling algorithms with low-complexity is extremely hard. In such cases, learning data/query-specific algorithms might provide an interesting alternative.

2.3 What Models Should be Used

A key challenge is choosing the right model for SageDB's brain. The answer is surprisingly simple: whatever works. In some cases, histograms may be sufficient; in others neural nets may be the right approach. However, what we found so far is that histograms are often too coarse grain or too big to be useful. On the other hand, at least for CPUs, deep and wide neural nets are often too costly to be practical, although this will likely change in the future with the advances of TPUs and other neural processors.

As of today, we found that we often need to generate special models to see significant benefits. As an example of a generated model, consider the RMI structure presented in [19] (see Figure 1). For the one-dimensional case, the core idea of RMI is very simple: (1) fit a simple model (linear regression, simple neural net, etc.) over the data; (2) use the prediction of the model to pick another model, an expert, which more accurately models the subset of the data; (3) repeat the process until the last model makes a final prediction. Since the RMI uses a hierarchy of models to divide the problem space into sub-areas, each with its own expert model, the RMI resembles the hierarchical mixture of experts [16]; however, the RMI is not a tree, but rather a directed acyclic graph. In [19] we showed that the RMI model can significantly outperform state-of-the-art index structures and is surprisingly easy to train. At the same time, this type of model will probably not be used as

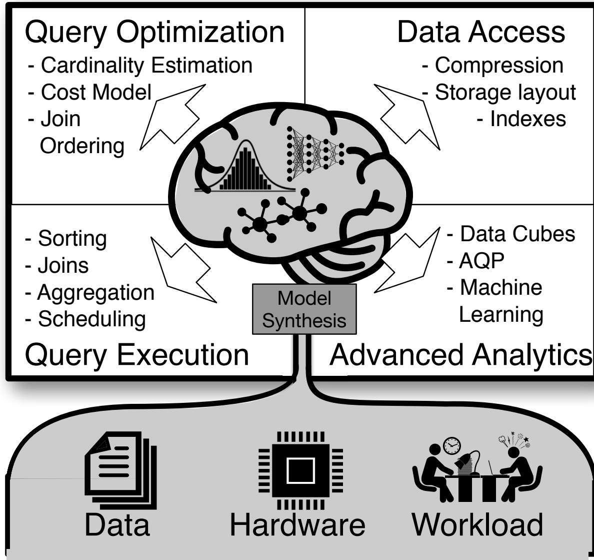


Figure 2: SageDB Overview

part of predictive modeling as it has a tendency to overfit; it creates a very good representation of $P(X \leq t)$ for the empirical distribution but not necessarily the underlying data distribution.

However, RMI is just a starting point. For example, it is possible to make the top model or bottom model more complex, replace parts of the models at a particular level stage with other types of models, use quantization, vary the feature representation, combine models with other data structures, and so on. We therefore believe we will see an explosion of new ideas on how to most efficiently generate models for database components to achieve the right balance between precision, low latency, space, and execution time for a given workload (see Sections 4-5).

2.4 Architecture Overview

Figure 2 shows the overall architecture of SageDB. As described above, our key idea to build or select ("synthesize") the best implementation of each component of the data processing engine for a particular application.

This synthesis is done by first learning one or more data distributions, workload, and hardware models. While in theory a single, "master" model might be sufficient, in practice we expect to create several models to balance the execution time vs. accuracy. As with RMI, we expect that many of the model architectures are very different from the large, deep networks currently popular in the machine learning literature for tasks like perception. This set of synthesized components form then the "brain" of SageDB.

Currently our components (e.g., index structure) "only" use these models. However, our grand vision is that the system synthesizes individual components specialized to the data, workload, and hardware, and that the specific choice of components and their interactions will also be environment-dependent. For example, in an update-intensive setting, different models and components may be chosen than in a more analytics focused environment, or in a distributed setting models more appropriate for batched and distributed updates may be selected.

Individual components can often share the previously learned models (e.g., the optimizer and indexes can both use the same distributions). In some cases the components themselves may be entirely learned (e.g., see Section 4.3 as an example) or form a hybrid between a model and some generated code. Therefore, while in Figure 2 there is a clear separation of model and algorithm, we expect a more fluid transition in practice. For example, reinforcement learning techniques are likely to be crucial in many domains (e.g., query optimization), creating a more iterative process between modeling and synthesis.

In the following, we outline some initial results on how to use learned approaches and models for various database components in more detail.

3. DATA ACCESS

The storage layout and index structures are the most important factors to guarantee efficient data access, and both are amenable to be enhanced by data and workload models as this section will show.

3.1 Single Dimensional Indexes

Index structures are already models, because they "predict" the location of a value given a key. As an example, consider an in-memory store in which all data is stored in one continuous array sorted by timestamp. In this setup, a cache-efficient B-Tree often provides better performance than simple search because of caching-effects and can be seen as a mapping from lookup key to a position inside the sorted array of records. Thus, the B-Tree is a model, in ML terminology a regression tree: it maps a key to a position and consequently, we can replace the index with any other type of model. While one might be wondering, how other models can guarantee that they also find all the relevant data, it is surprisingly simple: the data has to be sorted to support efficient range requests, so any error is easily corrected by a local search around the prediction.

Initial results: In [19] we showed that by using the previously mentioned RMI index, models can outperform state-of-the-art B-Tree implementations by a factor of two while being orders of magnitude smaller (note, that the updated arXiv version contains new results). Since then we extended the idea to also support data stored on disk, compression, inserts, as well as multi-dimensional data (see next section).

3.2 Multi-Dim. Indexes and Storage Layout

The analog to a B-Tree for multi-dimensional data is an R-Tree, which stores a tree of bounding boxes over the data; each node in the tree is a bounding box whose children are the smaller bounding boxes contained within it. As such an R-Tree is an approximate model that predicts the pages whose points lie within the query boundaries using this tree structure. In particular, when data is stored contiguously on disk or memory, an R-Tree is a model that maps a query rectangle to a list of index ranges $[s_i, t_i]$, such that the index of every point lying in the rectangle is contained in the union of these ranges. Similar to the B-Tree, it is an approximate model as these ranges must later be exhaustively scanned to find only those points that satisfy the query.

Interestingly and as before, other models can be used as a drop-in replacement for an R-Tree. However, as with the one-dimensional case we need to ensure that we can efficiently "correct" any imprecision of the model, which is

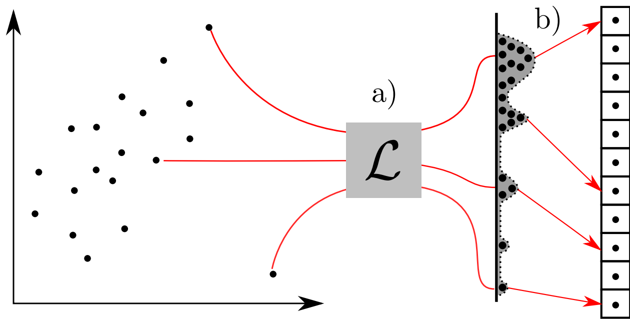


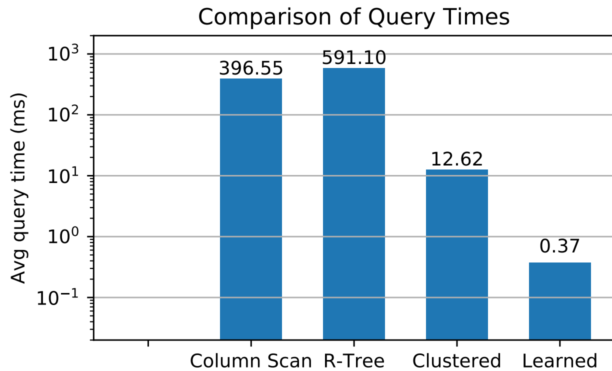
Figure 3: Two stages process for multi-dimensional indices: a) Points are projected onto a 1-dim. space using a $\mathcal{L}$ model, b) Physical location is obtained using a trained CDF model (e.g., an RMI)

much harder in the multi-dimensional case because no obvious storage order exists (recall, in the one-dimensional case, we assume the data is sorted by key and that any imprecision can be easily corrected by a localized search). Thus, the success of a learned index depends not only on the choice of model, but also on how the data is arranged in memory/disk, i.e., its *layout*; a poor layout might result in lower accuracy of the model and/or to many ranges every query needs to look up.

Choosing a layout: The layout can be described by a projection function $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$, mapping each data record of dimension d to a sort key that determines its relative location in the ordering (see Figure 3). The most natural choice for such a layout is a multi-dimensional CDF, i.e., $\mathcal{L}(r) = P(X_1 < r_1, \ldots, X_d < r_d)$. Unfortunately, this suffers from a major drawback: two points that have the same sort key may be far from each other in $\mathbb{R}^d$. Even a small query rectangle around one of those points results in a sort key range that includes a large number of extraneous points.

While many possible projection strategies exist, we found that successively sorting and partitioning points along a sequence of dimensions into equally-sized cells produces a layout that is efficient to compute, learnable (e.g., in contrast to z-order, which is very hard to learn), and tight (i.e., almost all points in the union of the index ranges satisfy the query). The idea of partitioning is not new: existing database systems allow administrators to manually select multiple columns on which to partition [13, 23]. Our contribution is (1) to allow projections which are not axis-aligned and more complex than static hierarchy partitions, (2) use models to actually do the projection and find the data on storage, and (3) use the data and query distribution to automatically learn both the partition dimensions and the granularity of each partition without requiring imprecise, manual tuning. At a coarse level, our partitioning also bears some similarity to locality sensitive hashing (LSH) [11], which has primarily been used for similarity search. As in LSH, projecting by sorting and partitioning along a sequence of dimensions creates multi-dimensional "cells" that partition the data space. In our learned index, points are ordered by the index of the cell that they occupy.

Initial results: We implement the learned index over an in-memory column store [33] with compression, and compare our results to the following three baselines implemented over the same column store (Fig 4):



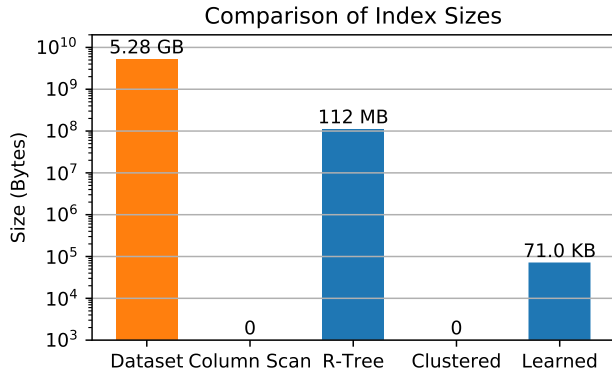


Figure 4: Learned multi-dimensional index performance on TPC-H data using a column store. Note the log-scale.

- *Column Scan*: a full scan of all in the query involved columns.
- *Clustered*: a clustered index that sorts the data by the column which provides the best overall performance (e.g., if all columns are queried equally often, this would be the most selective dimension). At query time, if the involved column is used as part of a predicate, it first performs a binary search over the sort column to narrow down the range of points that are scanned.
- An *R-Tree* index over the dimensions present in the query workload.

All benchmarks are single-threaded and use data from the lineitem table of the TPC-H benchmark, with 60 million records. All data is pre-loaded into memory (there is no disk access). The query workload consists of randomly-generated range queries over up to three dimensions, and the same three dimensions are used for all queries. Each query has a selectivity of around 0.25%. After performing the range filter, each query also performs a `SUM` aggregation on a fourth dimension. Figure 4 shows that the Learned Index boosts range query speed by an average of 34× compared to the clustered index, with only a small space overhead. Note that the R-Tree performs worse than a full column scan since it must access every column in its index, even if they do not appear in the query.

Performance Analysis: A clustered index performs best when the clustered dimension is present in the query filter. If the clustered dimension is absent, the clustered index reverts to a full column scan. On the other hand, a learned index is

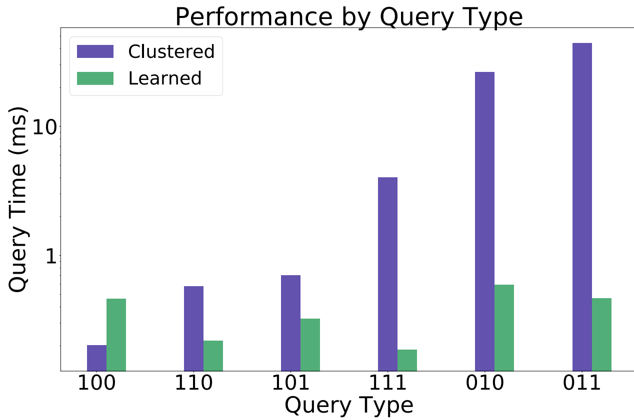


Figure 5: Learned Index performance versus a Clustered Index baseline, for various query types. Note the log scale.

sorted on multiple dimensions. If we assume that the clustered dimension is also present in the learned index's sorted dimensions, the learned index should outperform a clustered index in two scenarios:

1. The query filters on the clustered column and at least one other dimension used in the learned index. The learned index uses the fact that the data is partially sorted by the second dimension to limit the number of extraneous scanned points.

2. The query's filter contains a dimension in the learned index, but not the clustered dimension. As opposed to a clustered index, which performs a full column scan, the learned index is able to limit the region of points that are scanned by considering only the relevant cells.

To quantify the performance gains of the learned index, we break down the previous results (Figure 4) by the query *type*, which denotes the dimension filtered in each query. The learned index sorts on three dimensions, so for brevity, each query type is denoted by a three digit binary number: the ith most significant digit is 1 if the query filters on the ith dimension in the sequence of dimensions used by the learned index. For example, the query type '110' indicates queries that filter on the first and second dimension used in the learned index, but not the third. The baseline, a clustered index, is sorted on the learned index's first dimension, which is the most selective dimension in our query workload.

Figure 5 confirms that the learned index outperforms the clustered index on almost every type of query. The only query type where the learned index does not improve over the clustered index is when the clustered dimension is the *only* dimension in the query. In this case, a clustered index is able to optimize its scan over exactly the points that match the query, while the learned index must incur the overhead of scanning the extraneous points located in the same cells as the matching points.

All queries for each query type have the same selectivity of around 0.25%, to control for the impact of result size on query time. As a result, the range of the filter on a single dimension depends on how many other dimensions are part of the query: a query with type '110' must query a larger range on the first dimension compared with a query of type '100' in order to achieve the same selectivity. This effect explains why the clustered index queries with type '111' are noticeably slower than '110' and '101', which are in turn slower than '100'. On the other hand, the learned index avoids this slowdown since its query time scales with the range of the entire query, instead of just the range of the first dimension.

Algorithm 1 Learned sorting algorithm

Input a - the array to be sorted
Input F - the CDF model for the distribution of a
Input m - the over-allocation parameter
Output o - the sorted version of array a

```
1: procedure LEARNED-SORT(a, F, m)
2:     o ← [∞] ∗ (a.length ∗ m)
3:     s ← {}
4:     // STEP 1: Approximate ordering
5:     for i in a do
6:         pos ← F(i) ∗ a.length ∗ m
7:         if o[pos] = ∞ then
8:             o[pos] ← i
9:         else
10:            s ← s ∪ {i}
11:    // STEP 2: Touch-up
12:    INSERTION-SORT(o)
13:    QUICKSORT(s)
14:    // STEP 3: Merging
15:    return MERGE-AND-REMOVE-EMPTY(o, s)
```

4. QUERY EXECUTION

The previous section outlined how learning the data distribution, a model of the CDF, can replace traditional index structures with potentially tremendous advantages for TPUs/GPUs. Maybe surprisingly, a very similar idea can be used to speed-up sorting, joins, and even group-bys.

4.1 Sorting

The basic idea to speed up sorting is to use an (existing) CDF model F to put the records roughly in sorted order and then correct the nearly perfectly sorted data as shown in Algorithm 1. Depending on the execution cost of the model and its precision, this sorting technique can have significant performance advantages over alternative sorting techniques.

For example, assume the following query: `SELECT * FROM customer c ORDER BY c.name` with a table size of N and a secondary learned index on `c.name` whereas the data is stored in order of the customer id *not* name. To quickly sort the data by name we put every record based on its key k into a roughly sorted order by scaling the output of the CDF model over the name to the number of elements in the array ($pos = F(k) * N$). At this point we map each record into a position in the output array and, if there is a collision, we store it some overflow array (Lines 5-10 in Algorithm 1). In order to decrease the number of collisions, we can instead allocate an output array that is m-times larger than N(e.g., $m = 1.37$) and then remove any empty positions at the end of this mapping. Note that for this case, the predicted position will be calculated as $pos = F(k) * m * N$ (Line 6 in Algorithm 1). In addition, we can sort into buckets rather than into positions $bucket = F(k) * N/bs$, where bs is the bucket size to further reduce the number of conflicts. The assumption here is, that if we make the buckets a multiple

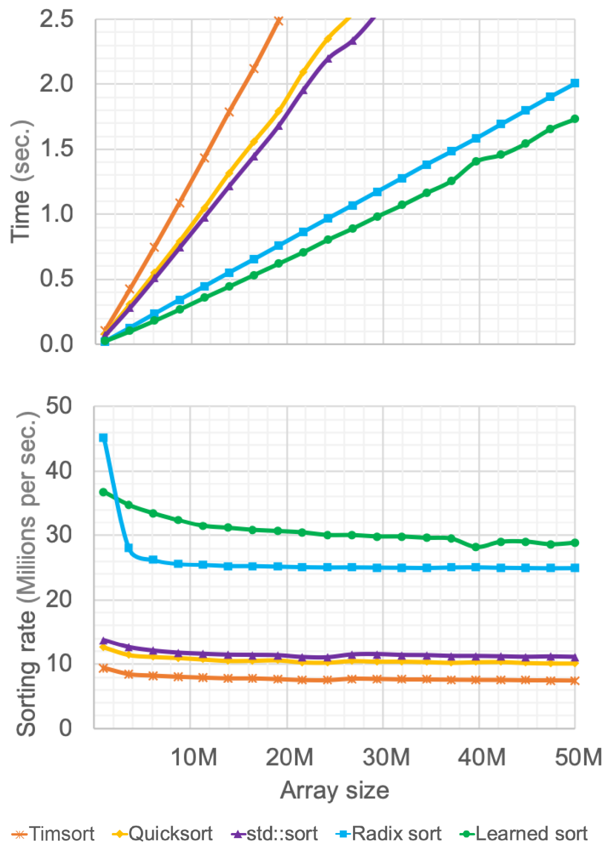


Figure 6: Model-based Sorting

of a cache-line, that we can quickly sort within a bucket, while reducing the chance of conflicts (e.g., that we map more items to a bucket than their are slots). A similar idea is used as part of cuckoo hashing [25]. In either case, if our model is non-monotonic, we have to use an efficient local-sort algorithm (i.e. insertion sort, which works very fast with almost-sort arrays) to correct any sorting mistakes (Line 12 in Algorithm 1). Finally, we sort the overflow array (Line 13) and merge the sorted array with the overflow array while removing empty elements (Line 15).

Conceptually, the model-based sorting is very similar to Radix sort. Radix sort has a complexity of $O((n) * \log_b(k))$, with k being the maximum possible value and b being the base for the prefix sorting. That is, if we would Radix sort arbitrary 32-bit integers based on 1 Byte at a time (i.e., a base of 2^8), the complexity would be $O(4 * N)$. The idea of our learned-based sorting is to have the base b close to k, which would allow to sort in $O(N)$. Obviously, this disregards the model complexity, which is likely to increase with k and n in many real-world scenarios. At the same time, it is easy to come up with scenario for which a model-based sorting approach would vastly outperform a Radix-based sort. For example, if we have keys from a very large domain (i.e, k is really big), but most keys are concentrated in a small area of the domain space. In that case a model could learn a very compact representation for the CDF and thus, potentially sort close to $O(N)$, whereas traditional Radix sort algorithm would suffer from the large domain size. The sensitivity to the domain size, is also one of the reasons most algorithm libraries don't implement Radix sort, but rather use variants of Quicksort.

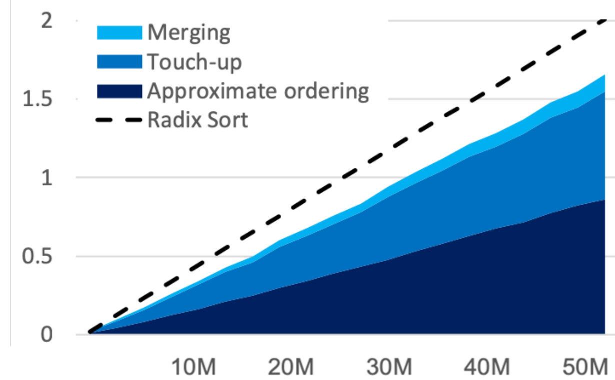


Figure 7: The performance of each algorithmic step for Learned Sort as compared to Radix sort.

Figure 6 shows early results of a learned approach to sorting for increasingly large data sizes consisting of 64-bit doubles randomly sampled from a normal distribution running single threaded. We use both the bucketing and m-times larger array allocation and our model is non-monotonic, requiring the touch-up step of Line 12 in Algorithm 1. The figure compares Timsort, which is the default sorting algorithm for Java[2] and Python[3], Quicksort, std::sort from the C++ library, Radix sort, and a learned sort. We report the total sorting time (left) as well as the sorting rate (right). Note, that none of the implementations were tuned to take advantage of SIMD or FMA instructions. As it can be seen the learned variant has a significant performance benefit over the comparison based approaches and an average margin of 18% faster execution than Radix sort. Even more interestingly, the sorting rate for the learned approach is very steady. Note that the RMI model building time was not included in this case for the learned sorting approach. If we would train a small model based on a (logarithmic) sample of data the difference between Radix sort and learned sort shrinks to 10%.

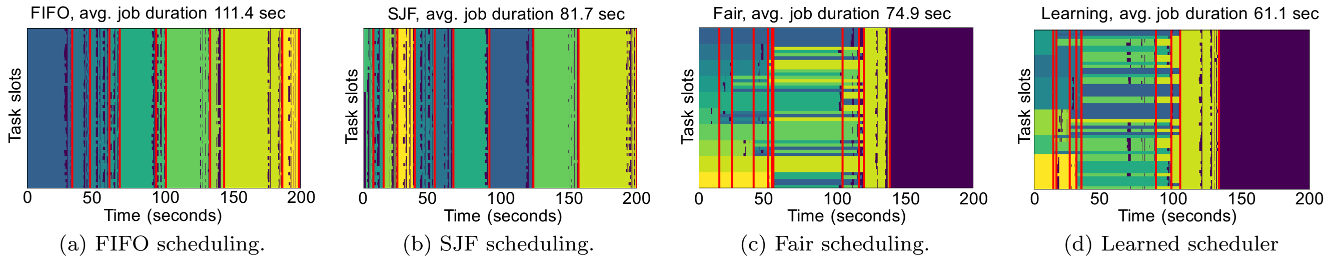
Figure 7 shows the time-breakdown of the learned sorting between the approximate ordering (Line 5-10 in Algorithm 1), touch-up (Line 12-13), and merging the results (Line 15). We believe, that this result shows the potential for learned sorting approaches, which we expect to be the biggest for distributed sorting, large key sizes, and with GPUs/TPUs/FPGAs accelerators. Hence, many open research challenges still remain from properly determining the complexity class to building SIMD optimized implementations.

4.2 Joins and other operations

Joins can be sped up in a similar was as sorting. For example, the CDF model could be used to skip over portions of data as part of a merge sort. Consider a column store in which the two join columns are sorted and we have one CDF model per column. Now during a merge-join, we can use the model to skip over data that will not join.

The learned models could also be helpful to improve other algorithms. For example, a model could be used as a more

[2]https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-java.lang.Object:A-
[3]https://svn.python.org/projects/python/trunk/Objects/listsort.txt



(a) FIFO scheduling. (b) SJF scheduling. (c) Fair scheduling. (d) Learned scheduler

Figure 8: Our learned scheduler improves average job completion time of 10 randomly-sampled TPC-H queries by 45% over Spark's default FIFO scheduler, and by 19% over a fair scheduler on a cluster with 50 task slots (executors). Different queries are shown in different colors; vertical red lines show job completions.

efficient hash-function as part of a group-by. Here the challenge is to adjust the learned model if the not all keys are selected and efficiently integrate into the query plan. Similarly, the model could be used as cardinality estimations as part of the group-by or other operations, for example, to allocate enough memory for intermediate results.

4.3 Scheduling

Today's database systems use simple scheduling policies like first-come-first-serve for their generality and ease-of-implementation. However, a scheduler customized for a specific workload can perform a variety of optimizations; for example, it can prioritize fast and low-cost queries, select query-specific parallelism thresholds, and order operations in query execution to avoid bottlenecks (e.g., leverage query structure to run slow stages in parallel with other non-dependent stages). Such workload-specific policies are rarely used in practice because they require expert knowledge and take significant effort to devise, implement, and validate.

In SageDB we envision a new scheduling system that automatically learns highly-efficient scheduling policies tailored to the data and workload. Our system represents a scheduling algorithm as a neural network that takes as input information about the data (e.g., using a CDF model) and the query workload (e.g., using a model trained on previous executions of queries) to make scheduling decisions. We train the scheduling neural network using modern reinforcement learning (RL) techniques to optimize a high-level system objectives such as minimal average query completion time.

Initial results: We implemented a first RL-based scheduling system [21] for batch data analytics jobs and a prototype in Spark [38]. Our system uses a graph neural network [3, 18] to encode scheduling algorithms for queries represented as directed acyclic graphs (DAGs) of processing stages. The neural network decides both how to divide executors between DAGs and how to order the execution of stages within each DAG. We train the neural network through a large number of simulated experiments, where it schedules a workload, observes the outcome, and gradually improves its policy using a policy gradient RL algorithm.

Figure 8 visualizes the schedules imposed by *(a)* Spark's default FIFO scheduling; *(b)* a shortest-job-first (SJF) policy that strictly prioritizes short jobs; *(c)* a fair scheduler that dynamically divides task slots between jobs; and *(d)* the learned scheduling policy, each running a random mix of TPC-H queries. The learned scheduler improves average job completion time (JCT) by 45% over Spark's default FIFO scheduler, and by 19% over the fair scheduler. It achieves this speedup by *(i)* completing short jobs quickly — note the five jobs that finish in the first 40 seconds (shown as vertical red lines); and *(ii)* maximizing cluster efficiency. Unlike the SJF policy, which naïvely dedicates task slots to small jobs in order to finish them early (but inefficiently), the learned scheduler runs jobs near their parallelism "sweet spot". By controlling parallelism, it reduces the total time to complete all jobs by 30% compared to SJF. Further, unlike fair scheduling, it learns to partition task slots non-uniformly across jobs, improving average JCT.

5. QUERY OPTIMIZER

Traditional query optimizers are extremely hard to build, maintain, and often yield sub-optimal query plans. The brittleness and complexity of the optimizer makes it a good candidate to be learned. Indeed, there have been several recent approaches that aim to learn more efficient search strategies for the best join-order [20, 22], to improve cardinality estimation [17], or to learn the entire plan generation process through reinforcement learning [24].

However, they largely make unrealistic assumptions. For example, [20] assumes perfect cardinality estimation for predicates over the base table, whereas both [17] and [24] do not show if the improve cardinality estimation actually lead to better query plans. The reason is, that it is relatively easy to improve the average cardinality estimation but extremely hard to improve the cardinality estimation in the cases where it really matters: when it actually changes the join ordering or when an estimate causes the "wrong" join algorithms to be selected. Furthermore, existing approaches suffer from the initialization problem; they often require huge amounts of training data and assume no ad-hoc queries.

We therefore have to started to explore alternative approaches to query optimization. Most notable, we tried to make a traditional cost model (e.g., the Selinger model [29]) differentiable. That is, we start with the traditional hand-tuned model as, for example, defined in [29], but make it differentiable so that we can improve the model after every query to customize it for a particular instance of the data. The biggest advantage of this approach is, that it solves the initialization problem and the question of how the model would perform for ad-hoc queries. In the beginning the system uses the standard model, which is then refined over time. Initial results of this approach has shown that we can indeed improve the quality of the model, but that we can not achieve huge gains without also making significant improvements in the cardinality estimation problem.

We therefore started to explore hybrid model-based approaches to cardinality estimation. The core idea is, that try to find the best balance between a model, which learns the underlying patterns and correlations of the data, and exception/outlier lists, which capture the extreme — hard to learn — anomalies of the particular instance of the data.

6. OTHER OPPORTUNITIES

Beyond query optimization, data access path optimization, and query execution, a model-centric database design can provide additional benefits from approximate query processing (AQP) to Inserts as outlined next:

AQP and Data Cubes: A CDF model opens up completely new possibilities for approximate query processing, especially data cube operations. Recall, that a multi-dim CDF allows us to arbitrarily filter by attributes. Furthermore, it is likely that the several dimensions of a typical data cube have correlations, which can be learned. This in turn can lead to compact models, which can be used to approximate queries over them with high accuracy. While other works have explored the use of models for AQP [26, 8], the key difference is that SageDB uses the same type of models for indexing, AQP, query optimization, etc.

Predictive Modeling: Similarly, CDF models can be used to build predictive models over data. However, predictive models require generalizability to new data, whereas our models focus on learning an empirical distribution where overfitting is a good thing. Of course, there is a strong connection between the empirical and underlying data distribution, which potentially can be exploited to speed-up the training of predictive models.

Inserts/Updates: While the current focus of SageDB is on analytics, there exists huge potential for update-heavy workloads. For example, assume that the data from the inserts follow a similar distribution to the already stored data and that the model generalizes over the data. In that case no re-training (i.e., re-balancing) is necessary. For indexes, this means that inserts would become $O(1)$ operations. In [19] we discuss the potential for update-heavy workloads in more detail, though clearly much more work has to be done.

Complexity Analysis and Robustness Guarantees: There exists several interesting theoretical questions. The most obvious one is, how to define a complexity class for the learned algorithms and data structures. The obvious starting point is the definition of instance-optimality which defines that an algorithm B is instance optimal over a class of algorithm A and a database d, if

$$cost(B, d) \leq c \cdot cost(a, d) + c' \quad (1)$$

for every choice of $a \in A$. Hence, the goal of customization through Self-Design and Learning, can probably be phrased as the goal of finding the best possible algorithm for a given database.

Similarly, we would like provide worst-case performance guarantees if the data distribution or the workload shifts. For example, in [19] we proposed a hybrid learned data structure, which in the worst-case provides the look-up performance of a B-Tree $O(\log N)$. Yet, how to provide these guarantees more broadly is largely an open question and has a strong connection to *smoothed analysis* [30, 31] and robust machine learning [32].

7. CONCLUSION

SageDB presents a radical new approach to build database systems, by using using ML models combined with program synthesis to generate system components. If successful, we believe this approach will result in a new generation of big data processing tools, which can better take advantage of GPUs and TPUs, provide significant benefits in regard to storage consumption and space, and, in some cases, even change the complexity class of certain data operations. We presented initial results and a preliminary design that show the promise of these ideas, as well as a collection of future directions that highlight the significant research opportunity presented by our approach.

8. REFERENCES

[1] Moore Law is Dead but GPU will get 1000X faster by 2025. https://tinyurl.com/y9uec4w6.

[2] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 359–370, 2004.

[3] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[4] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft SQL server. In *VLDB*, 1997.

[5] S. Chaudhuri and V. R. Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, 2007.

[6] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[7] B. K. Debnath, D. J. Lilja, and M. F. Mokbel. Sard: A statistical approach for ranking database tuning parameters. In *2008 IEEE 24th International Conference on Data Engineering Workshop*, pages 11–18, April 2008.

[8] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 588–599, 2004.

[9] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *PVLDB*, 2(1):1246–1257, 2009.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pages 102–113, New York, NY, USA, 2001. ACM.

[11] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[12] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K.*, pages 208–219, 1997.

[13] IBM Knowledge Center. Table partitioning and multidimensional clustering tables. https://www.ibm.com/support/knowledgecenter/en/

SSEPGG_9.5.0/com.ibm.db2.luw.admin.partition.doc/doc/c0021605.html, 2018.

[14] S. Idreos, K. Zoumpatianos, M. Athanassoulis, N. Dayan, B. Hentschel, M. S. Kester, D. Guo, L. M. Maas, W. Qin, A. Wasay, and Y. Sun. The periodic table of data structures. *IEEE Data Eng. Bull.*, 41(3):64–75, 2018.

[15] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 535–550, 2018.

[16] M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural Comput.*, 6(2):181–214, Mar. 1994.

[17] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *CoRR*, abs/1809.00677, 2018.

[18] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[19] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *SIGMOD*, pages 489–504, 2018.

[20] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning, 2018.

[21] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. *arXiv preprint arXiv:1810.01963*, 2018.

[22] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 3:1–3:4, 2018.

[23] Oracle Help Center. Database vldb and partitioning guide: Partitioning concepts. https://docs.oracle.com/cd/B28359_01/server.111/b32024/partition.htm, 2018.

[24] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, DEEM'18, pages 4:1–4:4, New York, NY, USA, 2018. ACM.

[25] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[26] Y. Park, A. S. Tajik, M. Cafarella, and B. Mozafari. Database learning: Toward a database that becomes smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 587–602, New York, NY, USA, 2017. ACM.

[27] A. Pavlo, E. P. C. Jones, and S. B. Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *PVLDB*, 5(2):85–96, 2011.

[28] J. Rao, C. Zhang, N. Megiddo, and G. M. Lohman. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 558–569, 2002.

[29] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[30] D. Spielman and S. Teng. Smoothed analysis: Motivation and discrete models. In *Workshop on Algorithms and Data Structures (WADS)*, 2003.

[31] D. Spielman and S. Teng. Smoothed analysis: An attempt to explain the behavior of algorithms in practice. *Comm. ACM*, 52(10):76–84, 2009.

[32] M. Staib and S. Jegelka. Distributionally robust deep learning as a generalization of adversarial training. In *NIPS workshop on Machine Learning and Computer Security*, 2017.

[33] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[34] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer. Using probabilistic reasoning to automate software tuning. *SIGMETRICS Perform. Eval. Rev.*, 32(1):404–405, June 2004.

[35] V. Thummala and S. Babu. ituned: a tool for configuring and visualizing database parameters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 1231–1234, 2010.

[36] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 101–110, 2000.

[37] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, New York, NY, USA, 2017. ACM.

[38] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.