Joint Index, Sorting, and Compression Optimization for Memory-Efficient Spatio-Temporal Data Management

Keven Richly
Hasso Plattner Institute
University of Potsdam, Germany
keven.richly@hpi.de

Rainer Schlosser
Hasso Plattner Institute
University of Potsdam, Germany
rainer.schlosser@hpi.de

Martin Boissier
Hasso Plattner Institute
University of Potsdam, Germany
martin.boissier@hpi.de

***Abstract*—The wide distribution of location-acquisition technologies has led to large volumes of spatio-temporal data, which are the foundation for a broad spectrum of applications. Based on these applications' performance requirements, in-memory databases are used to store and process the data. As DRAM capacities are limited and expensive, modern database systems apply various configuration optimizations (e.g., compression) to reduce the memory footprint. The selection of cost and performance balancing configurations is challenging due to the vast amount of possible setups consisting of mutually dependent individual decisions. In this paper, we present a linear programming approach to determine fine-grained configuration decisions for spatio-temporal workloads. By dividing the data into partitions of fixed size, we can apply the compression, sorting, and index selections on a fine-grained level to reflect spatio-temporal access patterns. Our approach jointly optimizes these configurations to maximize performance under a given memory budget. We demonstrate on a real-world dataset that models specifically optimized for spatio-temporal data characteristics allow us to reduce the memory footprint (up to 60% by equal performance) and increase the performance (up to 80% by equal memory size) compared to established rule-based heuristics.**

***Index Terms*—Database optimization, in-memory data management, linear programming, spatio-temporal data management**

I. Introduction

Spatio-temporal data reflects the trajectories of moving objects and enables the analysis of movement patterns, which are increasingly used in various applications. A moving object's trajectory is represented by a chronologically ordered sequence of timestamped geographical coordinates accumulated by various positioning systems (e.g., GPS). To store and process spatio-temporal data is not a trivial task due to the massive volumes of continuously captured data and the need for interactive response times. The increased performance requirements of spatio-temporal data mining applications have shifted the data management to in-memory architectures [1]. Especially for main-memory optimized databases that keep the most data in relatively limited and expensive DRAM, a more efficient utilization of the available resources can significantly affect the operating costs [2]. While removing auxiliary data structures (e.g., indexes) or applying compression techniques with higher compression rates reduce the memory footprint, they equally affect the runtime performance. Modern database systems support fine-grained decisions for these configurations [3]–[7]. This approach enables applying different optimizations such as compression, indexing, and ordering configurations for various partitions of the data independently. All single configuration decisions have an impact on the overall memory consumption and runtime performance. Additionally, they mutually influence each other, which makes the determination of performance-optimized and memory-efficient configurations difficult [8], [9]. There are several general approaches in existing work that optimize specific aspects like the compression schema selection [2] or the selection of optimized index structures [10], [11]. As the different configuration decisions mutually influence each other, we seek to jointly optimize the compression, index, and ordering configuration to determine the best runtime performance for a given workload, data characteristics, and memory budget. Note, each of those individual tuning problems is, in general, already challenging. We are still able to 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 columns and few query types. Further, to obtain a manageable problem complexity, we focus on single-attribute indexes.

In this paper, we make the following contributions. First, we introduce fine-grained table configurations to optimize data management by reflecting spatio-temporal access patterns in the storage layer. Second, we develop two linear programming (LP) models to determine workload-aware fine-grained table configurations for spatio-temporal applications, which jointly optimizes data compression, ordering, and indexing. Third, we evaluate our approaches on a real-world dataset to demonstrate the applicability and effectiveness compared to established rule-based heuristics.

II. Fine-Grained Configuration Decisions for Spatio-Temporal Data

This section introduces the architecture and the chunk concept of the research database *Hyrise* [5]. We describe the possibilities of fine-grained configuration decisions for spatio-temporal data and summarize the optimization process.

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Table		Column *a* Object ID	Column *b* Longitude	Column *c* Latitude	Column *d* Timestamp	
Chunk #n	mutable Chunk Ordering ***Unsorted***	Segment *a* **Unencoded**	Segment *b* **Unencoded**	Segment *c* **Unencoded**	Segment *d* **Unencoded**	...
⋮		⋮	⋮	⋮	⋮	
Chunk #1	immutable Chunk Ordering ***Column b***	Segment *a* **Compression B** ***Index 1***	Segment *b* **Compression C**	Segment *c* **Compression B** ***Index 2***	Segment *d* **Compression A**	...
Chunk #0	immutable Chunk Ordering ***Column c***	Segment *a* **Compression B** ***Index 2***	Segment *b* **Compression B**	Segment *c* **Unencoded**	Segment *d* **Compression C**	...

Fig. 1. Depiction of the storage layout for an exemplary table with n chunks.

A. Chunk-Based and Segment-Based Configurations in Hyrise

Hyrise is a columnar main memory-optimized database. Each table in *Hyrise* is divided into n 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. With this approach, 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, more efficient workload distributions, and simplified tiering. For that reason, similar concepts are applied by other databases [4], [6], [7]. Furthermore, the concept enables fine-grained optimizations on immutable chunks. As shown in Figure 1, it allows the selection of compression schema on a segment basis [2]. This means that we can define different encodings (e.g., dictionary-encoding, run-length encoding) for various segments of a column. We can also define auxiliary data structures (e.g., indices) and sorting decisions independently for each chunk. Like the compression schema selection, the different segments of a column can be without an index or indexed with varying approaches (e.g., binary tree). All these fine-grained decisions have an impact on the overall performance and data footprint.

B. Reflecting Access Patterns in the Data Management Layer

The most common data format to store spatio-temporal data is the sample point format, in which the trajectory of a moving object is stored as a sequence of observed locations [12]. This format is well suited for the relational schema of database systems as each sample point is represented by a tuple consisting of a moving object identifier, the timestamped location, and additional attributes. In contrast to standalone storage systems specialized for trajectory data, database systems enable a simplified integration of further data sources (e.g., business data). Consequently, modern data management platforms are including specialized engines for specific data types (e.g., spatial and spatio-temporal data) [13], [14]. By integrating spatio-temporal data management into relational database systems, the data querying benefits from the ongoing development, the highly optimized data processing capabilities, and the advanced compression techniques of such systems [15]. Due to the high volumes of continuously accumulated trajectory data and the cost-related limited main memory resources, database optimizations that leverage the characteristics of spatio-temporal applications are valuable. One aspect of spatio-temporal data management is that the specific data access patterns are often implemented in the application layer, but are not reflected in the storage layer. The access frequency and access characteristics of spatio-temporal data points changes over time. Additionally, a limited number of query types like spatio-temporal range queries and trajectory-based queries dominate spatio-temporal workloads in a plurality of applications [12]. As several applications with different access patterns commonly work on the same spatio-temporal data, complexity increases. For transportation network companies (e.g., Uber), we could observe applications with high selectivity queries on the most current data (e.g., request dispatching). These applications partially ignore data after a specific timeframe, as the data do not reflect the current situation (e.g., traffic situation) anymore. Additionally, there are queries with a low selectivity for sophisticated analytical applications (e.g., demand prediction). Another aspect is that the spatio-temporal data characteristics can vary between different timeframes strongly (e.g., seasons, day vs. night) [1].

C. Determining Configurations for Spatio-Temporal Data

Concerning spatio-temporal data volumes, the used DRAM capacities are an important cost factor for in-memory databases [2]. The systems' operating costs can be reduced by minimizing the data footprint or more efficient utilization of the available resources. To address this problem, various vendors apply threshold-based data tiering and compression approaches. Based on a defined threshold (e.g., data volume, timeframe), data partitions are transferred to lower-cost storage mediums with higher latencies. Another method is to apply stronger compression techniques or to reduce auxiliary data structures. For both, the implications on the runtime performance are difficult to estimate for a database administrator. We introduce a workload-aware approach to determine optimized table configurations based on a combination of tracked database statistics and measured benchmark queries. During runtime, modern database systems track various parameters to optimize the performance autonomously. As input for the LP models, we use query templates determined based on the information provided by the query plan cache and chunk access statistics, which are tracked by the database or estimated via min/max statistics for each chunk [3]. Additionally, we execute isolated column scan operations as benchmark queries to get information about the runtime performance and memory consumption of different encoding types. Alternatively, estimated costs could be used [2]. After the computation of an optimized table configuration, each chunk's determined configuration can be applied asynchronously to reduce the overhead [5].

III. Workload-Aware Configurations for Spatio-Temporal Data

In Part A, we describe the problem to determine memory-efficient table configurations for spatio-temporal data. In Part B, we present an LP model to solve the specified problem. In Part C, we introduce a heuristic refraining from some tuning dependencies. Part D includes database-specific restrictions.

A. Problem Description

We consider a table with N attributes and M chunks (cf. Section II-A). The problem is to find a valid table configuration for a given memory budget B and a given workload consisting of Q query templates, such that the overall performance is maximized by minimizing the workload's total execution time.

A valid table configuration consists of (i) a sorting decision for each *chunk* and (ii) a compression and index decision for each *segment*. For each segment (m, n), a configuration has to be selected from a number of available compression E and index I options. Note, these sets also include a basic option, i.e., when data is unsorted, unencoded, or not indexed.

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 $D(q)$ returns all scan operations of a query template $q \in Q$. S describes the set of all scan operations for a given workload:

$$S = \bigcup\nolimits_{q \in Q} D(q). \tag{1}$$

For the scan operations of a query exists an 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 (executed after a previous scan operation of the same query template) operates only on a subset of the data, we introduce a scan factor w_s. This factor w_s is determined by the ordered sequence of (consecutively executed) scan operations of a query template. To determine w_s, we consider the selectivity factor of the j-th operation of a query template q denoted by $\tilde{w}_{q,j}$. By default, the selectivity factor of the *first* scan operation of a query template is defined as $\tilde{w}_{q,1} = 1$. Accounting for the combined selectivities of consecutive operations within a query template q, cf. (1), for its scan operation s with operation order $J_{s,D(q)} \in \{1, ..., |D(q)|\}$ we obtain the scan factor, $s \in D(q)$,

$$w_s = \prod\nolimits_{j=1,\dots,J_{s,D(q)}} \tilde{w}_{q,j}. \tag{2}$$

Besides the selectivity, each scan operation s has the following attributes: (i) the scanned column n_s, (ii) the frequency f_s, and (iii) the type of the scan operation (e.g., between scan, less than equal scan, equal scan). The frequency f_s is the same for all scan operations of a query template q and defines the frequency of occurrence of query template q compared to all other query templates in Q.

The costs of scan operation s on segment n_s of chunk m are denoted by $c_{m,s,e,o,i}$ and determined by the segment's encoding $e \in E$ and index decision $i \in I$ as well as the chunk's ordering decision $o \in O := \{0\} \cup N$, where O includes all columns of the table plus the unsorted option ('0'). For $m \in M, s \in S, e \in E, o \in O, i \in I$, we define:

$$c_{m,s,e,o,i} := p_{s,e,o,i} \cdot a_{m,s} \cdot \omega_s \cdot f_s \cdot u_{s,e}. \tag{3}$$

The parameter $p_{s,e,o,i}$ defines the measured performance of the scan operation s executed as isolated scan operation on column n_s 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 (3) we use the successive scan penalty $u_{s,e}$ 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 $p_{s,e,o,i}$ of the benchmark queries (cf. Section II-C), we multiply $p_{s,e,o,i}$ of all consecutive scan operations with the fixed parameter $u_{s,e}$ for each value $e \in E$. This penalty value u is database-specific and can be measured with simple benchmark queries.

Finally, based on statistics and filters maintained by database systems, entire chunks can be pruned during query execution to increase the scan performance [5]. For that reason, we introduce the parameter $a_{m,s}$, cf. (3), which describes the proportional size of segment (m, n_s) in relation to the amount of data scanned within a complete column scan on column n_s. For pruned (not accessed) chunks we let $a_{m,s} := 0$. For accessed chunks m, we define $a_{m,s}$ by their relative share of actually scanned chunks, i.e., by 1 divided by the number of not pruned chunks. The memory consumption of a segment (m, n) with configuration e, o, i is described by the parameters $b_{m,n,e,o,i}$. The total memory budget used must not exceed a given budget B.

B. Optimal SSD Model: Segments with Sorting Dependencies

The SSD model allows to solve the problem described in Part A. It allows to include intra-chunk dependencies between segments with regard to the chunk-based ordering decision. The sorting dependencies are determined by costs, cf. (3), and budgets of the measured memory footprint and scan performance. To solve the SSD model, we propose the following LP formulation. The objective is to minimize the cost (in this case, the runtime) for a given workload, cf. S,

$$min \sum\nolimits_{s \in S, m \in M, e \in E, o \in O, i \in I} x_{m,n_s,e,o,i} \cdot c_{m,s,e,o,i} \tag{4}$$

where the binary variables $x_{m,n,e,o,i}$ describe whether a certain tuning configuration, cf. $e \in E, o \in O, i \in I$, for segment $n \in N$ of chunk $m \in M$ is used ('1') or not ('0'). The overall cost is calculated as the sum of the costs c of all selected segment configurations, cf. (4). To ensure valid table configurations, we define different sets of constraints. The constraints are divided into model-specific and database-specific constraints. The model-specific constraints define general requirements for the table configurations. Database-specific constraints to incorporate technical restrictions and limitations of different database systems are discussed in Section III-D). For the SSD

model, we define three types of model-specific constraints. The first one guarantees that the accumulated memory consumption of all segments (m, n) with their selected configurations e, o, i (cf. $b_{m,n,e,o,i}$) does not exceed the budget B, i.e.,

$$\sum\nolimits_{m \in M, n \in N, e \in E, o \in O, i \in I} x_{m,n,e,o,i} \cdot b_{m,n,e,o,i} \leq B. \quad (5)$$

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.,

$$\sum\nolimits_{o \in O} y_{m,o} = 1 \quad \forall m \in M. \quad (6)$$

Further, we use binary variables $z_{m,n,e,i}$ to ensure a unique index-encoding combination for chunk m's segment n,

$$\sum\nolimits_{e \in E, i \in I} z_{m,n,e,i} = 1 \quad \forall m \in M, n \in N. \quad (7)$$

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

$$x_{m,n,e,o,i} \geq y_{m,o} + z_{m,n,e,i} - 1 \quad (8)$$

$$x_{m,n,e,o,i} \leq y_{m,o} \quad \text{and} \quad x_{m,n,e,o,i} \leq z_{m,n,e,i} \quad (9)$$

C. Heuristic Solution: Independent Segment Effects (ISE)

This heuristic approach to optimize tuning configurations for the problem described in Part A is based on 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 $O_{01} := \{0, 1\}$ 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 $c_{m,s,e,0,i}$, cf. (3). If a segment (m, n) is sorted, we use $c_{m,s,e,n,i}$. 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.

Compared to the SSD model, in the relaxed ISE model we use less variables and constraints. Specifically, we use a smaller family of binary decision variables $x_{m,n,e,o,i}$, where the ordering option only reflects the binary set $o \in O_{01} = \{0, 1\}$. The variables y and z, cf. (6) - (9), are not required. The objective of the ISE model is, cp. (4),

$$min \sum_{s \in S, m \in M, e \in E, o \in \{0,1\}, i \in I} x_{m,n_s,e,o,i} \cdot c_{m,s,e,o \cdot n_s,i} \quad (10)$$

where we use $o \cdot n_s \in O$ to include the costs defined in (3) via $c_{m,s,e,o \cdot n_s,i}$. Similar to (5), the budget constraint ensures that the accumulated (approximated) memory consumption of all segments (m, n) with their selected configurations e, o, i (cf. $b_{m,n,e,o \cdot n,i}$, $o \in O_{01}$, $o \cdot n \in O$) does not exceed the budget

$$\sum_{m \in M, n \in N, e \in E, o \in \{0,1\}, i \in I} x_{m,n,e,o,i} \cdot b_{m,n,e,o \cdot n,i} \leq B. \quad (11)$$

Note, the relaxed use of c and b in (10)-(11) only approximates the exact values. Further, we directly use x to ensure that for each chunk m at most one column is sorted, cp. (6),

$$\sum\nolimits_{n \in N, e \in E, i \in I} x_{m,n,e,1,i} \leq 1 \quad \forall m \in M \quad (12)$$

and that for each segment, a unique configuration of compression e, sorting o, and indexing decision i is chosen, i.e.,

$$\sum\nolimits_{e \in E, o \in \{0,1\}, i \in I} x_{m,n,e,o,i} = 1 \quad \forall m \in M, n \in N. \quad (13)$$

D. Database-Specific Configuration Constraints

Adding database-specific constraints to the model-specific constraints enables the models to reflect certain properties of various database systems. These constraints define combinations of indexing and encoding decisions that are incompatible for a database. For *Hyrise*, secondary indexes require dictionary encoded segments as they exploit the dictionary in order to improve space efficiency [16]. Consequently, indexes on all non-dictionary segments are forbidden. This is realized via

$$x_{m,n,e,s,i} \leq v_{e,i} \quad \forall m \in M, n \in N, e \in E, s \in S, i \in I, \quad (14)$$

where the binary parameters $v_{e,i}$, $e \in E$, $i \in I$, describe whether an index i is valid (=1) for encoding e or not (=0).

IV. Evaluation

We evaluate our models on a real-world dataset introduced in Part A. In Part B, we discuss the LP models' accuracy and compare them against rule-based heuristics in Part C.

A. Dataset and Benchmark Workload

We consider the dataset of a transportation network company that consists of ten million dispatch process-related observed locations of drivers for three consecutive days in the City of Dubai [17]. Besides the timestamp, latitude, longitude, and the driver's identifier, a status attribute is tracked for each sample point (Section II-B). This status indicates the driver's status (free or occupied). 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. We define a mixed workload Q that consists of six query templates. Q is dominated by two query templates, which represent 80% of all queries, that return the driver's positions with the status free in the last hour and all positions of free drivers in the last hour in an approximate two by two km area. These kinds of queries are often used in order dispatch processes [17]. The four other query templates select all trips of sets of drivers in (i) an 8-hour timeframe, (ii) a timeframe of two days, (iii) a specific area, and (iv) all trips of free drivers in a specific area and a timeframe of two hours.

B. Evaluation of the LP Model's Accuracy

To evaluate the different models introduced in Section III, we use the database *Hyrise*. We define the input parameters based on the supported encoding and indexing properties of the database. The set of available encodings consists of run-length, dictionary, lz4, and frame-of-reference encoding. As secondary

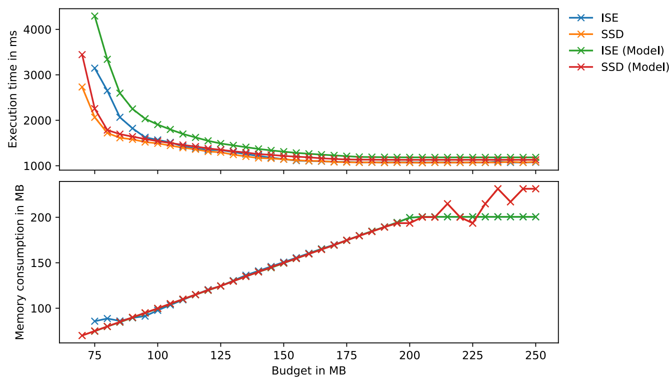

Fig. 2. Comparison of performance (top) and memory consumption (bottom) of our SSD and ISE model: synthetic vs. end-to-end results measured in *Hyrise* for the given workload and different memory budgets $B = 70, \ldots, 250$ MB.

indexes, we use an approach of Faust et al. [16] that leverages a segment's dictionary to increase the space efficiency. Consequently, we have to limit the number of valid configurations by defining (via the database-specific constraint) that indexes are only allowed on dictionary encoded segments. As we only use this indexing method in the evaluation, the index decisions I consist of the options indexed and not indexed. We partition the data into ten chunks containing one million sample points each. All measurements have been executed on a four-socket server equipped with Intel Xeon E7-4880v2 CPUs (2.50GHz, 30 logical cores). We use the numactl command to bind the thread and memory to one node to avoid NUMA effects. To solve the LP models, we used a standard *Gurobi Solver*.

As displayed in Figure 2, the predicted runtime and memory consumption of the LP models for different memory budgets B are pretty accurate compared to the end-to-end measured values for the corresponding table configurations in *Hyrise*. Overall, we can observe that there are high optimization potentials for fine-grained configuration decisions, especially for lower memory budgets. After a specific memory budget value, the performance only increases slightly or stagnates. Compared to the SSD model, the ISE model is able to determine competitive table configurations, especially for larger memory budgets. The ISE model underestimates the memory consumption, especially for lower memory budgets, and cannot determine a solution for the lowest memory budget. Here, the impact on other columns' data characteristics (e.g., number of identical values in succession) cannot be considered without detailed information about intra-chunk effects. These characteristics have an increased impact on the compression rate of encodings like run-length encoding or frame-of-reference encoding, which are used in particular for low memory budgets. The relaxation on the ordering dependencies in the ISE model enables the determination of table configurations in milliseconds compared to several seconds needed by the SSD model. Additionally, the solver runtime scales significantly better for larger problem domains (e.g., number of encodings).

C. Evaluation Against Rule-Based Tuning Heuristics

To evaluate our models against a common approach, we implemented a rule-based greedy heuristic. We calculate the

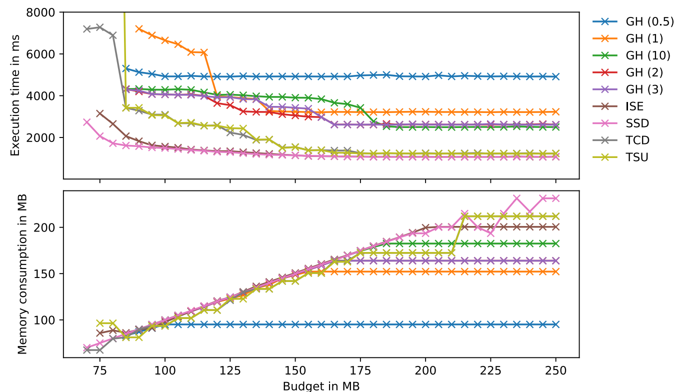

Fig. 3. Comparison of end-to-end measured performance (top) and memory consumption (bottom) for the given workload of the LP models (SSD and ISE) compared to the greedy heuristic approach with different α values (GH (α)) and column-based optimizations (TSU, TCD).

benefit of a segment $r_{m,n,e,o,i}$ based on a weighted ratio between memory consumption and runtime performance. For each segment (m, n) and each tuning option $e \in E$, $o \in O$, and $i \in I$, we define the benefit r as ($\alpha \geq 0$):

$$r_{m,n,e,o,i} = 1/\Big(b_{m,n,e,o,i} \cdot (\sum\nolimits_{s \in S} c_{m,s,e,o,i})^{\alpha}\Big). \qquad (15)$$

To calculate the costs $c_{m,s,e,o,i}$ we use (3) and consider intra-chunk dependencies. The α value is a factor to define the proportional balancing of the memory consumption and runtime performance [18]. To respect a unique sorting option per chunk, we used a two-phase implementation. In the first phase, we determine a valid base configuration by defining each chunk's sorting configuration based on the segment with the highest benefit. Afterward, we select for each segment (m, n) the tuning configuration with the lowest memory consumption that fulfills the chunks sorting constraint. In the second phase, we determine from all possible tuning options of all segments the option with the highest benefit difference to the currently selected tuning option and adapt the table configuration if the tuning option fits into the remaining memory budget. This step is repeated until no more changes are possible for the given memory budget. In our comparison, we also seek to analyze the impact of fine-grained configuration decisions on segment level against optimizations on columns. By considering the entire table with ten million entries as a single chunk, we used our LP models to compute configurations with sorting dependencies between columns (TSD) and without sorting dependencies (TSU). As displayed in Figure 3, the LP models outperform the greedy heuristics, cf. GH(α), and can leverage the available memory budget more efficiently. The greedy heuristics' measurements show that the selection of the α value has a significant impact on performance and memory consumption. Based on the two-phase approach, the greedy approaches select the sorting configuration for the entire table in the first phase. Thus, with such given sorting decisions, the computation of table configurations is not possible for low memory budgets. Further, the measurements show the expected stepwise memory and performance increase for the two column-based optimization approaches (TSU, TCD). The

approach with column dependencies (TCD) performs better than the TSU approach, especially for low memory budgets. The runtime of TSU is over 5× slower than the TCD approach for budgets less than 85 MB. Both approaches cannot reach the fine-grained models' performance as memory is used for sections of the data that are not accessed, and some queries are negatively affected by decisions made for the entire column.

V. Related Work

In this section, we briefly discuss related work from the adjacent research fields of workload-aware indexing and compression optimization with a focus on spatio-temporal data management. Zhang et al. [1] proposed a time-decay model for changing workload patterns by monitoring data distributions and adopting the indexing schema accordingly. Kimura et al. [19] presented an index selection approach that selects viable secondary indexes and considers compressed alternatives for each index based on a given memory budget. Schlosser et al. [20] introduced a selection approach that builds on a recursive mechanism and accounts for index interaction.

All these approaches do not consider different compression techniques to minimize the data footprint and create further space for auxiliary data structures. Data compression can reduce the storage requirements, allow more efficient processing (e.g., SIMD instructions), and mitigating bandwidth bottlenecks. Damme et al. [21] found that 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. Abadi et al. [22] presented a decision tree-based approach to determine compression schemas in C-store, solely based on data properties. Boissier et al. [2] introduced a workload-driven selection of compression configurations with memory constraints. A greedy heuristic determines configurations based on estimated runtime performances and resulting sizes using regression models. To the best of our knowledge, no spatio-temporal storage system applies a joint optimization approach to determine workload-driven storage configurations that consider given memory budgets. Kossmann et al. [3] introduces an LP approach to determine the order of multiple tuning features based on the pairwise dependencies between different options. Zilo et al. [9] also addressed this problem and described that mutually dependent features should be optimized simultaneously as long as the problem complexity allows a joint optimization.

VI. Conclusion

This paper demonstrates that fine-grained configuration decisions are a practical approach to reflect spatio-temporal access patterns in the database. We motivated that the identification of an optimal table configuration for a given workload and memory budget is not a trivial task as the various potential optimizations influence each other mutually. To jointly optimize a table's sorting, indexing, and compression configuration we introduce two LP models (cf. SSD and ISE) addressing cost dependencies at different levels of accuracy while allowing for trading complexity. Based on an evaluation of a real-world spatio-temporal dataset, we show that model-based fine-grained configuration decisions are superior to column-based optimization approaches. Compared to standard rule-based heuristic approaches, our LP models achieve an up to 80% increased performance for a given memory budget and gain a comparable workload runtime with up to 60% less required memory. Our results show that the SSD model reliably finds optimized tuning configurations if sorting dependencies are present. If sorting dependencies are not strong or shorter runtimes are in focus, we find that the relaxed ISE model is a suitable scalable alternative with near-optimal performance.

References

[1] Z. Zhang *et al.*, "Trajspark: A scalable and efficient in-memory management system for big trajectory data," in *APWeb-WAIM*, 2017, pp. 11–26.

[2] M. Boissier and M. Jendruk, "Workload-driven and robust selection of compression schemes for column stores," in *EDBT*, 2019, pp. 674–677.

[3] J. Kossmann and R. Schlosser, "Self-driving database systems: a conceptual approach," *Distributed and Parallel Databases*, pp. 1–23, 2020.

[4] H. Lang *et al.*, "Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation," in *Proc. SIGMOD*, 2016, pp. 311–326.

[5] M. Dreseler *et al.*, "Hyrise re-engineered: An extensible database system for research in relational in-memory data management," in *Proc. EDBT*, 2019, pp. 313–324.

[6] J. M. Patel *et al.*, "Quickstep: A data platform based on the scaling-up approach," *Proc. VLDB*, vol. 11, no. 6, pp. 663–676, 2018.

[7] A. Pavlo *et al.*, "Self-driving database management systems." in *CIDR*, 2017.

[8] D. Van Aken *et al.*, "Automatic database management system tuning through large-scale machine learning," in *Proc. SIGMOD*, 2017, pp. 1009–1024.

[9] D. C. Zilio *et al.*, "DB2 design advisor: integrated automatic physical database design," in *Proc. VLDB*, 2004, pp. 1087–1097.

[10] J. Kossmann *et al.*, "Magic mirror in my hand, which is the best in the land? An experimental evaluation of index selection algorithms," *PVLDB*, vol. 13, no. 11, pp. 2382–2395, 2020.

[11] D. Dash *et al.*, "CoPhy: a scalable, portable, and interactive index advisor for large workloads," *PVLDB*, vol. 4, no. 6, pp. 362–372, 2011.

[12] K. Richly, "A survey on trajectory data management for hybrid transactional and analytical workloads," in *IEEE Big Data*, 2018, pp. 562–569.

[13] V. Pandey *et al.*, "High-performance geospatial analytics in hyperspace," in *Proc. SIGMOD*, 2016, pp. 2145–2148.

[14] H. Wang *et al.*, "Storing and processing massive trajectory data on SAP HANA," in *Proc. ADC*, 2015, pp. 66–77.

[15] K. Richly, "Optimized spatio-temporal data structures for hybrid transactional and analytical workloads on columnar in-memory databases." in *Proc. VLDB, PhD Workshop*, 2019.

[16] M. Faust *et al.*, "Fast lookups for in-memory column stores: Group-key indices, lookup and maintenance," in *ADMS*, 2012, pp. 13–22.

[17] K. Richly, J. Brauer, and R. Schlosser, "Predicting location probabilities of drivers to improve dispatch decisions of transportation network companies based on trajectory data," in *ICORES*, 2020, pp. 47–58.

[18] G. Valentin *et al.*, "DB2 Advisor: An optimizer smart enough to recommend its own indexes," in *Proc. ICDE*, 2000, pp. 101–110.

[19] H. Kimura, V. R. Narasayya, and M. Syamala, "Compression aware physical database design," *PVLDB*, vol. 4, no. 10, pp. 657–668, 2011.

[20] R. Schlosser *et al.*, "Efficient scalable multi-attribute index selection using recursive strategies," in *Proc. ICDE*, 2019, pp. 1238–1249.

[21] P. Damme *et al.*, "From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms," *ACM Trans. Database Syst.*, vol. 44, no. 3, pp. 9:1–9:46, 2019.

[22] D. J. Abadi *et al.*, "Integrating compression and execution in column-oriented database systems," in *Proc. SIGMOD*, 2006, pp. 671–682.