Budget-Conscious Fine-Grained Configuration Optimization for Spatio-Temporal Applications

Keven Richly
Hasso Plattner Institute
Potsam, Germany
keven.richly@hpi.de

Rainer Schlosser
Hasso Plattner Institute
Potsam, Germany
rainer.schlosser@hpi.de

Martin Boissier
Hasso Plattner Institute
Potsam, Germany
martin.boissier@hpi.de

ABSTRACT

Based on the performance requirements of modern spatio-temporal data mining applications, in-memory database systems are often used to store and process the data. To efficiently utilize the scarce DRAM capacities, modern database systems support various tuning possibilities to reduce the memory footprint (e.g., data compression) or increase performance (e.g., additional indexes). However, the selection of cost and performance balancing configurations is challenging due to the vast number of possible setups consisting of mutually dependent individual decisions. In this paper, we introduce a novel approach to jointly optimize the compression, sorting, indexing, and tiering configuration for spatio-temporal workloads. Further, we consider horizontal data partitioning, which enables the independent application of different tuning options on a fine-grained level. We propose different linear programming (LP) models addressing cost dependencies at different levels of accuracy to compute optimized tuning configurations for a given workload and memory budgets. To yield maintainable and robust configurations, we extend our LP-based approach to incorporate reconfiguration costs as well as a worst-case optimization for potential workload scenarios. Further, we demonstrate on a real-world dataset that our models allow to significantly reduce the memory footprint with equal performance or increase the performance with equal memory size compared to existing tuning heuristics.

PVLDB Reference Format:
Keven Richly, Rainer Schlosser, and Martin Boissier. Budget-Conscious Fine-Grained Configuration Optimization for Spatio-Temporal Applications. PVLDB, 15(13): 4079 - 4092, 2022.
doi:10.14778/3565838.3565858

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/hyrise/table_configuration_optimizer.

1 INTRODUCTION

Large amounts of spatio-temporal data are continuously accumulated through the wide distribution of location-acquisition technologies. Positioning systems such as GPS enable the tracking of a broad-spectrum of moving objects [55]. Spatio-temporal data reflect the trajectories of moving objects and enable the analysis of movement patterns, which are increasingly used in various applications [19, 30, 39]. A moving object's trajectory is represented by a chronologically ordered sequence of observed locations (e.g., a set of timestamped coordinates in a geographical reference system). To store and process spatio-temporal data is not a trivial task due to the massive volumes of continuously captured data, varying access patterns and data characteristics for different applications, and changing workloads based on environmental influences [52, 54]. Based on the required interactive response times in different applications, in-memory computing systems are widely used to provide low latency query services [34, 49, 54].

In comparison to standalone storage systems specialized for trajectory data, relational database systems enable a simplified integration of further data sources [42]. Consequently, modern data management platforms are enhanced by engines for specific data types (e.g., spatial and spatio-temporal data) [32, 47, 49, 53]. By integrating spatio-temporal data management into relational database systems, the data querying benefits from the optimized data processing capabilities and advanced compression techniques [40]. Due to the relatively limited, expensive, and stagnating DRAM capacities of modern servers, the efficient utilization of the available resources is necessary to lower the memory footprint and consequently reduce the related total cost of ownership to store large volumes of spatio-temporal data [6, 20]. There are different aspects (e.g., auxiliary data structures or data compression) that impact such systems' memory footprint. While removing additional data structures (e.g., indexes) or applying compression techniques with higher compression rates reduce the memory footprint, they also influence the runtime performance. Based on the broad spectrum of tuning options (e.g., index structures [27]), the implications in a specific use case a hard to estimate [11]. To balance the various aspects, rule-based heuristics depending on data characteristics (e.g., data size or specific timeframes) are applied by database administrators [42]. By dividing the data of a table into various partitions, modern database systems enable fine-grained configuration decisions [15, 26, 33, 35]. This approach enables the independent definition of different optimizations for each of these data partitions, such as the sorting column, index configuration, data tiering, or applied compression scheme. All single configuration decisions have an impact on the overall memory consumption and runtime performance. Furthermore, they mutually influence each other, which makes the determination of performance-optimized and memory-efficient configurations difficult [3, 46, 56].

This paper introduces a joint linear programming (LP)-based approach to determine fine-grained configurations for specific spatio-temporal applications, which improves the cost-efficient storing of trajectory data in relational databases. In existing work, there are

Proceedings of the VLDB Endowment, Vol. 15, No. 13 ISSN 2150-8097.
doi:10.14778/3565838.3565858

several general approaches that optimize specific aspects like the compression schema selection [1, 5]. Other research focuses on the selection of optimized index structures [13, 43] or data placement strategies [7, 48]. As the different configuration decisions mutually influence each other, we seek to optimize the (i) compression, (ii) index, (iii) ordering, and (iv) tiering configuration *jointly* to determine the best runtime performance for a given workload and memory budget. Note that each of those individual tuning problems is already challenging in general. We can still address a joint optimization of these dimensions as we exploit the specific characteristics of spatio-temporal data and applications, i.e., a limited number of query types [39]. In contrast to business applications with hundreds of attributes per table [7], in general, spatio-temporal data has a manageable number of attributes, which enables us to reduce the solution space. Further, to obtain a manageable problem complexity, we focus on single-attribute indexes and discuss the use of specific selected multi-attribute indexes based on domain knowledge. Our contributions are the following:

- We develop three models to determine efficient fine-grained table configurations for spatio-temporal data by jointly optimizing (i) data compression, (ii) ordering, (iii) indexing, and (iv) tiering with different dependencies (Section 3).
- We provide extensions to (i) also gain robustness against different workload scenarios and (ii) to include modification costs for optimized reconfigurations (Section 4).
- We evaluate our LP-based approaches on a real-world dataset and demonstrate their applicability, effectiveness, and scalability in end-to-end experiments (Section 5).
- We show that compared to existing rule-based heuristics our configurations achieve an up to 70% better performance, i.e., regarding either performance or required memory.

2 OPTIMIZING TABLE CONFIGURATIONS

This section introduces the architecture of the research database *Hyrise* [16] and how its data partitioning concept enables fine-grained database optimizations (Section 2.1). We describe the optimization process (Section 2.2) and motivate the application-specific optimization of table configurations by demonstrating the impact of different tuning options on memory consumption and performance (Section 2.3).

2.1 Fine-Grained Database Tuning Options

Hyrise is a columnar main memory-optimized database. Each table in *Hyrise* is implicitly divided into horizontal partitions with a predefined maximum size (see Figure 1). A partition, called chunk, contains fragments of all columns of a table whereby the section of a column stored in a chunk is referred to as a segment. There are two types of chunks, mutable and immutable chunks. Only the most recent chunk is mutable, and consequently, all insertions, as well as MVCC-enabled updates, are appended to this unencoded chunk. When this write-optimized chunk's capacity is reached, it becomes immutable, and a new mutable chunk is created. A disadvantage of this approach is that we increase the memory footprint by additionally storing per-chunk metadata and redundant information (e.g., per-segment dictionaries for dictionary-encoded segments). In exchange, the database system can benefit from pruning during query execution, distribute the workload more efficiently, and apply fine-grained optimizations of table configurations. For that reason, similar concepts are applied by various databases [26, 33, 35].

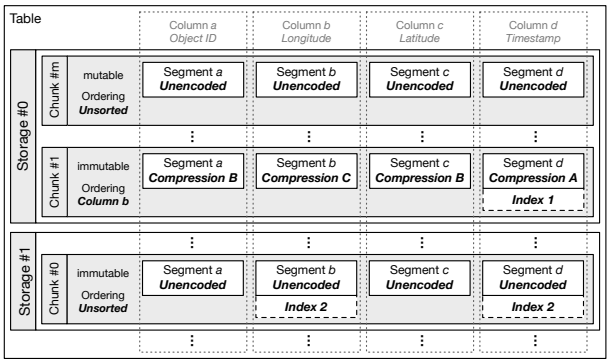

Figure 1: Depiction of the storage layout for an exemplary table configuration for two storage devices. Besides the allocation decision, we are able to select for each segment a compression, indexing, and sorting tuning option.

Fine-grained table configurations enable the application of different tuning approaches for various data chunks and segments of a table. Consequently, the configuration of different parts of the data can be optimized for specific workloads and data characteristics. *Hyrise* provides different tuning options, such as the sorting, indexing, and compression configuration of a chunk [42]. Also, *Hyrise* supports the tiering of entire chunks or segments to more cost-effective storage devices (e.g., NVRAM, SSD, or disaggregated memory, cf. [14, 51]). Hyrise uses C++'s polymorphic memory resources to provide a uniform interface that allows allocating data in DRAM, NVRAM, or on block devices using *UMap* [36, 37]. *UMap* is a user-space page fault handler that allows – in contrast to *mmap* – limiting the buffer size of cached pages. This enables the storing of data on multiple storage devices. All these configuration decisions have an impact on the system's performance and memory consumption. For in-memory databases, the used DRAM capacities are an important cost factor [4, 31]. Concerning spatio-temporal data volumes, minimizing the data footprint can significantly reduce the system's operating costs. Different tuning options reduce the memory consumption but also have implications on the performance, which are difficult to estimate for database administrators [3, 8, 11]. Consequently, various vendors apply relatively simple threshold-based approaches. Based on a defined threshold (e.g., data volume), data partitions are transferred to lower-cost storage mediums.

2.2 Process Overview

The optimization process is made up of a controller, benchmark engine, and configuration optimizer. The controller operates the process and acts as intermediary between the target database management system (DBMS) and the configuration optimizer. It collects the runtime and benchmark data from the target DBMS for the configuration optimizer, which uses the data to recommend a new table configuration based on the internal tuning models. Based on the selected tuning model and workload, the benchmark engine executes a set of benchmark queries on different table configurations.

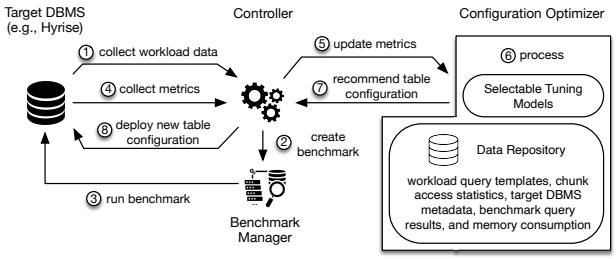

Figure 2: Optimization process - The controller collects the workload information of the target DBMS. Based on the workload data, the benchmark manager executes a set of benchmark queries. The workload data and the benchmark metrics are transferred to the configuration optimizer, which processes the updated data and uses the selected algorithm to generate an optimized table configuration. Then, the new table configuration is applied to the DBMS.

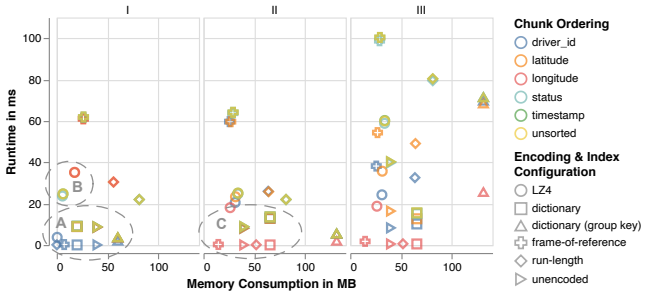

Figure 3: Impact of different tuning decisions on memory consumption and performance for ten million observed locations partitioned into ten chunks and specific scan operations: a *LessThanEquals* scan (selectivity: 0.01) on (I) the driver id column, (II) the longitude column, and (III) a *Between* scan on the longitude column with a selectivity of 0.1.

The optimization of a table configuration can be triggered based on specific metrics (e.g., time intervals, performance constraints) or after a mutable chunk reached the defined maximum size. As shown in Figure 2, in the first step ①, the controller collects the workload data from the target DBMS. During runtime, modern database systems track various parameters to optimize the performance of a DBMS autonomously [25]. Here, the SQL plan cache is used to extract the query templates of the workload. Each template describes a set of similar executed queries. Additionally, the segment access statistics captured by *Hyrise* are collected for the different query templates [14]. We can use min/max statistics of each segment to determine the relevant chunks for a specific query template. These workload statistics are used in various physical database tuning tools [8]. Based on the query templates and the selected tuning model, the benchmark engine ② creates a set of benchmark queries. In step ③, the benchmark consisting of isolated single column scan executions is conducted on different configurations to get information about the runtime performance and memory consumption of different encoding types for each column.

Due to the limited number of attributes in spatio-temporal data tables, the number of queries is manageable. Alternatively, estimated cost models [6, 28] or what-if analysis [2, 10] could be used to predict the runtime performance and memory consumption. By determining the input parameters for the LP models based on the stored trajectory data and queries, we can consider the application-specific characteristics in the optimization process. The controller ④ collects the benchmark results and ⑤ transfers the results as well as the workload data to the configuration optimizer. The configuration optimizer uses the data and its tuning models to ⑥ calculate an optimized configuration. Based on the recommended new table configuration ⑦, the controller ⑧ applies the configuration on the target DBMS. Here, each chunk's determined configuration can be applied asynchronously to reduce the overhead [16].

2.3 Implications of Configuration Decisions

To demonstrate the impact of different table configurations on the memory consumption and the runtime of a scan operation, as well as for the evaluation of our optimization approaches (see Section 5), we use the real-world dataset of a transportation network company (TNC) as a running example. The dataset consists of 400 million observed locations of drivers for three consecutive months in the City of Dubai [41] (raw size 15.9 GB). In comparison to other passenger transportation datasets (e.g., NYC Taxi Rides [44]), the dataset has a significantly finer granularity as the position of a driver is tracked multiple times per minute. Besides the timestamp, latitude, longitude, and the driver's identifier, a status attribute is tracked for each observed location. The status indicates the driver's current state (free or occupied). All attributes are stored as integers. Based on the insertion order, a certain temporal ordering of the sample points exists, but we cannot guarantee that the timestamp column is sorted due to transmission problems and delayed transmissions.

The selection of tuning configurations is a trade-off between performance and costs (e.g., memory consumption). In Figure 3, we visualize the implications of different tuning configurations measured in the research database *Hyrise*, which has comparable performance to other database systems [15]. We compare the memory consumption and runtime performance of isolated executed table scan operations on a single column for the different chunk ordering options (incl. unsorted) and encoding configurations (incl. unencoded). The set of encodings includes the four compression approaches (i) LZ4 encoding, (ii) dictionary encoding, (iii) frame-of-reference encoding, and (iv) run-length encoding.

In Figure 3-I, we can observe that the different tuning options significantly impact a scan operation's performance and memory consumption. For the *LessThanEquals* scan on the id column, the tuning option with the lowest runtime is about 1 000 times faster compared to the one with the highest runtime. Concerning the memory consumption, we can observe that the tuning option with the highest memory consumption needs about 500 times more space compared to the one with the lowest data footprint. Additionally, we can observe in Ⓐ that the sorting of a column has an impact on performance and, for some compression techniques (e.g., run-length encoding), also on memory consumption. A scan operation on a sorted column has a better performance compared to an unsorted one. Moreover, Ⓑ demonstrates that the sorting order of other columns can also have an impact on the data footprint and performance. In Figure 3-II, the same scan operation is executed

on the longitude column. Based on the changed data characteristics, the runtime performance and memory consumption of several tuning options changed ©. Consequently, we have to consider the specific data characteristics of a column in the optimization process. In Figure 3-III, we can observe that the given workload influences the effectiveness of the different tuning options. Based on the measurements, we can summarize that various aspects determine the efficacy of specific tuning options. Overall, Figure 3 motivates that the selection of suitable tuning options for a particular application context can be highly beneficial for memory consumption and performance. Further, the selection of performance and cost balancing configurations consisting of mutually depending tuning options is challenging due to the number of potential setups. Thus, it is difficult for database administrators to estimate the impact of specific tuning decisions [11]. Especially for workloads with a mix of different types of queries, the impact of individual choices is hard to predict and hence, the overall tuning is hard to optimize.

3 PROBLEM DESCRIPTION AND LP-BASED SOLUTION APPROACHES

In this section, we describe the problem of database systems to determine memory-efficient table configurations for spatio-temporal data (Section 3.1). In Section 3.2, we introduce a general linear programming (LP) model that solves the specified problem. Further, we present heuristic solutions based on segment-specific costs (Section 3.3) using specialized LP models with (Section 3.4) and without (Section 3.5) sorting dependencies. In Section 3.6, we show how our models can be adapted to include database-specific restrictions.

3.1 Problem Definition

We consider a table with a set of attributes N and a set of chunks M (cf. Section 2). The problem is to find a valid table configuration for a given set of available storage units B and a given workload consisting of Q different query templates q, which occur with frequency f_q, such that the overall performance is maximized by minimizing the workload's total execution time. A valid table configuration consists of a (i) sorting, (ii) data placement, (iii) compression, and (iv) index configuration for all columns n within each *chunk* m. Consequently, for each segment (n, m) with $n \in N, m \in M$, we have to select a configuration from sets of available compression E, indexing I, storage B, and sorting O options. Note, these sets also include basic options, i.e., data can be unsorted, unencoded, or not indexed. A notation table is provided in the Appendix.

As the required DRAM capacities of spatio-temporal data represent a significant cost factor or even exceed the available resources of modern systems, partitions of the data have to be transferred to slower and less expensive storage locations. To reflect these properties in the model, we assume a given storage budget G_b for each storage medium $b \in B$, which must not be exceeded. The size of a segment (m, n) with configuration e, o, i (with $e \in E, o \in O, i \in I$) is described by the parameters $\phi_{m,n,e,o,i}$ under the assumption that the used storage medium has no impact on the needed amount of bytes to store a segment. Note, $\phi_{m,n,e,o,i}$ also includes the memory consumed by the index (if an index is applied on the segment).

For a chunk $m \in M$ we consider potential joint configurations k from a given set of feasible options K. An option k characterizes *combinations* of configurations on a segment level, i.e., by $e_{m,n,k} \in E$, $o_{m,n,k} \in O$, $i_{m,n,k} \in I$, and $b_{m,n,k} \in B$ we denote encoding, sort, index, and data placement decisions for column $n \in N$.

Further, as we are focusing on spatio-temporal range queries and trajectory-based queries, each query template can be described as a composition of various *scan operations*, where the set S_q returns all scan operations s of a query template q, $q \in Q$. For scan operation s of template q the corresponding costs for chunk m under a specific tuning configuration k are denoted by parameters $c_{q,m,s,k}$, $s \in S_q$, $q \in Q$, $m \in M$, $k \in K$.

3.2 General Model with Chunk-Based Configuration Dependencies (CCD)

First, we consider a general model with chunk-based configuration dependencies (CCD), which represents a solution approach accounting for full cost dependencies within a chunk. In this model, the costs associated with a segment can depend on *all* specific configuration decisions of all *other* segments. This enables the model to particularly include multi-attribute indexes (e.g., k-d tree on latitude and longitude) or multi-attribute sorting options (e.g., space-filling curves) as long as (i) the number of considered configurations is tractable and (ii) the necessary data is at hand.

In the CCD model, we use the binary variables $x_{m,k}$, to express whether for a chunk $m \in M$ the joint configuration $k \in K$ is chosen. The objective of the CCD model is to minimize the cost (runtime) for a given workload, cf. Q, f, over all x variables (denoted by $\vec{x}$)

$$min_{\vec{x}} \sum_{m \in M, k \in K} x_{m,k} \cdot \sum_{q \in Q, s \in S_q} f_q \cdot c_{q,m,s,k} \tag{1}$$

subject to the $|B|$ budget constraints, which guarantee that the accumulated memory consumption of all segments (m, n) with their selected configurations e, o, i, b (cf. $\phi_{m,n,e,o,i}$) does not exceed a tier's budget G_b, i.e., $\forall b \in B$ we use,

$$\sum_{m \in M, k \in K} x_{m,k} \cdot \sum_{\substack{n \in N: \\ b_{m,n,k} = b}} \phi_{m,n,e_{m,n,k},o_{m,n,k},i_{m,n,k}} \leq G_b. \tag{2}$$

To get a unique configuration option for each chunk m we use

$$\sum_{k \in K} x_{m,k} = 1 \quad \forall m \in M. \tag{3}$$

The CCD model (1)-(3) is linear and can be *optimally* solved using standard solvers. Naturally, the model's complexity and the required input is driven by the size of K, which can quickly become large when exhaustive combinations of tuning options are used. In this context, we recall that the options within K should be chosen by taking domain-knowledge into account such that only *reasonable* configurations are considered. Moreover, we note that within these options for a certain chunk we can reduce the number of options $|K|$ by excluding all options that are *dominated* by another option (with smaller required memory and better scan costs).

The CCD model can be, e.g., used in specialized domain settings, where it is crucial to be able to account for complex tuning dependencies. In applications with less complex dependencies, simpler models that rely on segment-based costs can be more suitable. Those are discussed next.

3.3 Segment-based Cost Estimation

Cost estimations for different configurations are a crucial aspect. To determine them on a segment level, we consider the scan operations

of a query and their execution order, which is defined by the query optimizer. Based on the *Hyrise* query optimizer implementation, the order of the scan operations is determined by the operations' selectivity value, starting with the lowest selectivity value. To consider that a scan operation s of a query template q (executed after a previous scan operation of the same query template) operates only on a subset of the data, we introduce a scan factor $\omega_{q,s}$. This factor $\omega_{q,s}$ is determined by the ordered sequence of (consecutively executed) scan operations of a query template. To determine $\omega_{q,s}$, we consider the selectivity factor of the j-th operation of a query template q denoted by $\tilde{\omega}_{q,j}$. By default, the selectivity factor of the *first* scan operation of a query template is defined as $\tilde{\omega}_{q,1} = 1$. Accounting for the combined selectivities of consecutive operations within a query template q for its scan operation s with operation order $J_{q,s} \in \{1, ..., |S_q|\}$ we obtain the scan factor, $s \in S_q$,

$$\omega_{q,s} = \prod_{j=1,...,J_{q,s}} \tilde{\omega}_{q,j}. \quad (4)$$

Besides the selectivity, each scan operation s of query template q, $q \in Q$, $s \in S_q$, has the following attributes: (i) the scanned column $n_{q,s}$, (ii) the frequency f_q, and (iii) the type of the scan operation (e.g., between scan, less than equal scan, equal scan). The costs of the scan operations on segment n of chunk m (aggregated over all scan operations s that access n, i.e., $s \in S_q : n_{q,s} = n$, and weighted by the associated query frequency f_q) are denoted by $c_{m,n,e,o,i,b}$ and determined by the segment's encoding $e \in E$ and index decision $i \in I$ as well as the data placement decision $b \in B$ and ordering decision $o \in O := \{0\} \cup N$, where O includes all columns of the table plus the unsorted option ('0'). For $m \in M, n \in N, e \in E, o \in O, i \in I, b \in B$, we define:

$$c_{m,n,e,o,i,b} := \sum_{\substack{q \in Q, s \in S_q: \\ n_{q,s}=n}} f_q \cdot p_{q,s,e,o,i} \cdot a_{m,q,s} \cdot \omega_{q,s} \cdot u_{q,s,e} \cdot \tau_{e,i,b}. \quad (5)$$

The parameter $p_{q,s,e,o,i}$ defines the measured performance of the scan operation $s \in S_q$ of query $q \in Q$ executed as isolated scan operation on column $n_{q,s}$ stored in DRAM if for the entire column encoding $e \in E$, index decision $i \in I$, and for all chunks the ordering decision $o \in O$ are applied. Further, in (5) we use the successive scan penalty $u_{q,s,e}$ as we observed that consecutive scans are slower than single scan operations, depending on the applied compression technique e. To reflect this observation and to adopt the measured isolated scan performance $p_{q,s,e,o,i}$ of the benchmark queries (cf. Section 2), we multiply $p_{q,s,e,o,i}$ of all consecutive scan operations with the fixed parameter $u_{q,s,e}$ for each value $e \in E$. This penalty value u is database-specific and can be measured with a simple set of benchmark queries.

Based on statistics and filters maintained by database systems, entire chunks can be pruned during query execution to increase the scan performance [16]. This is especially the case for temporal range queries, which only scan specific sections of the data. For that reason, we introduce the parameter $a_{m,q,s}$, cf. (5), which describes the proportional size of segment $(m, n_{q,s})$ in relation to the amount of data scanned within a complete column scan on column $n_{q,s}$. As the costs for pruned chunks are neglectable, for not accessed chunks we let $a_{m,q,s} := 0$. For accessed chunks m, we define $a_{m,q,s}$ by their relative share of actually scanned chunks, i.e., by 1 divided by the number of not pruned chunks. Additionally, each storage medium has a penalty $\tau_{e,i,b}$, which reflects the difference between the measured access performance on DRAM and the access times on storage medium b, which can also depend on the index and encoding decision. Correspondingly, we multiply the estimated costs for an operation on a segment with the storage penalty $\tau_{e,i,b}$.

3.4 Special Case: Segment-Based Model with Sorting Dependencies (SMS)

The SMS model allows to solve the configuration problem with segment-based costs, cf. Section 3.3. It still allows to include intra-chunk dependencies between segments with regard to the chunk-based ordering decision. The corresponding segments' costs are determined based on (5) accounting for a chunk's specified sorting column. This enables the model to reflect the impact of the order decision on memory usage and scan performance, see Section 2.3.

For the specialized SMS model, we use an adapted LP formulation, cp. (1)-(3). The objective to minimize the costs is given by

$$min_{\vec{x},\vec{y},\vec{z}} \sum_{m \in M, n \in N, e \in E, o \in O, i \in I, b \in B} x_{m,n,e,o,i,b} \cdot c_{m,n,e,o,i,b}, \quad (6)$$

where the binary variables $x_{m,n,e,o,i,b}$ describe whether a certain tuning configuration, cf. $e \in E, o \in O, i \in I, b \in B$, for segment $n \in N$ of chunk $m \in M$ is used ('1') or not ('0'); $\vec{y}$ and $\vec{z}$ are auxiliary variables (see below). Similar to (1), the overall cost is calculated as the sum of the costs c of all selected segment configurations, cf. (6).

To ensure valid table configurations, we define different sets of constraints. We distinguish between model-specific and database-specific constraints. The model-specific constraints define general requirements for the determined table configurations. Database-specific constraints to incorporate technical restrictions and limitations of different database systems are discussed in Section 3.6.

For the SMS model, we define three types of model-specific constraints. The first one describes tiering-specific budget constraints, cp. (2), that defines that the accumulated memory consumption of all segments (m, n) with their selected configurations e, o, i on tier b (cf. $\phi_{m,n,e,o,i}$) does not exceed a tier's budget G_b, i.e., $\forall b \in B$,

$$\sum_{m \in M, n \in N, e \in E, o \in O, i \in I} x_{m,n,e,o,i,b} \cdot \phi_{m,n,e,o,i} \leq G_b. \quad (7)$$

Secondly, to guarantee that for each chunk m a unique ordering option is chosen, we use binary variables $y_{m,o}$, which describe whether ordering o is used for chunk m, i.e.,

$$\sum_{o \in O} y_{m,o} = 1 \quad \forall m \in M. \quad (8)$$

Thirdly, the binary variables $z_{m,n,e,i,b}$ ensure a unique index-encoding-tiering combination for chunk m's segment n,

$$\sum_{e \in E, i \in I, b \in B} z_{m,n,e,i,b} = 1 \quad \forall m \in M, n \in N. \quad (9)$$

The chunk variables y and segment variables z shall together specify the configuration $x_{m,n,e,o,i,b} = y_{m,o} \cdot z_{m,n,e,i,b}$. To express the x variables linearly we use the following auxiliary coupling constraints $\forall m \in M$, $n \in N$, $e \in E$, $o \in O$, $i \in I$, $b \in B$,

$$x_{m,n,e,o,i,b} \geq y_{m,o} + z_{m,n,e,i,b} - 1 \quad (10)$$

$$x_{m,n,e,o,i,b} \leq y_{m,o} \quad (11)$$

$$x_{m,n,e,o,i,b} \leq z_{m,n,e,i,b}. \quad (12)$$

Note, (6) is minimized over all families of variables x, y, z. The LP ensures *optimal* allocations. In case a chunk-tiering concept is

applied, we want to ensure that all segments of a chunk are stored on the *same* storage medium b, $b \in B$. For this purpose, we use (optionally) use the constraints, $\forall m \in M, n \in N$,

$$\sum_{e \in E, i \in I, b \in B} b \cdot z_{m,1,e,i,b} = \sum_{e \in E, i \in I, b \in B} b \cdot z_{m,n,e,i,b}. \quad (13)$$

3.5 Relaxed Model: Independent Segment Effects (ISE)

To heuristically solve the SMS model, we use a relaxation regarding the ordering dependencies of the cost effects between segments. In this simplified model, we only account for whether a certain chunk's segment is sorted ('1') or not ('0'). Hence, instead of the full set of ordering options $O = \{0\} \cup N$ for each chunk, we use the *simplified* binary set $\{0, 1\}$ of available ordering options for each chunk's segment. For the unsorted option ('0'), the rows' order is set by the insert sequence and we use the costs $c_{m,n,e,0,i,b}$, cf. (5). If a segment (m, n) is sorted, we use $c_{m,n,e,n,i,b}$. With this formulation, we reduce the complexity by abstracting the sorting decision's intra-chunk effects. Thus, the model approximates the exact implications on the memory footprint and scan performance caused by sorting a chunk by column n (cf. Section 2.3).

Compared to the SMS model, the relaxed ISE model has less variables and constraints. Specifically, we use a smaller family of binary decision variables $x_{m,n,e,o,i,b}$, where the ordering option only reflects the binary set $o \in \{0, 1\}$. The variables y and z, cf. (8)-(12), are not required. The objective of the ISE model is, cp. (6),

$$min_{\vec{x}} \sum_{\substack{m \in M, n \in N, e \in E, \\ o \in \{0,1\}, i \in I, b \in B}} x_{m,n,e,o,i,b} \cdot c_{m,n,e,o \cdot n,i,b}, \quad (14)$$

where we use $o \cdot n \in O$ to include the costs defined in (5) via $c_{m,s,e,o \cdot n,i}$. The budget constraints are, cp. (7), $\forall b \in B$,

$$\sum_{m \in M, n \in N, e \in E, o \in \{0,1\}, i \in I} x_{m,n,e,o,i,b} \cdot \phi_{m,n,e,o \cdot n,i} \leq G_b. \quad (15)$$

Note, the relaxed use of c and ϕ in (14)-(15) only approximates the exact values. Further, we directly use x to ensure that for each chunk m at most one column is sorted, cp. (8),

$$\sum_{n \in N, e \in E, i \in I, b \in B} x_{m,n,e,1,i,b} \leq 1 \quad \forall m \in M \quad (16)$$

and that for each chunk m's segment n, a unique configuration of e, o, i, and b is chosen, i.e., $\forall m \in M, n \in N$,

$$\sum_{e \in E, o \in \{0,1\}, i \in I, b \in B} x_{m,n,e,o,i,b} = 1. \quad (17)$$

The LP solutions of ISE are optimal. To obtain the same tiering for a chunk's segments we use, $\forall m \in M, n \in N$, cp. (13),

$$\sum_{\substack{e \in E, o \in \{0,1\}, \\ i \in I, b \in B}} b \cdot x_{m,1,e,o,i,b} = \sum_{\substack{e \in E, o \in \{0,1\}, \\ i \in I, b \in B}} b \cdot x_{m,n,e,o,i,b}. \quad (18)$$

3.6 Integration of Database-Specific Configuration Constraints

Additionally, we allow for database-specific constraints to the model-specific constraints, which enable the models to reflect certain properties of various database systems. The values for these constraints vary between databases and define combinations of indexing and encoding decisions that are incompatible. For *Hyrise*, secondary indexes require dictionary encoded segments as they exploit the dictionary in order to improve space efficiency [17]. Consequently, indexes on all non-dictionary segments are forbidden. In the ISE and SMS model, this is realized via the constraint, $\forall m \in M, n \in N, e \in E, s \in S, i \in I, b \in B$,

$$x_{m,n,e,s,i,b} \leq v_{e,i}, \quad (19)$$

where the binary parameters $v_{e,i}$, $e \in E$, $i \in I$, describe whether an index i is valid (=1) for a specific encoding e or not (=0). For the CCD model, constraint (19) can be directly satisfied by considering only corresponding valid configuration options within the set K. Similar database-specific constraints can also be treated like that.

4 EXTENSIONS

This section introduces two different enhancements, which are presented for the ISE and SMS models described in the previous section. The first extension (Section 4.1) enables the internalization of reconfiguration costs required for an updated table configuration (given an existing one). The second one addresses a robust configuration selection against multiple potential workload scenarios (Section 4.2), i.e., we look for allocations that are not optimized for one workload but "near-optimal" for various of them.

4.1 Minimal-Invasive State-Dependent Reconfigurations

Real-world workloads typically change over time. As a result, current data placements and configuration decisions might be outdated and have to be adapted to enable an optimized performance. However, the reorganization of configurations is costly and time-consuming [50]. The challenge is to identify 'minimally invasive' reallocations, which have a significant impact compared to their costs [25]. We extend the ISE model to exemplary show how to endogenize reconfiguration costs. We assume a current *configuration state*, e.g., characterized by parameters $\bar{x}_{m,n,e,o,i,b} \in \{0, 1\}$, which characterize the (old) configuration decisions. In case a segment (m, n) is transferred from a current configuration $\bar{\eta}_{m,n} := (\bar{e}_{m,n}, \bar{o}_{m,n}, \bar{i}_{m,n}, \bar{b}_{m,n})$ to a new configuration (e, o, i, b), we generally assume given reconfiguration costs $\Delta_{m,n,e,o,i,b}(\bar{\eta}_{m,n})$. Finally, to model reconfiguration costs, we replace the ISE objective as follows

$$min_{\vec{x}} \sum_{\substack{m \in M, n \in N, e \in E, \\ o \in \{0,1\}, i \in I, b \in B}} c_{m,n,e,o \cdot n,i,b} \cdot x_{m,n,e,o,i,b} + \alpha \cdot \sum_{\substack{m \in M, n \in N, e \in E, \\ o \in \{0,1\}, i \in I, b \in B}} \Delta_{m,n,e,o \cdot n,i,b}(\bar{\eta}_{m,n}) \cdot x_{m,n,e,o,i,b}. \quad (20)$$

The additional cost term in (20) (governed by the penalty factor α) prevents that configurations are widely reorganized while the performance increase is only marginal. Note, while for α=0, we obtain the original performance-maximizing model without reconfiguration costs, for large α any costly configuration changes will be prevented. The other constraints of the basic ISE model remain unchanged (cf. (i) the budget constraint, (ii) at most one sorted column, (iii) unique configurations for each segment). Additional variables or constraints are not required. Hence, the ISE model with reconfiguration costs can still be solved via standard solvers.

4.2 Robustness Against Different Potential Workload Scenarios

In general, spatio-temporal data characteristics and workloads are continuously influenced by the environment [54]. As future workloads are not entirely predictable, the performance can be negatively affected if the actual workload differs from the predicted one. This is a potential weakness of existing approaches that are only optimized for a specific workload. Hence, it is crucial to take potential workload scenarios into account to obtain a robust performance. Potential future workload scenarios (characterized by query frequencies) can be determined, e.g., (i) based on previously observed (seasonal) workloads or (ii) forecasts and their confidence intervals as well as (iii) domain expert inputs. Given potential scenarios, data allocations and tuning configurations can be optimized to maximize expected performance (risk-neutral) or more robust (risk-averse) objectives (cf., e.g., worst case, expected utility, mean-variance criteria, etc.). In particular, such risk-aware objectives seek to avoid the risk of poor performances. To be able to deal with diverse scenarios one is willing to sacrifice a certain share of the best possible expected performance.

We consider the set W of potential workload scenarios w, e.g., with probability P_w, $w \in W$, where $\sum_w P_w = 1$. We assume that a workload scenario w is characterized by a set of queries with given frequencies $f_q^{(w)}$ (within a certain time span). Hence, the workloads costs c defined in (5) generalize to multiple workloads $w \in W$ as, $m \in M, n \in N, e \in E, o \in O, i \in I, b \in B$,

$$c^{(w)}_{m,n,e,o,i,b} := \sum_{\substack{q \in Q, s \in S_q: \\ n_{q,s}=n}} f_q^{(w)} \cdot p_{q,s,e,o,i} \cdot a_{m,q,s} \cdot \omega_{q,s} \cdot u_{q,s,e} \cdot \tau_{e,i,b}. \quad (21)$$

For instance, a worst-case optimization for the ISE model reads as follows. Using the non-negative real-valued variable Z for the worst-case performance costs over all scenarios $w \in W$, we minimize

$$min_{\vec{x},Z} \quad Z \quad (22)$$

subject to the constraints (15)-(17) and the new ones, $\forall w \in W$,

$$\sum_{\substack{m \in M, n \in N, e \in E, \\ o \in \{0,1\}, i \in I, b \in B}} c^{(w)}_{m,n,e,o \cdot n,i,b} \cdot x_{m,n,e,o,i,b} \leq Z. \quad (23)$$

Note, the model (22)-(23) remains linear and is independent of the distribution P_w. As we only have one additional variable and $|W|$ new constraints, this extended/robust version of our ISE model has low overhead and the number of potential scenarios to be considered can be chosen such that the model remains tractable or the runtime does not exceed a targeted limit. Further, the proposed approach can also be directly used within the SMS or CCD model.

5 EVALUATION

In this section, we present the evaluation of our LP models based on the real-world dataset already introduced in Section 2.3. A description of the experimental setup is provided in Section 5.1. In Section 5.2, we use end-to-end experiments to compare our models against rule-based greedy heuristic approaches. Further, we briefly study and discuss the impact of the reconfiguration extension described in Section 4.1 (Section 5.3), the scalability of the models (Section 5.4), and the limitations of our approach (Section 5.5).

5.1 Experimental Setup

All end-to-end measurements have been executed on a server equipped with Intel Xeon Platinum 8180 CPUs (2.50GHz). For the benchmark queries and the evaluation of the determined configurations, we use the research database *Hyrise*. We define the input parameters based on the supported encoding and indexing properties of the database. The set of available encodings E consists of five options introduced in Section 2.3. As secondary indexes, we use the approach of Faust et al. [17]. Furthermore, we include multi-column indexes, which are implemented in Hyrise (cf. *compound group-keys* [18]), in the evaluation to demonstrate the capabilities of the CCD model. Both approaches leverage a segment's dictionary to increase space efficiency. Consequently, we have to ensure that indexes are only allowed on dictionary encoded segments (cf. Section 3.6). Besides 5 singles column indexes, we consider 46 *relevant* MCIs with lengths of 2-4 (cf. [23]). For combinations of the 4 tuning dimensions, we considered about $K = 120\,000$ options per chunk.

We partition the data into ten chunks containing one million observed locations each. We use two storage devices, DRAM and an Intel P4800X SSD. The τ value for each encoding is determined by the mean performance difference of a set of scan operations executed with different selectivity values on both storage devices. As the tiering of segments is based on *UMap* (cf. Section 2.1), we have to define a *UMap* page and buffer size. For the user-space page management, *UMap* uses a buffer of pages in DRAM [36]. To ensure that the data has to be read from the slower datastore and avoid caching effects, we selected a small buffer size of 1000 pages and a page size of 128 KiB. Note, our models are able to consider further index structures, storage devices, or encoding approaches for specific application scenarios. The LP framework is of general nature and ensures optimal configurations for a given set of tuning options and associated cost parameters. Naturally, this set of tuning options can be flexibly defined and varied.

To compute the table configurations, we used a server equipped with Intel Xeon E7-4880v2 CPUs (2.50GHz). Our different models are implemented in *Pyomo*, a Python-based optimization modeling language [9, 21]. To solve the LP models, we used the *Gurobi Solver* with 16 parallel threads (no relaxed optimality gaps or time limits). Based on the dataset of a TNC, cf. Section 2.3, we defined a workload Q. The workload is designed to represent the characteristic of spatio-temporal workloads and includes domain-specific access patterns of TNCs (e.g., order dispatching, demand predictions). As our models are of general nature and perform workload-driven optimizations, the approaches do not focus on a specific workload and can be applied to different workloads and datasets. The workload consists of six query templates. The first three query templates select the trajectory data of a specific set of drivers: all trips of a group of drivers (0.01% of all values) with the status free (q_0, 15% of the queries in the workload), all trips of a group of drivers (1% of all values) in a specific area (q_1, 10%), and all trips of a group of drivers (1% of all values) in a specific timeframe of about 40 days (q_2, 15%). Further, q_3 (15%) selects all trips of drives with the status free in a timeframe of 20 days and a specific region; q_4 (25%) queries all trips of drivers in a relatively large area (20 by 20 km) in the most recent six hours. Finally, q_5 (20%) returns all trips in a small region of 500 by 500 meters for a timeframe containing 50 percent of the data.

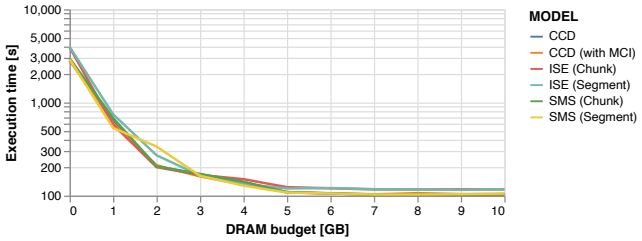

Figure 4: End-to-end measured runtime performance of the table configurations determined by the different LP models for the given workload, a fixed budget of 3 GB on SSD, and increasing DRAM budgets (incl. multi-column indexes (MCI)).

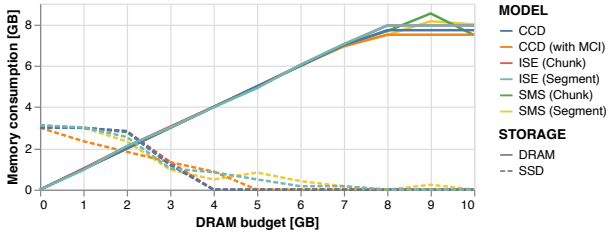

Figure 5: DRAM (solid line) and SSD (dashed line) memory consumption of the table configurations determined by the different LP models for the given workload, a fixed budget of 3 GB on SSD, and increasing DRAM budgets.

5.2 End-to-End Results for SMS, ISE & CCD

As displayed in Figure 4, the three LP models are able to improve joint table configurations for the given workload. Overall, we can observe that there are high optimization potentials for fine-grained configuration decisions, especially for lower memory budgets. After a specific DRAM budget value, the performance only increases slightly or stagnates after most of the segments are stored in the main memory. Compared to the SMS benchmark results, we can observe that the ISE model is able to determine competitive table configurations, especially for larger memory budgets. However, without detailed information about intra-chunk effects, the chunk-based and segment-based ISE models have a 12% decreased performance compared to the other approaches. Also, for the initial case, where all data is stored on SSD, the ISE models have 30% performance decrease. The ISE model cannot consider the effects of a specific sorting column on other columns' data characteristics (e.g., number of identical values in succession). These characteristics have an increased impact on the compression rate of compression approaches like run-length encoding or frame-of-reference encoding, which are used in particular for low memory budgets. Further, as the SMS and CCD model without multi-column indexes (MCI) results coincide, it is verified that the SMS model is equivalent to the CCD model if multi-column optimizations are not considered.

In Figure 5, we can observe that the different LP models are able to utilize the available DRAM capacities efficiently. Also, the models satisfy the given memory limitations. In contrast to the chunk-based optimizations approaches, we observe that segment-based approaches store infrequently or never accessed segments even for higher DRAM budgets on SSD to generate more space in DRAM for further optimizations. Finally, we can observe that the SMS and CCD model reached a configuration at 8 GB where no further improvement can be achieved (with more DRAM budget).

In this experiment, segment-based approaches have no significant advantage compared to chunk-based approaches. The reason for that is that our workload accesses 92% of all segments. For workloads that query only a limited set of segments, we can expect better performances for the segment-based approach as less DRAM budget has to be used for never accessed data. By storing infrequently accessed segments on the slower storage, light-weight compression techniques or additional index structures can be applied for frequently accessed segments in the main memory.

5.2.1 Rule-Based Tuning Heuristics. We seek to evaluate the determined table configurations of our LP models against existing approaches. As a common standard approach, we implemented two rule-based greedy heuristics. Similar to the SMS model (cf. Section 3.4), the heuristic includes intra-chunk dependencies by taking into account the specific sorting column. As a valid table configuration requires that one sorting option $o \in O$ is selected for each chunk, we must integrate this constraint into the selection process of the base configuration. Therefore, we apply a two-phase approach. In the first phase, we determine a base configuration with minimal memory consumption by selecting the tuning configuration with the lowest memory consumption for each segment. Here, the segment with the lowest memory consumption defines the sorting configuration for the entire chunk. In the second phase, we calculate the benefit $r_{m,n,e,o,i,b}$ for each tuning option and adopt the table configuration iteratively based on the calculated benefits and the available memory budget. For each segment (m, n) and tuning option $e \in E$, $o \in O$, $i \in I$, $b \in B$, we define r as ($\alpha \geq 0$):

$$r_{m,n,e,o,i,b} = 1/\left(\phi_{m,n,e,o,i} \cdot \left(\sum_{\substack{q \in Q, s \in S_q: \\ n_{q,s}=n}} c_{m,n,e,o,i,b}\right)^{\alpha}\right). \qquad (24)$$

To calculate the costs $c_{m,n,e,o,i,b}$ of the scan operations, we use the same equation as for the LP models, cp. (5). The α value is a factor to define the proportional balancing of the memory consumption and runtime performance within the objective. A similar approach is used by Valentin et al. [45] to determine optimized index configurations. We select the tuning option with the highest benefit compared to the currently applied configuration. Afterward, we check if the selected tuning option fits into the remaining storage budgets and change the configuration correspondingly if the change does not violate the remaining memory budget. The steps of the second phase are repeated until no more changes are possible. For the chunk-based approach, we additionally ensure that only entire data chunks can be transferred to a tier, cf. (13).

5.2.2 End-to-End Comparisons Against Greedy Heuristics. As displayed in Figure 6, the LP models outperform the greedy heuristics, cf. H(α), and can use the available memory budget more efficiently (cf. Figure 7). We observe that greedy heuristics struggle to compute configurations for scenarios with limited DRAM capacities. For a DRAM budget of 2 GB, the SMS model achieves a runtime of 200s, which is only 10% of the heuristics' runtimes. The heuristics need significantly more DRAM budget (7 GB) to achieve comparable performance results. We observe that the heuristics use the majority of the available DRAM resources to optimize single chunks or segments. Consequently, large parts of the data have to be stored

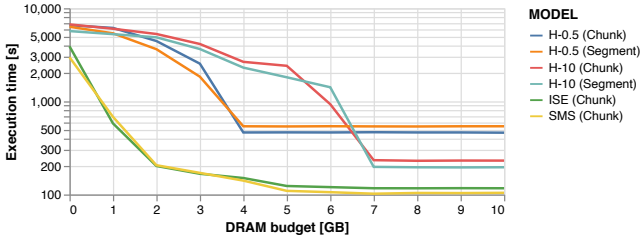

Figure 6: End-to-end measured runtime performance of the table configurations determined by heuristic approaches compared to the LP models for the given workload, a fixed budget of 3 GB on SSD, and increasing DRAM budgets.

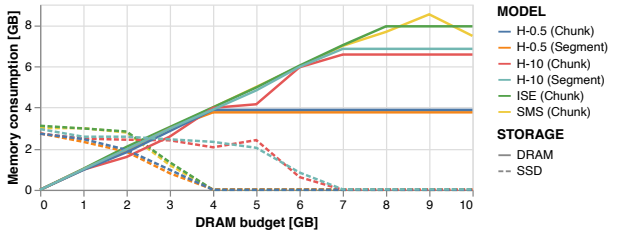

Figure 7: DRAM (solid line) and SSD (dashed line) memory consumption of the table configurations determined by heuristic approaches for the given workload, a fixed budget of 3 GB on SSD, and increasing DRAM budgets.

on the significantly slower SSD, even for higher DRAM capacities (cf. Figure 7). Additionally, the greedy heuristics' measurements show that the selection of the α value has a significant impact on performance and memory consumption. As the heuristics select the sorting configuration for each chunk in the initial phase, the sorting decisions can be sub-optimal for different budgets.

5.2.3 End-to-End Comparisons Against Existing Approaches. In this section, we compare our LP approach against existing solutions. A comparison is not straightforward as various joint optimization approaches focus on other aspects (e.g., materialized views, partitioning, or knob configurations). In Figure 8, we evaluate the SMS model regarding (i) the tiering dimension [48] and (ii) the consecutive joint tuning approach [25]. Concerning the tiering dimension, we re-implemented the *capacity mode* of *Mosaic's* presented linear optimization strategy (LOPT) [48]. This LP approach is based on Umbra and optimizes the data placement on columnar granularity. As *Mosaic* is optimized for data placement decisions on SSD and HDD devices, the cost model is based on the throughput and accessed data size. Furthermore, parallel scan operations on multiple devices are allowed. To represent the query processing in *Hyrise*, we adopted the cost estimation that scan operations are executed sequentially where each scan only processes the qualifying positions of the previous scan. An advantage of this cost model is that less detailed information about the workload (e.g., selectivity of queries) is required, and no benchmarking queries have to be executed. Based on the prerequisite that *Mosaic* requires all segments on one device to have the same encoding, we selected the best encoding configuration for the given benchmark setup. For a fair comparison, we partitioned the data to enable pruning during query execution and applied the same configuration for all segments of a column. Based

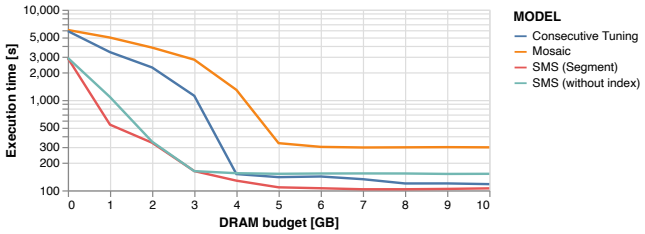

Figure 8: Comparison of the end-to-end measured runtime of the table configurations determined by *budget mode* of Mosaic, a consecutive tuning approach, and the SMS model.

on these restrictions, a relatively large amount of memory has to be used to store a single dictionary-encoded column in DRAM. As displayed in Figure 8, for small memory budgets, we can observe a comparably weak performance as significant parts of the available DRAM were unused by *Mosaic* as the remaining DRAM budget was not big enough to store another column. Furthermore, the model is not designed to optimize other aspects (e.g., sorting or indexing), which have a significant impact on the overall performance.

Furthermore, we compare the SMS model against the joint tuning approach [25], which proposes a heuristic to tune different dimensions in a subsequent manner for given budgets to be used in each step. We evaluated this approach as follows: Starting with an untuned base configuration, we choose a specific tuning order for our 4 considered dimensions. Given a budget, we solve our model for the first dimension by freeing the corresponding variables and fixing those for the other dimensions. The same is done for the remaining tuning dimensions. Instead of one LP, for [25], we solve 4 LPs of smaller size. Further, we iterated over all plausible tuning orders to not miss the best possible one. Comparing the results obtained, we determined the best configurations (displayed in Figure 8) by optimizing the encoding first, afterward the sorting and tiering decision, and the index configuration at the end. For the index selection, we reserved 10% of the available budget in the previous steps. We discover that our LP approach outperforms [25] by 15% for larger DRAM budgets and achieves up to 6.8 times better performance for more restrictive memory budgets. Naturally, the final results also depend on the distribution of the total budget (e.g., budget for indexes) to the 4 steps. Recall that this reveals another advantage of our model as the nontrivial optimization of the distribution of the budgets is included in our model.

5.2.4 Detailed Configuration Analysis. Figure 9 shows specific table configurations computed by the SMS model, the greedy heuristic (H-10), the CCD model with multi-column indexes, and the consecutive tuning approach. The configurations were computed for a budget of 3 GB on both storage mediums. For each configuration, the segments defined by the chunk (y-axis) and the column (x-axis) in the top area are stored in DRAM and the bottom ones on SSD. The color of each segment indicates the applied encoding and the circle in the middle represents the consumed memory. If an index is applied to a segment (indicated by a black rectangle or diamond in the top left corner), the consumed memory of the index is included in the size of the circle. A sorted column is represented by a black triangle in the top right corner of a segment. Note, for the CCD model, we force the tiering decision to be the same within a chunk.

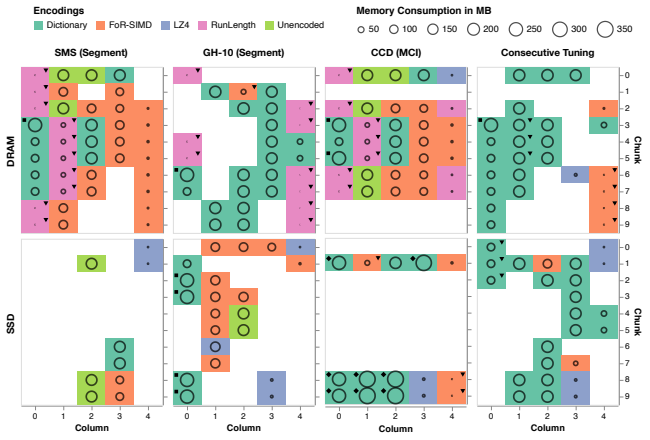

Figure 9: Comparison of configurations: (i) SMS model, (ii) H-10 heuristic with an α =10, (iii) CCD model with MCI, and (iv) consecutive tuning approach for a memory budget of 3 GB DRAM (top) and 3 GB SSD (bottom). A triangle in the top right corner of a segment indicates the sorted column of a chunk. A square in the top left corner indicates a single-column index for the segment; a diamond refers to an MCI.

We observe that the selected sorting configuration varies strongly between different approaches. As the greedy approach determines the sorting configuration based on the segments with the highest benefit in the first phase, this can be sub-optimal for lower budgets. Additionally, we can see that the greedy heuristic uses significantly more of the available DRAM budget to improve the performance of specific segments, which leads to an improved runtime for scan operations on these segments, but also consumes a significant part of the available main memory. In contrast, the SMS model distributes the available memory on more segments and selects compression techniques with higher compression ratios (e.g., frame-of-reference encoding) to store more segments in DRAM. The CCD model applies different multi-column indexes to mitigate the increased SSD access latency. In contrast, the CCD model uses single-column indexes in DRAM, which have a lower memory footprint. For the consecutive tuning approach, we observe weaknesses of the one-by-one execution of tuning steps. In the first step, the LP to optimize the encodings leverages the entire memory budget and applies encodings like dictionary encoding. Consequently, the tiering LP can only transfer a limited number of segments to DRAM.

5.3 Extension: Reconfiguration Costs

Based on the LP approach, minor workload or infrastructure (e.g., DRAM capacities) changes can lead to significant reconfiguration costs [25]. All individual tuning optimizations produce modification costs (e.g., changing the sorting order of a chunk consumes time and resources). To determine optimized configurations for the given input parameters, the models often apply numerous reconfigurations with only a minor impact on the overall performance. However, huge modification costs are not desirable in practice. By considering modification costs in the models (cf. Section 4.1) we are able to identify and perform only minimal-invasive modifications. There are various metrics to determine modification costs (e.g., reorganized data, estimated time, or selected by database administrator) [29, 43, 50]. For the evaluation, we defined for each modification a cost factor. A change of the applied sorting configuration has the highest costs and a reallocation of a segment the lowest. The database administrator can specify these costs per reconfiguration operation. For the given base configuration determined by the SMS model for a DRAM budget of 3 GB (cf. Figure 9), we increased the available memory budget by 1 GB and evaluated the configurations for different α values. The α enables the balancing of the trade-off between performance and reconfiguration cost and depends on different aspects (e.g., time interval between optimizations). The results of the end-to-end measured performance show that compared to the best possible α=0 solution, the α=1 configuration has only a 2.25% performance decrease with about 40% fewer reconfiguration costs. For α=5, there is a performance reduction of 17.6% and a reduction of about 78% of the reconfiguration costs.

5.4 Scaling of the LP-based Approaches

To analyze the scalability of the different models, we evaluated the runtime of the solver to compute the table configurations by scaling in the following dimensions: (i) number of chunks, (ii) number of scan operations that define the workload, (iii) number of storage devices, and (iv) the number of compression and index options. As benchmark setup, if not chosen differently, we use the settings described in Section 5.1 with $|Q| = 6$, $|\cup_q S_q| = 17$ scan operations, where $|M| = 10$, $|E| = 5$, $|N| = 5$, $|O| = 6$, $|I| = 2$, and $|B| = 2$. The memory budgets are defined as 3 GB for DRAM and 3 GB for SSD.

5.4.1 Impact of Data Size. Due to large spatio-temporal data volumes, scalability with regard to data size is crucial. In this context, we analyze our models' solve time for different numbers of chunks. Naturally, the complexity of all three models increases with the number of chunks, see Figure 10 (left top). The ISE & CCD model scale linear with the number of chunks, whereby the ISE model's solver times are orders of magnitude lower. For the given budgets, the segment-based ISE model needs 24ms for ten chunks and 613ms for 500 chunks to compute the configurations. In comparison, the CCD model needs 22s for ten chunks and a similar amount of time for 500 chunks. In contrast, the SMS models do not scale linearly with the cardinality of M; the models compute the table configurations in about 1.5s for ten chunks and 205s (500 chunks) for the segment-based approach and 2.5s for ten chunks and 413s (500 chunks) for the chunk-based approach.

5.4.2 Impact of Workload Size. We analyze the capabilities of the models to scale with the workload, cf. Q, S. For that reason, we evaluated the computation performance for an increasing number of scan operations. As shown in Figure 10 (bottom left), the workload size has no impact on the solver time. As described in Section 3, the amount of scan operation is only relevant for the calculation of the parameter c. The computation of the parameters can be executed highly parallel. Based on these observations, we can argue that the different models are capable of handling large workloads. Another observation is that the segment-based approaches have a faster runtime compared to their chunk-based equivalents.

5.4.3 Impact of the Number of Storage Devices. Next, we evaluate the impact of the number of storage devices on the solver time.

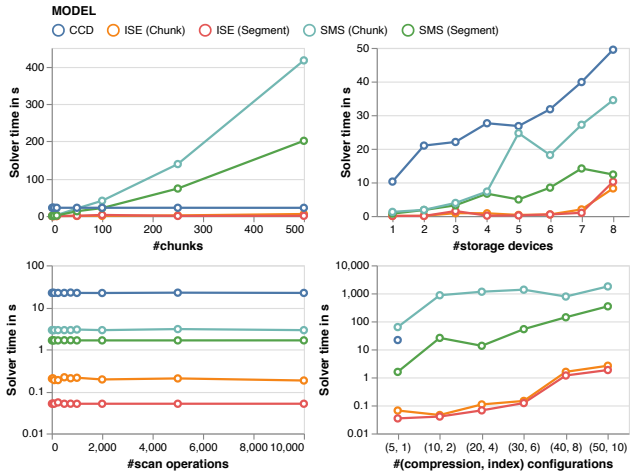

Figure 10: Comparison of solve times of 5 versions of our LP models for different numbers of tuning options: #chunks $|M|$ (top left), #scan operations $|\cup_q S_q|$ (bottom left), #storage devices $|B|$ (top right) and cardinality of the indexing $|I|$ and compression options $|E|$ (bottom right).

Modern systems feature a diversity of storage, from NVMe and SSD to network-interconnected memory and HDD [36]. The LP models are designed to support setups with multiple storage devices. We defined a base budget of 8 GB and divided this budget by the number of available storage devices. The storage penalty $\tau_{e,i,b}$ is set for all encodings (e) and index configurations (i) to a fixed value, defined by the storage device's number (b). As displayed in Figure 10, all five models can determine table configurations for infrastructures with eight different storage mediums in a suitable time (less than 50s). Here, we have to consider that the solver times are also depending on the specific setup (e.g., latency between storage devices, storage capacities).

5.4.4 Impact of the Number of Tuning Options. As modern database systems support various compression techniques and index implementations, we investigate the impact of the number of compression and index options on the solver time. For that reason, we generated additional values for up to 50 compression techniques and ten different index options. We apply no database-specific limitations for this benchmark so that correspondingly each index option can be applied in combination with each compression option. In Figure 10 (bottom right), we observe a slight increase in the solver time for the ISE model up to 2.7s (chunk-based and 1.9s (segment-based) for the setup with 50 compression techniques and ten indexes options. For the segment-based SMS model, the solver time increased from 1.5s for the *Hyrise* settings to 348s for the largest considered setup. Due to memory restriction, it was not possible to determine results for all setups for the CCD model. As already mentioned in Section 3.2, the number of possible chunk configurations $|K|$ can increase quickly. In this context, we further analyzed the applicability of CCD for reasonable numbers of $|K|$ based on randomly generated cost inputs. We evaluated that problems with selected six million tuning combinations (per chunk) can be solved in less than 60s, which, in general, shows the applicability of the model. Further, we recall that in specific applications, domain knowledge should be included to select only relevant tuning configurations, which particularly can include more complex concepts, such as multi-attribute indexes. Also, a hybrid optimization process could be possible, in which the CCD model compares the table configuration determined by the SMS or ISE model with several adopted versions (e.g., advanced sorting strategies).

5.5 Discussion, Insights, and Limitations

Our LP solutions allow us to reveal the main drivers of effective tunings and to infer recommendations for best practices (which columns to index, how to compress old/new chunks in case of small/large budgets, etc.). For example, we observe the following stylized patterns in optimal solutions. The sorting decision per chunk is based on the access frequency and memory saving of a segment. With increasing memory budgets, indexes are applied to segments with low selectivity queries, enabling sorting by another column. Based on the latency difference between the two storage devices, the determined models apply more heavy-weight compression in the main memory and compensate the latency with lightweight compression approaches on the slower SSD. Further, the possibility to tune chunks and their segments (accounting for their individual specifics) on a fine-grained level is, in general, heavily exploited, cf. Figure 9, and does contribute to improving overall performance.

Naturally, accurate cost parameter inputs are key in the model. Our cost estimations used in (5) provide a reasonable and viable starting point. More accurate estimations seem possible, e.g., using more sophisticated data-driven cost models, and would allow to further minimize the gap between model-based solutions and their actual end-to-end performance. However, more complex cost models may also add more overhead. Overall, the evaluation showed that fine-grained optimized table configurations are well-suited to reflect spatio-temporal access patterns in database systems. Nevertheless, although the model is of general nature and allows to attack complex and coupled tuning dependencies, there are the following limitations. First, we exploit the fact that in our usecase, the number of attributes $|N|$ is small, keeping the LP models tractable with regard to the number of variables and constraints. Second, we keep the model tractable by considering a comparably small number of indexes $|I|$ (cf. single-attribute indexes in the SMS model or distinguished multi-attributes in the CCD model). To optimize joint tuning problems for larger problems, hence, will require different most likely heuristic techniques. In this context, the proposed model may also serve as an upper bound reference to verify the quality of such techniques by providing optimal solutions for tractable setups of joint tuning problems.

6 RELATED WORK

In this section, we briefly discuss related work from the adjacent research fields of workload-aware compression and indexing optimization for database systems with a focus on spatio-temporal data management. Zhang et al. [54] noticed that the characteristics of spatio-temporal data change over time, which leads to a decreased efficiency of data structures (e.g., indexes). They proposed a time-decay model to monitor the data distribution and adopt the used indexing schema correspondingly. Schlosser et al. [43] introduced

a workload-driven index selection approach that builds on a recursive mechanism and accounts index interaction. In contrast to other approaches, the presented strategy also allows the scalable computation of index configurations for large workloads, cf. [24]. Kimura et al. [22] presented an index selection approach that selects viable secondary indexes and considers compressed alternatives for each index based on a given memory budget. This index size-aware approach uses a heuristic to prune index candidates and configurations of candidates.

All these approaches do not study whether using a more heavy-weight compression to make space for an additional secondary index can be beneficial. Compressing data is important for database systems to reduce the storage resources and allow efficient processing (e.g., SIMD instructions) as well as mitigating bandwidth bottlenecks. By analyzing light-weight integer compression, Damme et al. [12] found that sophisticated compression techniques can have significant impacts on both performance and compression ratios. The authors indicate that consideration of multiple dimensions is necessary to determine which technique is the best for a specific scenario. Based on similar observations and the fact that different decisions influence each other, we propose a joint optimization. For example, the sorting decision has a major impact on the data characteristics and, consequently, on the compression ratio of different approaches (e.g., run-length encoding). Raman and Swart [38] presented an approach that leverages successive similar values and correlations of sorted relations to improve the data compression. The sorting of a column has an impact on the runtime, which can lead to obsolete auxiliary data structures.

Abadi et al. [1] presented a decision tree-based approach to determine compression schema in C-store. The selection is based on workload and data properties but does not adapt to changing environments nor considers memory budgets. Boissier et al. [6] introduced a workload-driven selection of compression configurations with memory constraints for columnar databases. The approach uses a greedy heuristic to determine configurations based on data characteristics and estimated runtime performances based on regression models. Lang et al. [26] presented data blocks for HyPer. It selects compression schemes statically based on data characteristics for the data blocks. Mosaic [48] is a storage engine for scan-heavy workloads on relational database systems. It introduces a linear optimization to find data placement solutions that maximize I/O throughput for a given workload and budget. Mosaic places data on SSD and HDD with column granularity and considers different compression algorithms for each device. Boissier et al. [7] present a heuristic approach that evicts cold data to secondary storage. For a given DRAM budget and workload, the system determines a Pareto-optimal selection of columns that are stored in DRAM. Furthermore, Richly et al. [42] introduce an LP-based approach to optimize the sorting, compression, and index configuration of tables in DRAM. The proposed solution can neither determine configurations for tables that do not fit into main memory nor considers reconfiguration costs or robustness. To the best of our knowledge, no spatio-temporal storage system applies a joint optimization approach to determine workload-driven fine-grained storage configurations under budgets for different storage devices.

An approach to tune multiple mutually dependent features was introduced by Kossmann et al. [25]. The authors demonstrated the inter-dependencies of different feature tunings and presented an LP approach to optimize the order to tune features subsequently. Zilo et al. [56] also addressed the tuning order problem and classified the pairwise dependencies between options in non-dependent, unidirectional dependent, and mutually dependent. They described that mutually dependent ones should be optimized simultaneously as long as the problem complexity allows a joint optimization.

7 CONCLUSION

This paper demonstrates that workload-driven fine-grained configuration optimizations are a practical approach for improving the data management of spatio-temporal applications. The identification of an optimized table configuration for a given workload and storage setup is not a trivial task as the various potential tuning options influence each other. To our knowledge, this is the first approach to jointly optimize a table's sorting, indexing, tiering, and compression configuration. We introduced three LP models (cf. CCD, SMS, ISE) addressing cost dependencies at different levels of accuracy while allowing for trading their solve time. For an evaluation of a real-world spatio-temporal dataset, we have shown that our configuration decisions (requiring only seconds) are superior to existing rule-based heuristics. Our models achieve an up to 90% increased runtime performance for a given memory budget; a comparable runtime is obtained with up to 71% less required memory. Further, we discussed the scalability of our models and proposed extensions to include reconfiguration costs as well as robustness against different workloads. The SMS model reliably finds optimized tuning configurations if sorting dependencies are present. If those are not strong or shorter runtimes are in focus, we find that the relaxed ISE model is a suitable scalable alternative with competitive results.

In future work, we will work on improved data-driven cost models to better and more effectively estimate cost parameters for our model input. Another direction is to look for workload anticipations and risk-aware joint tuning configurations.

APPENDIX

Table 1: Notation Table

Sets	N	set of attributes n
	M	set of chunks m
	E	set of encoding types e
	I	set of indices i
	O	set of ordering options o
	B	set of tiering options b
	Q	set of queries q
	S_q	set of scan operations s of query q
	K	set of configuration options k (CCD model)
	W	set of potential scenarios w (robust model)
Parameters	f_q	frequency of query q
	$c_{m,n,e,o,i,b}$	scan costs for segment n of chunk m
	$\phi_{m,n,e,o,i}$	memory consumption for segment (n, m)
	G_b	memory budget for tier b
	$r_{m,n,e,o,i,b}$	benefit measure (greedy α-heuristic)
	$\Delta_{m,n,e,o,i,b}(\bar{\eta}_{m,n})$	reconfiguration cost given allocation state $\bar{\eta}$
Variables	$x_{m,n,e,o,i,b}$	decision if a configuration is active (SSD, ISE)
	$y_{m,o}$	decision (a chunk m's sorting, SSD)
	$z_{m,n,e,i,b}$	decision (a segment n's configuration, SSD)
	$x_{m,k}$	decision (a chunk m's configuration, CCD)

REFERENCES

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proc. ACM SIGMOD*. 671–682.

[2] Sanjay Agrawal, Nicolas Bruno, Surajit Chaudhuri, and Vivek R Narasayya. 2006. AutoAdmin: Self-Tuning Database SystemsTechnology. *IEEE Data Eng. Bull.* 29, 3 (2006), 7–15.

[3] Ana Carolina Almeida, Fernanda Baião, Sérgio Lifschitz, Daniel Schwabe, and Maria Luiza M Campos. 2021. Tun-ocm: A model-driven approach to support database tuning decision making. *Decision Support Systems* 145 (2021).

[4] Alexander Boehm. 2019. In-memory for the masses: enabling cost-efficient deployments of in-memory data management platforms for business applications. *Proc. VLDB Endow.* 12, 12, 2273–2275.

[5] Martin Boissier. 2022. Robust and Budget-Constrained Encoding Configurations for In-Memory Database Systems. *Proc. VLDB Endow.* 15, 4 (2022), 780–793.

[6] Martin Boissier and Max Jendruk. 2019. Workload-Driven and Robust Selection of Compression Schemes for Column Stores. In *EDBT*. 674–677.

[7] Martin Boissier, Rainer Schlosser, and Matthias Uflacker. 2018. Hybrid data layouts for tiered HTAP databases with pareto-optimal data placements. In *ICDE*. IEEE, 209–220.

[8] Michael Brendle, Nick Weber, Mahammad Valiyev, Norman May, Robert Schulze, Alexander Böhm, Guido Moerkotte, and Michael Grossniklaus. 2021. Precise, Compact, and Fast Data Access Counters for Automated Physical Database Design. *BTW* (2021).

[9] Michael L. Bynum, Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, and David L. Woodruff. 2021. *Pyomo–optimization modeling in Python* (third ed.). Vol. 67. Springer Science & Business Media.

[10] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-tuning database systems: a decade of progress. In *Proc. VLDB*. 3–14.

[11] Su Chen, Beng Chin Ooi, Kian-Lee Tan, and Mario A Nascimento. 2008. ST2B-tree: a self-tunable spatio-temporal B+-tree index for moving objects. In *Proc. ACM SIGMOD*. 29–42.

[12] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. 2019. From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms. *ACM Trans. Database Syst.* 44, 3 (2019), 9:1–9:46.

[13] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: a scalable, portable, and interactive index advisor for large workloads. *Proc. VLDB Endow.* 4, 6 (2011).

[14] Markus Dreseler. 2022. *Automatic Tiering for In-Memory Database Systems*. Ph.D. Dissertation. Universität Potsdam. https://doi.org/10.25932/publishup-55825

[15] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H choke points and their optimizations. *Proc. VLDB Endow.* 13, 8 (2020), 1206–1220.

[16] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. 2019. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *Proc. EDBT*. 313–324.

[17] Martin Faust, David Schwalb, Jens Krüger, and Hasso Plattner. 2012. Fast Lookups for In-Memory Column Stores: Group-Key Indices, Lookup and Maintenance. In *ADMS@VLDB*. 13–22.

[18] Martin Faust, David Schwalb, and Hasso Plattner. 2014. Composite Group-Keys: Space-efficient Indexing of Multiple Columns for Compressed In-Memory Column Stores. In *Proc. IMDM@VLDB 2014*. IMDM, 42–54.

[19] Zhenni Feng and Yanmin Zhu. 2016. A survey on trajectory data mining: Techniques and applications. *IEEE Access* 4 (2016), 2056–2067.

[20] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS.. In *CIDR*.

[21] William E Hart, Jean-Paul Watson, and David L Woodruff. 2011. Pyomo: modeling and solving mathematical programs in Python. *Mathematical Programming Computation* 3, 3 (2011), 219–260.

[22] Hideaki Kimura, Vivek R. Narasayya, and Manoj Syamala. 2011. Compression Aware Physical Database Design. *Proc. VLDB Endow.* 4, 10 (2011), 657–668.

[23] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms. *Proc. VLDB Endow.* 13, 12 (2020), 2382–2395.

[24] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395.

[25] Jan Kossmann and Rainer Schlosser. 2020. Self-driving database systems: A conceptual approach. *Distributed and Parallel Databases* 38, 4 (2020), 795–817.

[26] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proc. ACM SIGMOD*. 311–326.

[27] Ahmed R Mahmood, Sri Punni, and Walid G Aref. 2019. Spatio-temporal access methods: a survey (2010-2017). *GeoInformatica* 23, 1 (2019), 1–36.

[28] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Networks for Query Performance Prediction. *Proc. of the VLDB Endow.* 12, 11 (2019).

[29] Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, and Solomon Garber. 2018. NashDB: an end-to-end economic method for elastic database fragmentation, replication, and provisioning. In *Proc. ACM SIGMOD*. 1253–1267.

[30] Jean Damascène Mazimpaka and Sabine Timpf. 2016. Trajectory data mining: A review of methods and applications. *Journal of Spatial Information Science* 13 (2016), 61–99.

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

[32] Varun Pandey, Andreas Kipf, Dimitri Vorona, Tobias Mühlbauer, Thomas Neumann, and Alfons Kemper. 2016. High-performance geospatial analytics in hyperspace. In *Proc. ACM SIGMOD*. 2145–2148.

[33] Jignesh M Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A data platform based on the scaling-up approach. *Proc. VLDB Endow.* 11, 6 (2018), 663–676.

[34] Maria Patrou, Md Mahbub Alam, Puya Memarzia, Suprio Ray, Virendra C Bhavsar, Kenneth B Kent, and Gerhard W Dueck. 2018. DISTIL: a distributed in-memory data processing system for location-based services. In *Proc. ACM SIGSPATIAL*. 496–499.

[35] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems. In *CIDR*, Vol. 4. 1.

[36] Ivy Peng, Marty McFadden, Eric Green, Keita Iwabuchi, Kai Wu, Dong Li, Roger Pearce, and Maya Gokhale. 2019. UMap: Enabling application-driven optimizations for page management. In *IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 71–78.

[37] Ivy B. Peng, Maya B. Gokhale, Karim Youssef, Keita Iwabuchi, and Roger Pearce. 2022. Enabling Scalable and Extensible Memory-Mapped Datastores in Userspace. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 866–877.

[38] Vijayshankar Raman and Garret Swart. 2006. How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations. In *Proc. VLDB*. 858–869.

[39] Keven Richly. 2018. A Survey on Trajectory Data Management for Hybrid Transactional and Analytical Workloads. In *IEEE Big Data*. 562–569.

[40] Keven Richly. 2019. Optimized Spatio-Temporal Data Structures for Hybrid Transactional and Analytical Workloads on Columnar In-Memory Databases.. In *Proc. PhD Workshop@VLDB*.

[41] Keven Richly, Janos Brauer, and Rainer Schlosser. 2020. Predicting Location Probabilities of Drivers to Improve Dispatch Decisions of Transportation Network Companies Based on Trajectory Data. In *Proc. ICORES*. 47–58.

[42] Keven Richly, Rainer Schlosser, and Martin Boissier. 2021. Joint Index, Sorting, and Compression Optimization for Memory-Efficient Spatio-Temporal Data Management. In *Proc. ICDE*. 1901–1906.

[43] Rainer Schlosser, Jan Kossmann, and Martin Boissier. 2019. Efficient Scalable Multi-Attribute Index Selection Using Recursive Strategies. In *Proc. ICDE*. 1238–1249.

[44] NYC Taxi and Limousine Commission (TLC). 2020. Trip Record Data. https://www1.nyc.gov/site/tlc/about/data.page , accessed on 2022/06/01.

[45] Gary Valentin, Michael Zuliani, Daniel C Zilio, Guy Lohman, and Alan Skelley. 2000. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proc. ICDE*. 101–110.

[46] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proc. ICDE*. 1009–1024.

[47] Maarten Vermeij, Wilko Quak, Martin Kersten, and Niels Nes. 2008. Monetdb, a novel spatial columnstore dbms. In *Proc. FOSS4G*. Citeseer, 193–199.

[48] Lukas Vogel, Viktor Leis, Alexander van Renen, Thomas Neumann, Satoshi Imamura, and Alfons Kemper. 2020. Mosaic: a budget-conscious storage engine for relational database systems. *Proc. VLDB Endow.* 13, 12 (2020), 2662–2675.

[49] Haozhou Wang, Kai Zheng, Hoyoung Jeung, Shane Bracher, Asadul Islam, Wasim Sadiq, Shazia Sadiq, and Xiaofang Zhou. 2015. Storing and processing massive trajectory data on SAP HANA. In *Proc. ADC*. 66–77.

[50] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: Universal Database Optimization Using Reinforcement Learning. *Proc. VLDB Endow.* 14, 13 (sep 2021), 3402–3414.

[51] Marcel Weisgut, Daniel Ritter, Martin Boissier, and Michael Perscheid. 2022. Separated Allocator Metadata in Disaggregated In-Memory Databases: Friend or Foe?. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS Workshops*. 1202–1208.

[52] Xike Xie, Benjin Mei, Jinchuan Chen, Xiaoyong Du, and Christian S Jensen. 2016. Elite: an elastic infrastructure for big spatiotemporal trajectories. *The VLDB Journal* 25, 4 (2016), 473–493.

[53] Ningyu Zhang, Guozhou Zheng, Huajun Chen, Jiaoyan Chen, and Xi Chen. 2014. Hbasespatial: A scalable spatial data storage based on hbase. In *Proc. IEEE*

TRUSTCOM. 644–651.

[54] Zhigang Zhang, Cheqing Jin, Jiali Mao, Xiaolin Yang, and Aoying Zhou. 2017. Trajspark: A scalable and efficient in-memory management system for big trajectory data. In *Proc. APWeb-WAIM*. 11–26.

[55] Yu Zheng. 2015. Trajectory data mining: an overview. *Proc. ACM TIST* 6, 3 (2015), 1–41.

[56] Daniel C Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 design advisor: integrated automatic physical database design. In *Proc. VLDB*. 1087–1097.