Robust and Budget-Constrained Encoding Configurations for In-Memory Database Systems

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

ABSTRACT

Data encoding has been applied to database systems for decades as it mitigates bandwidth bottlenecks and reduces storage requirements. But even in the presence of these advantages, most in-memory database systems use data encoding only conservatively as the negative impact on runtime performance can be severe. Real-world systems with large parts being infrequently accessed and cost efficiency constraints in cloud environments require solutions that automatically and efficiently select encoding techniques, including heavy-weight compression. In this paper, we introduce workload-driven approaches to automaticaly determine memory budget-constrained encoding configurations using greedy heuristics and linear programming. We show for TPC-H, TPC-DS, and the Join Order Benchmark that optimized encoding configurations can reduce the main memory footprint significantly without a loss in runtime performance over state-of-the-art dictionary encoding. To yield robust selections, we extend the linear programming-based approach to incorporate query runtime constraints and mitigate unexpected performance regressions.

PVLDB Reference Format:
Martin Boissier. Robust and Budget-Constrained Encoding Configurations for In-Memory Database Systems. PVLDB, 15(4): 780 - 793, 2022.
doi:10.14778/3503585.3503588

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

1 ENCODING IN DATABASE SYSTEMS

Encoding in database management systems has been studied for several decades (cf. [30, 68, 73]). Disk-based database systems employ encoding to compress data to limit the number of I/O accesses [26]. Ideally, the usage of suitable compression and encoding schemes can lower costs by reducing storage requirements and at the same time increase performance, e.g., by mitigating bandwidth bottlenecks or enabling more efficient processing routines (cf. [74]). Consequently, an array of research studied encoding schemes [14, 27, 38, 65, 66, 68], the interplay of query execution and encoding [21, 23, 73, 75], and the selection of them [1, 4, 41, 67].

Proceedings of the VLDB Endowment, Vol. 15, No. 4 ISSN 2150-8097.
doi:10.14778/3503585.3503588

The trend towards cloud-based database deployments emphasizes the cost aspect of running database systems, even more so for main memory-optimized databases that keep most data in relatively scarce and expensive DRAM. The main memory footprint of such systems can be reduced by various means, e.g., the removal of auxiliary data structures (e.g., secondary indexes), data tiering, or data encoding. While removing auxiliary data structures and the eviction of data to slower secondary storage tiers can significantly lower the main memory footprint, they also tend to have significant impacts on the runtime performance. This paper introduces means to keep data as long as possible main memory-resident by reducing the memory footprint through data encoding[1].

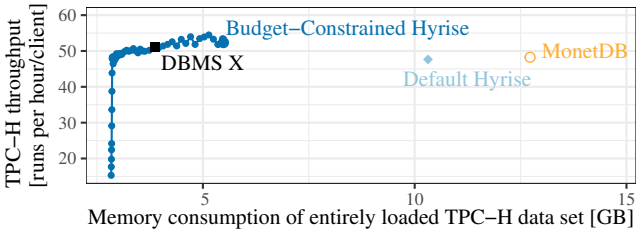


Figure 1: TPC-H performance (scale factor 10, 30 threads, 20 concurrent clients): comparing Hyrise's budget-constrained solution for varying memory budgets with DBMS X and MonetDB. The budget-constrained solution is able to reduce memory consumption *and* increase throughput.

Current approaches to automatically select suitable encoding schemes are usually conservative and do not adapt to the current workloads. To avoid potentially severe performance degradations, current approaches neglect heavy-weight compression schemes and cannot be tuned to yield configurations within a particular memory budget. We argue that both issues limit the potential of reducing the footprint in modern database systems. First, every real-world system exhibits some form of workload skew, e.g., attributes being more frequently accessed than others, which should be exploited to use heavier encoding schemes for infrequently accessed data. Second, the trend towards self-tuning database systems (or simply better tunable systems for the database administrator) demands that the database system dynamically applies encoding schemes adhering to externally given constraints.

[1] Note on heavy-weight compression vs. data tiering: A recent server with two AMD EPYC 7F72 CPUs can provide an aggregated LZ4 decompression bandwidth of over 240 GB per second. This bandwidth exceeds the bandwidth of highly tuned SSD-based systems significantly and is available in most cloud offerings. Latency-wise, for the later introduced LZ4-encoded columns, decompressing a single 4 KB block is on par with the access latencies of recent low-latency SSDs.

We introduce a budget-constrained configuration selection for the relational database system Hyrise [19]. Figure 1 shows the performance of the proposed budget-constrained approach and compares it with the default configuration of Hyrise using dictionary encoding, MonetDB, and a commercial in-memory database system for HTAP workloads (DBMS X) for TPC-H[2]. The budget-constrained and workload-driven encoding configurations are able to balance memory consumption and runtime performance while improving performance over state-of-the-art dictionary encoding.

For the ability to efficiently determine encoding configurations, we formulate three essential requirements: **(i)** While the selection of suitable encoding configurations is the eventual outcome, accurate estimations of the impact of this selection without upfront applying encodings are necessary, **(ii)** the selection process needs to yield (near-)optimal results for the entire admissible range of memory budgets, **(iii)** the selection process needs to be robust against unexpected performance regressions. We make the following contributions to address these requirements:

- To estimate and predict the performance of various encoding alternatives, we present our learned cost models. These models are efficient and accurately predict compression ratios and query runtimes (Section 3).
- For the selection of encoding configurations, we introduce a linear integer programming-based (LP) solution that yields optimal configurations and a greedy hybrid heuristic that is efficient and scalable (Section 4).
- For TPC-H, TPC-DS, and the Join Order Benchmark, we show that memory usage can be reduced significantly before impacting the runtime performance negatively (Section 5).
- We show how to yield robust encoding configurations with query runtime constraints and balanced performance gains by extending the linear programming-based solution. (Section 6).

2 AUTOMATED SELECTION OF ENCODING CONFIGURATIONS

Over the past decade, there has been an increasing interest in high-performance analytics and HTAP database systems. Most of these systems store the vast majority of their data in main memory to enable "real-time analytics" [61]. But until today, main memory remains an expensive and limited resource compared to secondary storage tiers such as solid-state drives. Most of these modern database systems share an architecture in which data is stored in a columnar layout and horizontally partitioned tables. This can be logical partitioning, a separation in write- and read-optimized partitions (e.g., SAP HANA [22]), or fine-grained horizontal partitioning concepts used in HyPer [23], DuckDB [64], and Quickstep [59]. In these systems, each column in each horizontal partition can be encoded individually.

The dominant encoding scheme for these systems is domain/dictionary encoding (cf. [19, 22, 40, 59]). Variants of dictionary encoding are a decent trade-off with acceptable compression rates and high performance. However, encoding alternatives must be considered as soon as higher compression rates are required (e.g., due to limited memory budgets). These alternatives can further reduce the DRAM requirements but come with processing overhead [14]. Depending on the frequency and type of access patterns, even heavy-weight compression such as LZ4 [13] can be viable.

Similar to related physical design optimizations such as index selection, encoding configurations need to be restrictable. For example, they are constrained by a given memory budget or domain constraints provided by third parties (cf. Section 6). To assess the quality of the encoding selection, the resulting configuration should ideally be optimal or allow estimating how far the result is off from the optimum. Finally, the system should provide the ability to accurately predict the resulting runtime performance without the requirement to physically modify any data upfront.

Hence, the main objective of encoding selection is to determine an encoding configuration within a given memory budget whose runtime and footprint are accurately predicted, and which maximizes the expected runtime performance.

2.1 Hyrise

We implemented the system for automated encoding selection in the open-source[3] relational database system Hyrise [19]. Hyrise uses a columnar table layout and stores its data primarily in main memory. Each table is divided into fixed-size horizontal partitions, called chunks. The parts of each column within chunks that constitute a logical table column are called segments.

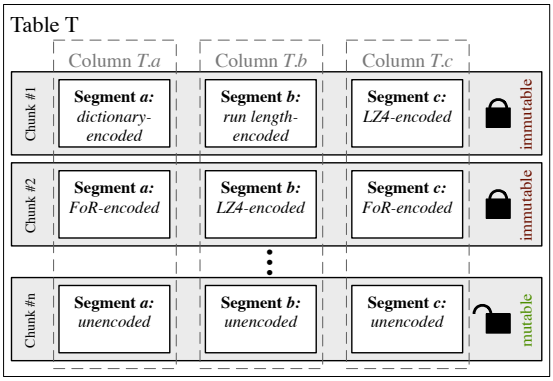


Figure 2: Storage layout for a table T with three columns and n chunks. Segments can be encoded independently.

When the most recent chunk reaches its size limit, it is marked as immutable, and a new chunk is appended for inserts. Hyrise only encodes immutable chunks. The segments can be encoded independently (see depiction in Figure 2). Modifications always append to the table, and MVCC is employed for concurrency control [69]. This architecture ensures that segment encoding does not interfere with running transactions and can be executed concurrently as an asynchronous process. Segment encoding does not impact data-modifying queries as these are executed on unencoded partitions and MVCC information is not compressed in Hyrise.

[2]We evaluated MonetDB 11.41.5 and the most recent release of DBMS X (Oct. 2021). Not all systems are equally optimized for parallel OLAP users, nor do all systems necessarily keep all data in DRAM. The goal of this comparison is to show that Hyrise's performance is sufficient for meaningful results, not to establish a ranking.

[3]Hyrise on GitHub: https://git.io/hyrise

Hyrise supports dictionary, fixed-size string dictionary, frame-of-reference (FoR), run-length, fast static symbol table (FSST) [8], and LZ4 encodings[4]. Further, segments can also be stored without any encoding and compression (so-called *Unencoded* segments). In a cascading manner, several encodings further compress their internal integer vectors (e.g., delta values in frame-of-reference encoding). This is done either using the smallest applicable integer type (e.g., `uint8_t` for vectors with values in $[0, 256)$) or bit-packing.

2.2 Encoding Selection Architecture

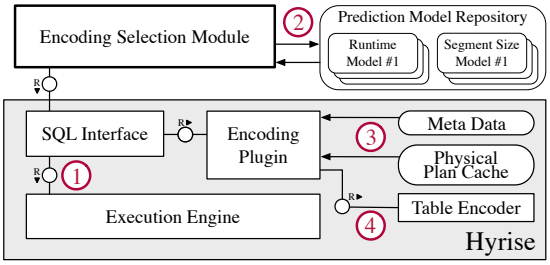


Figure 3: Overview of the encoding selection system: ① Calibration queries are executed and recorded. ② Models for runtime and size prediction are learned. ③ The physical query plan cache and meta data are analyzed. ④ The resulting encoding configuration is applied. Steps ① and ② are executed once per machine (FMC notation).

Figure 3 visualizes the architecture of our approach. The encoding selection module is an additional component of Hyrise that can be optionally used. It first executes a set of calibration queries used to train models for size and runtime prediction (①). For this paper, the query set for calibration is the set of queries of all executed benchmarks (see Section 5.2). The calibration queries are used to learn regression-based models to predict both runtimes and storage requirements (②). The module then queries the encoding plugin (cf. Hyrise's plugin concept [19]) in Hyrise to parse the database's physical query plan cache to obtain the current workload (③). With that workload information and the trained models, the encoding selection is executed, which determines a configuration that is applied by the table encoder (④). The model calibration, workload analyses, and the eventual encoding selection can be executed online or offline on another machine when necessary.

3 PERFORMANCE PREDICTION

Accurate predictions are crucial for physical design optimizations [2, 25, 36]. While it might be feasible to try out several variants of short-running query plans (cf. [33]), most physical optimizations are too expensive, both in terms of applying them as well as potential performance impacts on running systems. Moreover, there are often too many to enumerate and evaluate them efficiently.

The ability to predict the performance for a given physical design change is of high value in real-world scenarios. Database administrators (DBAs) are often reluctant to use automated design optimizations because of unpredictable performance outcomes [3].

[4]For heavy-weight compression, we have chosen LZ4 as it has a high decompression performance while compressing relatively slow compared to alternative heavy-weight schemes. We assume segments to be significantly more often read than re-encoded.

Being able to predict the performance for a range of possible configurations helps DBAs gain confidence in automated optimizations of the physical design. Moreover, accurately predicting the performance of arbitrary configurations can increase robustness by enabling means to avoid SLA-constraint violations (cf. Section 6).

3.1 Learned Cost Models

We use learned cost models for the runtime and size predictions. These models should be efficiently computable and sufficiently accurate. Efficiency is important as several optimization processes often enumerate a large array with thousands of alternatives to evaluate (e.g., the encoding selection module or the query optimizer). The source of our learned cost models is Hyrise's physical query plan cache. The cached plan stores execution data such as input/output cardinalities and runtimes for each operator of the plan. Further, each operator can store additional information, e.g., the ratio of items filtered by Bloom filters in the hash join. This approach avoids uncertainties such as estimating join cardinalities but requires that a query has been executed before we can predict the effects of physical design changes for that particular query.

We use linear regression models and regression trees to estimate the runtime performance of database operators and sizes of encoded segments [37, 60]. While linear regressions are not a good fit from a theoretical perspective as operator runtimes are usually not homoscedastic, their efficiency and robustness render them a suitable choice in our use case. Using more sophisticated models such as neural networks could probably further improve accuracy at the expense of decreased interpretability and efficiency. Moreover, while such models can take hours for learning (cf. [53]), the regression models used in Hyrise take seconds (cf. Section 3.4).

We build runtime models for each operator and size models for every encoding type. While it is also possible to use two compound models that estimate runtimes and sizes, we found simple single models to be easier to tune, debug, and maintain. To estimate the runtime of a given operation (e.g., a join) and an encoding configuration, we gather relevant meta data (e.g., data types, input and output cardinalities, distinct counts) and pass these parameters to the regression model for the operator, which returns the estimated runtime. To estimate the resulting size of an encoded segment, we again gather relevant meta data (e.g., data type, distinct count, min/max values; avg. and max string lengths for string columns) and pass it to the encoding's regression model, which returns the estimated memory consumption.

3.2 Feature Engineering

The set of features used for runtime prediction includes (i) features tracked during Hyrise's query executions and (ii) offline added features focussing on access patterns. For some operators, it is important to differentiate between steps dependent on the encoding and steps that are not. To allow tracking these operator steps, Hyrise stores *staged runtime counters*, which later allow analyzing the runtime of each operator stage (e.g., build and probe phases in hash join). This way, we can separate encoding-critical paths (e.g., the materialization of potentially encoded data in the hash join as part of the radix clustering) from parts that do not access

encoded data (e.g., the following building and probing on materialized data). The second group of features is constructed by analyzing the query graph. To estimate the runtime costs of accessing block-based encodings such as LZ4, it is important to recognize sequential and non-sequential accesses. We analyze the query graph to determine whether an operator's input position list (cf. [19]) might be scrambled (e.g., due to a previous join) or not.

The majority of features reflects the number of items read per encoding type, data type, and the type of the position list (sequential, non-sequential but monotonous, or random), resulting in sparse feature matrices. The feature array may be further extended per operator, e.g., the hash join model has features for the hash table sizes of different join types (e.g., inner and semi joins, cf. [19]).

In general, which and how many features are required depends on the target use case (e.g., OLAP or OLTP) and the workload. Accurate predictions of `LIKE` predicates might be less relevant for TPC-H and TPC-DS, but they are highly relevant for the Join Order Benchmark. If one wants to build a generalized model, such features need to be included. For our approach of a model per operator, we handle every optimization aspect that significantly impacts an operator's performance. E.g., the skipping rate of Bloom filters in the join implementation or optimization for integer-only group-by aggregations (cf. choke point analysis in [18]). Having specialized operator models means there is a need to keep features up to date in case of significant changes to the operator and new models have to be created for additional operators. This approach has a higher maintenance overhead than generalized complex models (cf. [54]) or zero-shot models (cf. [34]), which both have many desirable properties. However, we argue that the much better interpretability of less complex models as well as their efficiency is a fair trade-off.

3.3 Relative Error Minimization

Most machine learning frameworks provide an efficient implementation of linear regressions. These models commonly use a squared error metric (e.g., ordinary least squares, OLS) to minimize. This objective works well for a broad range of applications. However, the resulting decisions can be arbitrarily inaccurate when very short-running OLTP queries are significantly less accurately predicted than long-running queries (which are outnumbered, depending on the learning data). It might be more relevant for problems such as physical database design optimization to have accurate predictions over the entire range of runtimes. As it is infeasible to build segregated models for classes of runtimes, the high variance needs to be handled differently. One obvious objective would be the minimization of the absolute runtime. However, this objective still tends to optimize towards long-running queries and is not efficiently computable with off-the-shelf linear regression libraries.

For Hyrise, we aim to minimize relative errors. We use least squares percentage regressions (LSPR) to achieve that for the linear regression models [70]. In contrast to OLS regressions, LSPR minimizes *squared relative errors*. The advantage over comparable relative approaches, such as MAPE regressions [55], is that efficient off-the-shelf regressions can be used. LSPR can be implemented efficiently by simply dividing both the dependent variable as well as all explanatory variables by the dependent variable and disabling the intercept. Predictions are made using the resulting model without any further transformations.

3.4 Prediction Accuracy & Runtime Analysis

We executed the calibration queries (cf. Section 2) and trained the operator models to evaluate operator predictions. We varied the applied encoding scheme for every executed benchmark to cover all encoding schemes and data types combinations. Usually, the whole calibration data set is used for training the models and these models are then applied on a different traced workload. For better comparability, if not stated otherwise, the results reported in the following sections are gathered by holding out 20% of the calibration data set for testing and using the remainder as the training set.

We evaluated three regression models for various error metrics to predict operator runtimes. The models include an OLS-optimized linear regression, the least squares percentage regression (LSPR), and gradient-boosted regression trees (GBR, we use XGBoost [11] with 100 trees, a learning rate of 0.2, and a max depth of 7). The chosen models are widely available, efficient, and have shown to be good candidates for runtime and size regressions (cf. [10, 51]). We have also evaluated random regression trees, linear trees, and Huber regressions, but found them to be either inaccurate or inefficient to train. The error metrics are the average absolute error, the average relative error, and the root mean squared error (RMSE, which reflects the default minimization objective for most models), which are commonly used in the literature (cf. [48, 51, 54]).

Table 1 shows the results for the aggregate operator (evaluated on a TPC-H run with scale factor (SF) 10). Green denotes the lowest error per metric and experiment, red the largest. We varied the hold-out and report the mean errors of 10 runs. To evaluate the accuracy of predicting short-running executions for approaches using squared error metrics, we divided the data set into two parts: one part containing observations with a runtime less than the median runtime and one part with the remainder. As expected, the measurements of all observations show that the linear regression and GBR outperform LSPR for the RMSE metric. Looking at the shorter running observations shows that LSPR performs best. Even though GBR minimizes squared errors, it can better predict short runtimes than the OLS regression due to the use of an ensemble of decision trees. For longer runtimes, GBR performs best. Overall, both GBR and LSPR perform well.

The right side of Table 1 shows the relative error for varying hold-outs (i.e., train/test splits; 100% denotes using the entire data set for both training and testing). The linear models require less training data and show more stable results than GBR.

Concerning the model runtimes, Table 2 shows the training and prediction runtimes for the evaluated regression methods. Both linear regressions train an order of magnitude faster than GBRs.

3.5 End-to-End Workload Prediction Accuracy

To assess the end-to-end applicability, we ran experiments evaluating end-to-end predictions, analyzing single query predictions, incomplete training data, and out-of-sample predictions.

Figure 4 shows the prediction errors per query and for the entire benchmark run for TPC-H. Both LSPR as well as the boosted trees perform well for estimating the overall workload runtime. However,

Table 1: Overview of error metrics for analyzed regression models of the aggregate operator and TPC-H (SF 10). Right hand side shows the average relative error for different hold-outs (i.e., train/test splits).

	Errors (20% hold-out): Runtimes < Median			Errors (20% hold-out): Runtimes ≥ Median			Errors (20% hold-out): All Observations			Avg. Rel. Err.: Varying Hold-outs		
	RMSE	Avg. Abs. Err.	Avg. Rel. Err.	RMSE	Avg. Abs. Err.	Avg. Rel. Err.	RMSE	Avg. Abs. Err.	Avg. Rel. Err.	100%	20%	50%
Gradient-Boosted Regression Trees	3.03	1.75	9.48	597.10	80.49	0.33	422.22	41.12	4.90	5.40	4.90	4.71
Linear Regression (LSPR)	0.99	0.48	0.44	5 704.10	601.28	0.37	4 033.41	300.88	0.40	0.40	0.40	0.40
Linear Regression (OLS)	39.09	32.98	152.35	1 134.83	421.78	2.64	802.94	227.38	77.50	80.80	77.50	78.76

Table 2: Number of observations and runtimes of training and prediction (calibration data set, 20% hold-out).

	# Observations		Runtimes (ms)					
			Training Runtime (ms)			Prediction Runtime (ms)		
	Testing	Training	GBR	LSPR	Lin. Regr.	GBR	LSPR	Lin. Regr.
Aggregate	5 796	23 184	5 226.2	239.8	182.3	424.2	54.4	57.2
Join	100 194	400 782	28 253.7	720.5	644.9	2 652.6	80.2	82.0
Projection	3 225	12 903	752.8	19.6	9.5	51.4	8.3	6.6
Table Scan	39 295	157 181	54 119.8	2 223.9	1 930.0	6 068.6	1 062.2	933.7

looking at single query predictions reveals that LSPR yields larger errors for every single query prediction and achieves the overall accuracy as under- and overprediction almost equal each other out in this case. While the largest errors for LSPR are 0.49 and 1.28, they are 0.94 and 1.23 for the gradient-boosted model.

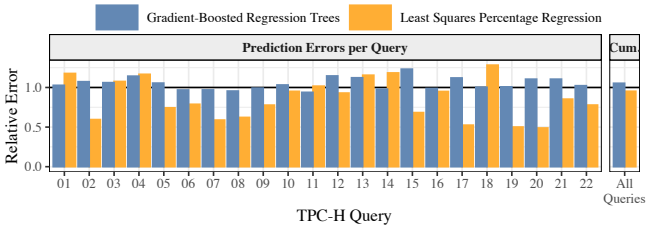


Figure 4: Relative error (predicted/actual) per query and cumulative query runtimes for TPC-H (scale factor 10).

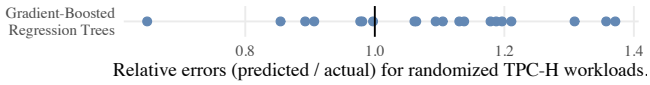


Figure 5: Relative prediction errors of GBR-predicted cumulative workload runtimes (TPC-H SF 10) with random training sets/workloads (11 of 22 TPC-H queries randomly selected).

We ran an experiment with random workloads to evaluate the prediction accuracy in case of incomplete workload knowledge. We created 20 workloads (each with 11 of 22 TPC-H queries randomly selected) for model training and evaluated the prediction accuracy of a full TPC-H run (SF 10, GBR for prediction). The error ranges from 0.65 to 1.37. Looking at the queries of each random workload shows that it is unnecessary to include every query in the training phase. But particularly long-running queries with rather unique patterns (e.g., TPC-H Q1's wide aggregation) can decrease the accuracy when they are not present in the training data.

Besides the ability to estimate unseen queries, another important aspect is the ability to estimate workloads significantly larger or smaller than used for training. Table 3 shows the prediction error for

Table 3: Relative error (predicted / actual) of estimated TPC-H runtimes for different scale factors and regression models.

TPC-H Scale Factor	1	10*	30	100
Least Squares Percentage Regr.	0.803	0.975	0.852	1.074
Gradient-Boosted Regression Trees	2.477	1.053	0.494	0.175
Linear Regression (OLS)	1.805	1.173	1.323	1.124

* Models trained on scale factor 10.

the evaluated models and four different scale factors of TPC-H. As shown in Figure 4, both GBR and LSPR predict SF 10 well. However, when predicting workloads of different scale factors, linear models outperform the GBR. GBRs regress by weighting and aggregating matching terminal nodes of ensemble trees, which store scalar values. Such approaches are thus not able to extrapolate to unseen (out-of-sample) observations.

Concluding, we found there is no single best model. If the goal is to have a well generalizing model that can be used for systems of different sizes (e.g., a multi-tenant database system where the model is trained once for the hardware instance and used for all tenants, no matter of their size), least squares percentage regressions are superior to both other models. If it is feasible to create a prediction model for a given database instance and if accurate predictions of single queries are of importance (cf. Figure 4, e.g., for query runtime constraints presented in Section 6), the GBR model is superior. For the remainder of this paper, we will use the gradient-boosted regression trees to predict runtimes and segment sizes.

4 ENCODING SELECTION APPROACHES

There is a multitude of approaches to optimize the physical database design. The literature includes reinforcement learning [35, 43], linear programming [16], greedy heuristics [41, 71], and dynamic programming [57]. To our best knowledge, there is no adequate approach for encoding selection in the literature, which automatically selects encoding schemes for a given workload and memory budget constraints. This section introduces a linear integer programming-based model and an efficient greedy heuristic.

4.1 Linear Programming Model Description

The objective is to determine optimal encoding configurations. We denote the set of tables as T. A_t denotes the set of attributes of a given table $t, t \in T$ and P_t denotes the set of horizontal partitions[5] of a given table $t, t \in T$. The set of all supported encoding schemes is denoted as E with $E = \{\texttt{LZ4}, \texttt{FSST}, \ldots\}$. We precompute two matrices $\mathbf{R}$ and $\mathbf{S}$ with the order of $|T| \times M \times N \times |E|$ with $M = max_{t \in T}(|A_t|)$ (i.e., the maximal column count of all tables), $N = max_{t \in T}(|P_t|)$

[5]For simplicity, we consider every chunk to be a horizontal partition in this paper. To handle larger problems, a partition can also cover multiple chunks.

(i.e., the maximal horizontal partition count of all tables), and $|E|$ being the number of selectable encoding schemes. **R** and **S** store aggregated runtimes and encoding sizes for every segment and encoding. These values can be exhaustively measured, estimated, or provided by the DBA. In our case, the values are estimated using regression methods presented in the previous section. We iterate over all physical query plans in the database's physical query plan cache, estimate the runtime for each encoding scheme that each accessed segment incurs, and store the cumulative estimated runtimes for the given workload. An encoding configuration is stored in the matrix **C** with the same order as **R**, with each value $c_{tape} \in \{0, 1\}$ denoting the binary decision variable which signals whether encoding scheme e is used for the segment in table t, attribute a, and horizontal partition p. As not all encoding schemes support every data type (e.g., frame-of-reference encoding is limited to numeric data types), the matrix **D** with $d_{tae} \in \{0, 1\}$ stores binary decision variables denoting whether a segment encoding e supports the data type of attribute a in table t (signaled by 1 for supported, and 0 for unsupported).

We define optimality as the minimal aggregated runtime of all queries of a given workload in the form of R and a given memory budget B. We formulate the linear problem as follows:

$$\text{minimize} \sum_{t \in T, a \in A_t, p \in P_t, e \in E} c_{tape} \cdot r_{tape} \tag{1}$$

$$\text{subject to} \sum_{t \in T, a \in A_t, p \in P_t, e \in E} c_{tape} \cdot s_{tape} <= B, \tag{2}$$

$$c_{tape} \cdot (1 - d_{tae}) = 0 \qquad \forall t \in T, a \in A_t, p \in P_t, e \in E, \tag{3}$$

$$\sum_{e \in E} c_{tape} = 1 \qquad \forall t \in T, a \in A_t, p \in P_t \tag{4}$$

The objective is to find a configuration with the minimal runtime (Eq. (1)), constrained by the cumulated size being within the given memory budget B (Eq. (2)), the selected encoding schemes supporting the data type (Eq. (3)), and the solution yielding only exactly one encoding scheme per segment (Eq. (4)).

4.2 Scalability of Solving Runtimes

Table 4: Solving times for variously sized synthetical problems. Evaluated solvers include the open-source solver Cbc, SCIP, and the commercial solver Gurobi (single- and multi-threaded). All solver runtimes limited to 10 minutes with an optimality gap of 0.01.

		Solving Runtime (s)				
		Single-Threaded			Multi-Threaded (#threads)	
Columns	**Chunks**	**Cbc**	**SCIP**	**Gurobi**	**Gurobi (2)**	**Gurobi (4)**
128	512	64.6	27.4	18.4	18.3	18.3
128	2048	DNF	126.5	82.1	82.6	81.7
128	8192	DNF	598.2	358.0	357.8	355.4
8	512	1.5	1.5	1.1	1.1	1.1
32	512	7.5	6.1	4.2	4.2	4.2
128	512	64.6	27.4	18.4	18.3	18.3
512	512	DNF	124.7	81.8	82.2	81.8
2048	512	DNF	590.6	350.7	354.2	347.9

A major issue with linear programming – particularly with binary integer problems – is the scalability for large problems. We created a series of variably sized synthetical data sets and workloads and evaluated the solving times of three solvers: the state-of-the-art commercial solver Gurobi [28], SCIP [24], and the open-source solver Cbc [49]. For Gurobi, we further measured the impact of multi-threading. Table 4 summarizes the solving times. Even though runtimes of minutes are often acceptable, given that the actual application of a new encoding configuration takes significantly longer, the point where solving takes too long can easily be reached.

The formulation of the LP problem is independent of the number of queries and the queries' complexity. However, the solving time increases more than linearly with the number of segments to handle. Concerning the selection of solvers, the evaluated open-source solvers are only capable of handling small problem instances. Gurobi is capable of handling larger instances but does not profit from multi-threading.

The solving runtimes lead to the question of what problem sizes need to be handled in real-world systems. Therefore, we analyzed a large real-world production ERP system to quantify the impact of the table to consider. Table 5 shows the number of tables for the ERP system and the TPC-DS benchmark to account for a certain cumulative percentage of the entire system size. Even though TPC-DS is considered more realistic than previous TPC benchmarks, it still does not incorporate all characteristics of existing large real-world systems. Considering the runtimes listed in Table 4, optimizing the entire ERP system using the LP approach is not feasible. However, there are effective simplifications that significantly reduce the problem size. First, as temporal skew on large tables usually does not differ between neighboring chunks, not every single chunk has to be considered. Instead, groups of chunks can be considered, which then share the same encoding configuration. Second, Table 5 shows that only considering 64 tables of the studied ERP system is sufficient to optimize 50% of the system's data. One possibility is to use the LP approach for optimizing the largest tables and use efficient heuristics (see next section) to handle the long tail. Third, in case the LP solution is desirable for a very large number of tables (e.g., due to robustness constraints, cf. Section 6), decomposition heuristics that split the problem into smaller sets which can then be solved optimally using linear programming, have shown to yield near-optimal results for related problems (cf. [29]).

4.3 Greedy Heuristic

While optimal solutions are desirable, linear programming-based solutions might be infeasible in production due to their long runtime for large problems or the availability of commercial solvers. We have implemented a greedy heuristic that is efficient and scalable. We have adapted this heuristic from a related field in physical design optimization: index selection. The heuristic is an adaption of the index selection used in IBM DB2. The discussed heuristic weights candidates by their benefit-to-cost ratio [71].

In contrast to the binary index selection problem (add index or not), the encoding selection selects exactly one encoding scheme from a set of schemes per segment. As the initial step, we determine the smallest possible configuration as the start configuration, enabling early exits for infeasible small budgets. Then, for every applicable encoding and every segment of the system,

we evaluate possible alternatives. For a currently active encoding $\bar{e}$, we determine the smallest ratio of saved runtime per byte: $argmin_{e \in E}(r_{tape} - r_{tape\bar{e}})^{\alpha}/s_{tape}, \forall t \in T, a \in A_t, p \in P_t$. We keep the alternative encodings with the smallest ratio for each segment. The weighing factor $\alpha \in \mathbb{R}^+$ enables balancing space consumption and performance. The larger α, the higher runtimes are weighted by the heuristic. In the actual greedy selection, we iteratively substitute segment encodings with the lowest weighted alternative encoding that still fits into the remaining memory budget.

Table 5: Minimal number of tables to cover different shares of the overall data footrpint for an SAP ERP system of a Forbes 500 company and the TPC-DS benchmark.

	SAP ERP Data Set	TPC-DS Data Set
Size	1,930 GB	1,234 MB
# Tables	135 807	25
50%	64	2
60%	96	3
70%	156	3
80%	252	4
90%	506	5

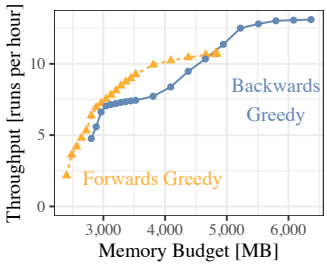


Figure 6: Forwards/backwards greedy heuristic predictions for JOB. The simple heuristics do not cover the whole admissible range of memory budgets.

The quality of the heuristic depends on α. With an emphasis on size, the heuristic tends to yield reasonable solutions for small budgets but is not able to exploit larger budgets. This is shown as the "Forwards Greedy" solution in Figure 6. As the initial greedy selection emphasizes size over runtime, no well-performing alternatives are available when the budget allows them.

As a consequence, we added the "Backwards Greedy" heuristic. This heuristic starts with the configuration that has the lowest possible runtime. For a currently active encoding $\bar{e}$, we determine the encoding with the smallest ratio of saved bytes per runtime: $argmin_{e \in E}(s_{tape} - s_{tape\bar{e}})/r_{tape}{}^{\alpha}, \forall t \in T, a \in A_t, p \in P_t$. The results are shown in Figure 6. The backwards heuristic can determine fast configurations for large budgets but falls short for small budgets.

Therefore, we combine both heuristics as a hybrid heuristic. For every budget, we take the heuristic solution with the lower *predicted* runtime. For the remainder of this paper, we only show results for the hybrid greedy heuristic. The hybrid greedy heuristic, which needs to determine two encoding configurations, is ~100× faster than the LP solution for TPC-H with a scale factor of 10.

5 EVALUATION

We evaluate the results for the heuristic, the linear programming solution, as well as static configurations using TPC-H, TPC-DS[6], and the Join Order Benchmark. Besides the well-known TPC-H benchmark, we have chosen TPC-DS as it includes both data and workload skew [62]. The Join Order Benchmark (JOB) is based on a real-world data set from the internet movie database (imdb.com). For TPC-H and TPC-DS, we used a scale factor of 10. Unless noted otherwise, we report the throughput (i.e., the number of sequential runs of all shuffled benchmark queries per hour).

[6]As of December 2021, Hyrise supports 47 TPC-DS queries.

All runtime predictions have been made with the initially created models for the calibration phase of running all three benchmarks. Please note, to create a fair calibration data set and thus more generalized models, we generated the runtime and size models using the complete calibration data and did not create benchmark-specific ones. All evaluations have been done using another set of workloads than the ones used for learning. How prediction models can be efficiently trained with a small set of calibration queries is ongoing research (cf. [34, 51]).

In addition to the LP solution and the heuristics, we added two static configurations for reference that do not adhere to a budget. The first static configuration statically chooses for every segment the encoding scheme with the lowest runtime-size product (i.e., $argmin_{e \in E}(r_{tape} \cdot s_{tape}), \forall t \in T, a \in A_t, p \in P_t$; please note that this strategy also requires accurate runtime and size predictions). The second one is Hyrise's default configuration using dictionary encoding. We evaluated various α for the greedy heuristic.

5.1 Runtime and Size Prediction

Figure 7 shows the *predicted* results for TPC-H. The LP-based approach performs slightly better than the greedy heuristics. For TPC-H, the default configuration using dictionary encoding uses significantly more space than all alternatives while also being slower.

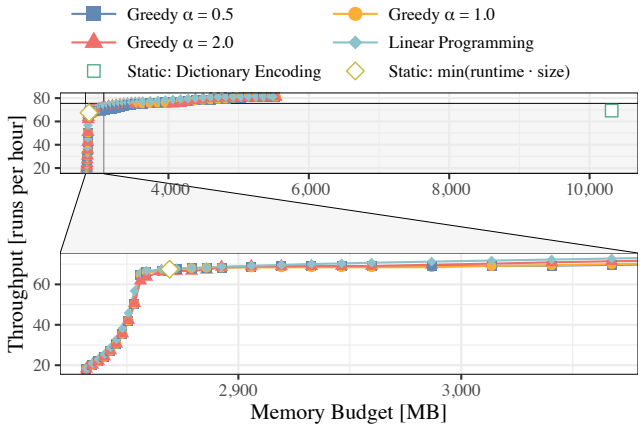


Figure 7: *Predicted* results for TPC-H (scale factor 10). Comparing hybrid greedy heuristics with varying α values, two static configurations, and the linear programming-based approach.

5.2 End-to-End Evaluation

In this section, we study how well the predictions translate to actual results when evaluating TPC-H configurations as well as configurations for TPC-DS and the Join Order Benchmark. All measurements have been executed on a server equipped with an Intel Xeon E7-8890 v2 CPU (2.80 GHz, 15 cores, 30 threads). Hyrise has been compiled with Clang 9.0.0 (-o3). No update streams have been executed. Unless noted otherwise, we evaluated single-threaded executions to exclude potential multi-threading and scheduling overheads which might hide the effects of changing encodings.

Figure 8 shows the results. The predicted runtimes for TPC-H estimate accurately the actual measured runtimes. The performance

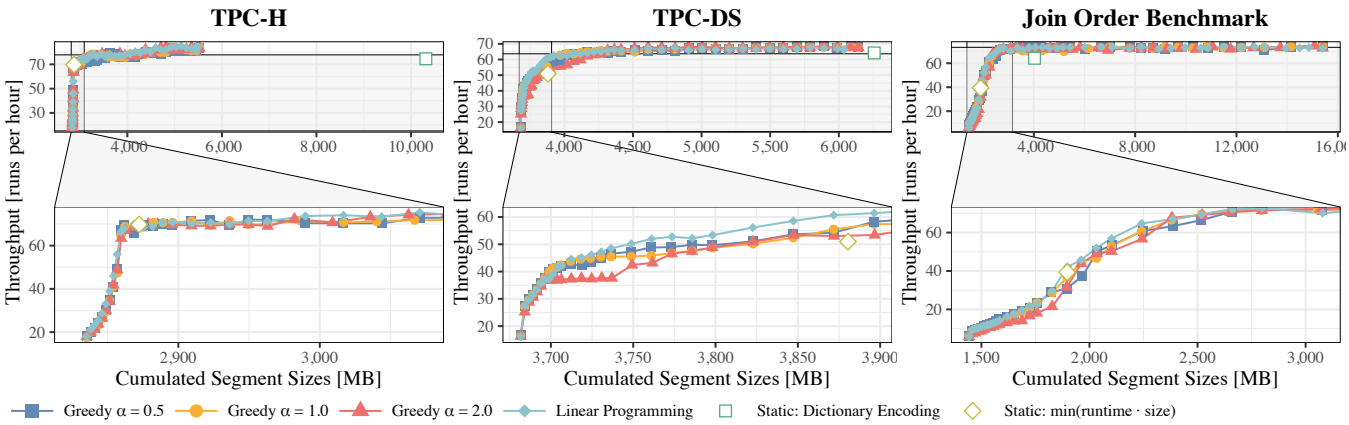


Figure 8: Results for TPC-H (SF 10), TPC-DS (SF 10), and Join Order Benchmark. Comparing greedy heuristics with varying α values, two static configurations, and linear programming-based approach. The upper row shows the entire spectrum of configurations, lower row zooms into from the smallest possible configurations up to the budget where the LP solution achieves 90% of the maximum throughput.

order of solutions is well reflected, with the optimal LP solution slightly ahead of the heuristics in most situations. The observation that memory consumption can be reduced significantly over a static default such as dictionary encoding holds for all three benchmarks. The large string columns of TPC-H, which are infrequently or never accessed (e.g., `l_comment`), allow significant reductions when LZ4 or FSST are used, leading to a significant reduction compared to static dictionary encoding. The TPC-H, all three greedy heuristics perform very well. The LP approach and the greedy heuristics outperform static dictionary encoding with better runtime performances and smaller footprints for all benchmarks. The static configuration that minimizes the runtime and size product yields well performing configurations.

5.2.1 Transactional Workloads. To evaluate the performance of analytical workloads with concurrent modifying transactions (so-called HTAP workloads), we executed the CH-benCHmark [12]. The CH-benCHmark is an extension of the TPC-C benchmark, which runs reformulated TPC-H queries on the TPC-C dataset in parallel. We hypothesize that transactional processing is significantly less impacted by encoding. The reason is that the main bottlenecks are less on the data access path but rather the transaction and conflict handling. Hyrise uses MVCC to handle concurrent workloads and versioning, similar to the approach discussed in [69]. The additional columns to store commit information are never compressed in Hyrise as they are always mutable.

We evaluated 15 minutes runs. We used ten TPC-C warehouses, five parallel TPC-C clients, and a single analytical query stream (scale factor 1.0). Smaller configurations do not impact the transactional throughput for a wide range of budgets. Only for the two smallest configurations, the throughput degrades. The throughput is 48% of the optimum for the smallest possible configuration. The analytical performance is more impacted by smaller budgets and shows a comparable curve as seen with TPC-H and TPC-DS. Interestingly, the worst throughput is only a 36% drop from the optimum

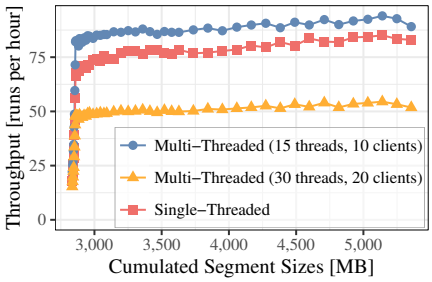


Figure 9: Comparison of TPC-H (scale factor 10) throughput for varying memory budgets and single-threaded and multi-threaded execution (LP configurations).

(compared to 23% for TPC-H), which we attribute to the scheduling overhead of Hyrise caused by the parallel TPC-C transactions, hindering better analytical performance.

5.2.2 Parallel Processing. To measure how well the expected single-threaded performance maps to parallel execution, we evaluated TPC-H with two parallel settings: running ten parallel clients on a Hyrise instance with 15 cores and a 20 clients/30 cores variant. The results are shown in Figure 9. Due to parallel processing overheads and increased pressure on the shared caches, the performance drops compared to single-threaded execution even though each user theoretically has 1.5 cores. When all 15 physical cores are used, the throughput drops by less than 10%, while it drops by ~40% when all 30 logical cores are used. However, the general performance pattern remains stable, showing that the encoding selection is applicable to parallel systems and workloads.

5.2.3 Impact on Running System. Hyrise applies encoding configurations asynchronously in the background. Figure 10 shows an evaluation for TPC-H with 20 clients and the Hyrise server using 30 cores. First, we evaluated the impact of a single encoding job (for configuration changes #1, #2, and #3). For the first configuration, which applies a configuration with a 2.8 GB budget, the single sequential encoding job takes over 10 minutes. The reason for this runtime is the encoding of string columns (e.g., lineitem's `l_comment`) with LZ4. Applying the same configuration again on

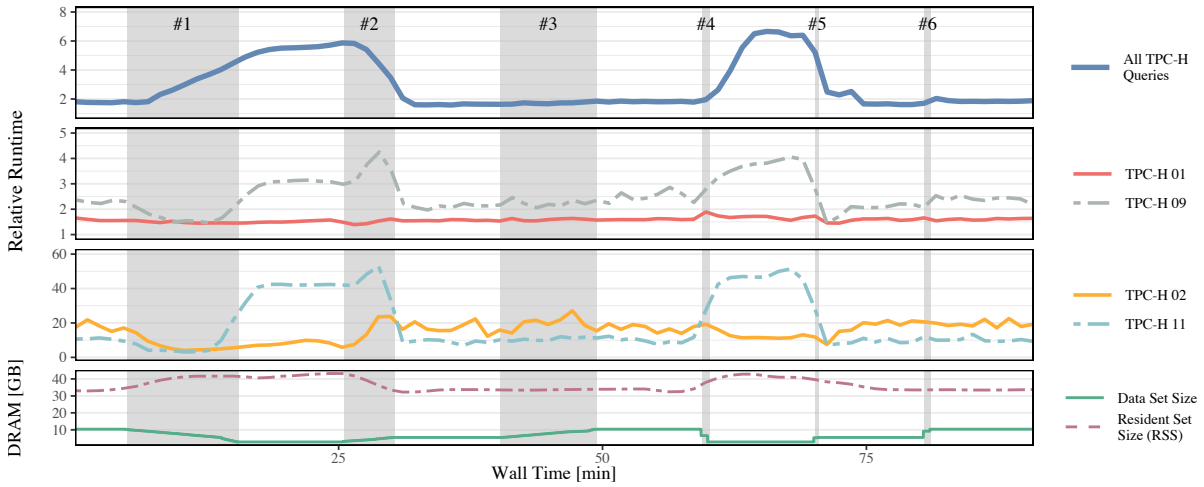


Figure 10: Impact of asynchronous encoding on TPC-H performance (relative runtime (runtime/min(runtime)), smoothed) and DRAM consumption (20 clients). Hyrise uses 30 cores. Plot on top shows the runtime performance of complete TPC-H runs. The next two plots show the queries that are the least (TPC-H 01 & 09) and the most (TPC-H 02 & 11) affected by different encoding configurations. The setup starts with dictionary encoding. Using a single encoding job, the smallest (#1) and fastest (#2) LP solutions are applied (encoding duration shaded in gray). #3 applies dictionary encoding. The same configurations are applied again (#4, #5, and #6) with 20 concurrent encoding jobs.

a dictionary-encoded data set with 20 concurrent encoding jobs improves the encoding duration from 10 minutes to 46 seconds.

The impact on query performance of concurrently running encoding jobs is acceptable as the background jobs are executed with a low priority. While a single sequential encoding job did not show a measurable impact on performance, the runtime of applying a configuration can be very long. The impact of the configurations on single queries can be significant, as most visible by the impact of small budgets on query 11. Looking at the resident set size (RSS) on the bottom of Figure 10 shows that the footprint reductions of very small memory budgets can increase the RSS under heavy load. One of the main reasons for the RSS increase is Hyrise's currently inefficient handling of non-sequential accesses to LZ4 segments, where each single row access might decompress another LZ4 block, which is cached for the lifetime of the segment iterator.

6 ROBUSTNESS

When autonomous decisions are made in a dynamic system, measures must be taken to lower the chances of unexpected performance regressions. With regards to the automatic determination of encoding configurations, yielded configurations should be robust. As stated by Haritsa in [31], there is no single definition for robustness. We define robustness as a property of the encoding selection, which minimizes negative runtime performance effects. We see two negative impacts that we want to address. First, we want to avoid unexpected performance degradation for queries with strong performance requirements, which can happen even though the main objective (i.e., cumulative workload runtime) decreases. Second, we want to avoid unexpected performance cliffs when the memory budget changes. Ideally, the relative performance improvements within a set of queries should not be overly skewed.

6.1 Constraints on Query Runtime Changes

Applied configurations of the LP-based approach – even though results are optimal with respect to the cumulative workload runtime objective – can yield surprising query performance results where certain queries degrade even though the budget is increased. We talked to several database engineers of commercial database systems to get their perspectives on automated database encoding. Two of the most often mentioned aspects have been that vendors are (i) keen on having the means to compress data further but are also (ii) reluctant to apply stronger compression since their main concern are runtime regressions of single queries rather than the performance degradation of the entire workload. Unexpected single query regressions are amongst the major reasons for customer complaints about autonomous/self-tuning components.

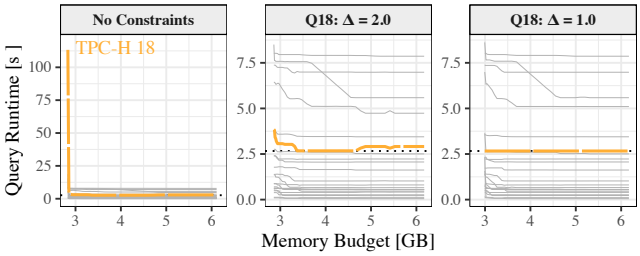


Figure 11: Query runtime constraints for TPC-H query 18 (emphasized in orange, all other queries shown in gray). The very left shows an unconstrained solution. The other two plots show constraints for query 18 with Δ values of 2.0 and 1.0. The dotted line marks the performance of query 18 for the default dictionary encoding.

While ideally, an autonomous system would identify such queries by itself (even though they are often not recognizable by frequency

or runtime), we approach this problem by accepting a set of runtime-constrained queries. A domain expert manually declares queries as runtime-constrained together with a permitted relative runtime factor Δ. One of the advantages of the linear programming solution is that we can directly include such constraints into the model while ensuring optimal results. However, the resulting minimally possible memory consumption might increase due to the added constraints.

We denote CQ as the set of constrained queries and extend $\mathbf{R}$ to $\bar{\mathbf{R}}$ which stores the predicted runtimes further split by queries (i.e., $\bar{r}_{tapeq} \in \mathbb{R}^{|T|\times M\times N\times|E|\times|CQ|}$). For each query $q \in CQ$, Δ_q denotes the factor for the maximum permitted runtime of q ($\Delta_q = 2.0$ enforces that query q's runtime does not degrade more than by a factor of two, no matter what the budget is). Coming back to the problem description in Section 4.1, we add the following constraint:

$$\text{subject to} \sum_{t\in T,a\in A_t,p\in P_t,e\in E} \bar{r}_{tapeq}\cdot c_{tape} \leq \Delta_q \cdot \sum_{t\in T,a\in A_t,p\in P_t} \bar{r}_{tap\tilde{e}q} \quad ,\forall q \in CQ$$

with $\tilde{e}$ denoting the default encoding for data tables (i.e, dictionary encoding in Hyrise).

The left plot of Figure 11 shows the relative runtime of every TPC-H query for different memory budgets. Query 18 – shown in orange – is the slowest query when the smallest possible TPC-H configuration is applied, with a 40× longer runtime than the default configuration of Hyrise with dictionary encoding. Let us now assume that this particular query is of high importance for the database user, and thus query 18's performance shall be constrained. We evaluated two different Δ values of 2.0 and 1.0 (middle and right of Figure 11). Using these Δ settings, the DBA can ensure that the performance of query 18 is always within the maximum allowed runtime relative to the runtime when using dictionary encoding (marked with the dotted line). This constraint is especially important for small budgets. Interestingly, for $\Delta = 2.0$, configurations with budgets between 3.5 and 4.5 GB have the same runtime for query 18 query as with dictionary encoding, while the performance degrades for larger budgets. This pattern can be seen with other queries as well, but less pronounced. For budgets around 4 GB, the optimal configuration yields good performance for query 18 but larger budgets allow further reducing the cumulative workload runtime at the expense of query 18's performance.

6.2 Equally Distributed Performance Gains

One main objective of our robustness measures is to minimize unexpected performance degradation that a user might observe when applying an automated encoding selection. Such an unexpected situation could be that a slightly increased memory budget has a significant positive impact on some queries' performance while others' performance remains unchanged. The left plot of Figure 12 shows the normalized query runtimes (relative to each query's maximum runtime over all budgets) for TPC-H. As stated before, we want to reduce unexpected performance changes that a user might observe when using an automated encoding selection. Assume a user starts with the smallest possible configuration and iteratively increases the budget to observe the runtime effects. For only slightly increased budgets, the runtime of query 18 decreases significantly, while budgets larger than 3.5 GB appear to have almost no effect on query 18 at all. Query 13 on the other hand benefits from budgets larger than 4.5 GB. Even though these patterns are explainable by the cumulative runtime performance, they can be counterintuitive for users and hard to comprehend.

To mitigate such effects, we extend the linear integer problem. We add two new continuous variables $\beta \in \mathbb{R}^+$ and $z \in \mathbb{R}$. β is provided by the user, while z is an added variable to the linear program, resulting in a mixed-integer linear program. With β, one can set a lower and upper runtime bound for all queries Q. Each query's runtime must be within these bounds which are relative to the queries' runtimes of the fastest configuration. The variable z allows the linear program to move both bounds to accommodate smaller budgets. The purpose of this adaption is to enforce that all queries have a similar relative runtime change compared to their optimal performance. For every query $q \in Q$, $\hat{r}_q$ denotes the query's runtime for the configuration with the smallest cumulative runtime. We add the following two constraints to Eqs. (1) to (4):

$$\begin{aligned}\text{subject to} \quad \frac{1}{\sqrt{\beta}}\cdot z\cdot \hat{r}_q &\leq \sum_{t\in T,a\in A_t,p\in P_t,e\in E} \bar{r}_{tapeq}\cdot c_{tape} \ ,\\ \sqrt{\beta}\cdot z\cdot \hat{r}_q &\geq \sum_{t\in T,a\in A_t,p\in P_t,e\in E} \bar{r}_{tapeq}\cdot c_{tape}\\ &\forall q \in Q. \qquad (5)\end{aligned}$$

The middle and right plots of Figure 12 show the effect for an unconstrained configuration and two configurations with β values 1.2 and 1.05. Comparing the results shows that the runtime changes of queries over the array of memory budgets are less skewed.

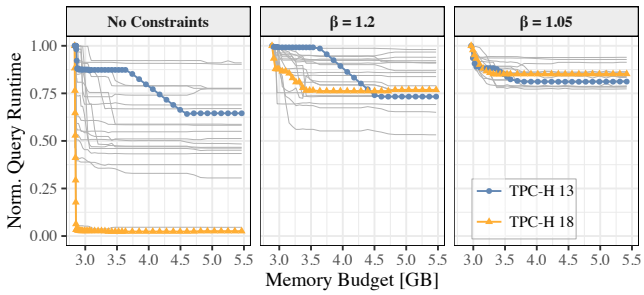
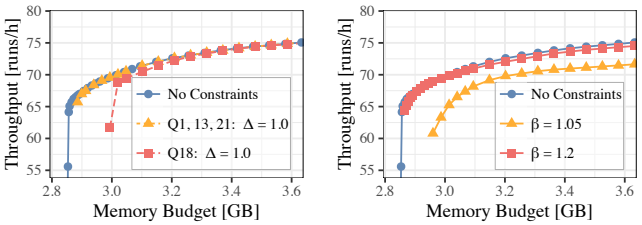


Figure 12: Relative performance per query for varying memory budgets and TPC-H (scale factor 10) showing an unconstrained solution and two constrained solutions with different β values.

6.3 Performance and Solving Runtimes

As both added constraints limit the possible solution space, we evaluated the overall effect on TPC-H. In addition to the constraint for TPC-H query 18, we added another one that constrains the three slowest queries of TPC-H in Hyrise with dictionary encoding (i.e., query 1, 13, and 21) with $\Delta = 1.0$ (i.e., no change over the default encoding). As shown in Figure 13a, for the constraint on the three most expensive queries, the smallest possible configuration increases by 53 MB compared to the unconstrained solution, while the smallest possible configuration has a 3.6× higher throughput. For the constraints on query 18, the effect is much larger due to query 18's significant performance degradation for small budgets. The smallest possible budget increases by 162 MB.



(a) Configurations constrained on the permitted query slowdown.

(b) Configurations constrained on equal performance gains.

Figure 13: Predicted performance and memory budgets for TPC-H (scale factor 10) for the two introduced robustness measures.

The effect for constraints of equal performance gains is comparable. For $\beta = 1.2$, runtime performance is slightly worse than the unconstrained solution, but the smallest possible budget is larger. The solution for $\beta = 1.05$ has a larger impact on both runtime and memory budgets as the queries are much more constrained.

Another aspect is the solving runtime shown in Table 6 for TPC-H, TPC-DS, and JOB as well as different constraint times. All benchmarks are solved in less than ten seconds using Gurobi without robustness constraints. For Gurobi, using multiple threads does not improve the runtime and can even be slower. The query-based constraints add a small overhead. The models constraining performance gains between queries run significantly slower.

Table 6: Solving times for evaluated benchmarks (Cbc considered several problems infeasible; scale factor 10 for TPC-H and TPC-DS).

			Solving Runtime (s)			
			Single-Threaded		Multi-Threaded (#threads)	
Benchmark	**Columns**	**Chunks**	**SCIP**	**Gurobi**	**Gurobi (2)**	**Gurobi (4)**
JOB	108	2027	12.7	5.1	5.1	5.2
TPC-DS	425	1231	10.8	5.5	5.5	5.5
TPC-H	61	1326	19.9	7.2	7.2	7.2
TPC-H (β=1.2)	61	1326	663.7	59.4	41.5	154.4
TPC-H (β=1.6)	61	1326	616.6	35.2	34.6	33.7
TPC-H (β=2.0)	61	1326	617.0	25.7	26.5	26.3
TPC-H (Δ=1.2)	61	1326	17.7	8.3	8.3	8.4
TPC-H (Δ=1.6)	61	1326	21.0	8.3	8.3	8.3
TPC-H (Δ=2.0)	61	1326	20.9	8.2	8.2	8.3

6.4 Robustness-Considerate Application

Solutions that yield robust configurations with runtime constraints rely on accurate runtime predictions. While we consider prediction accuracy sufficient to estimate the runtime of entire benchmarks, single queries can be mispredicted.

We iteratively apply a configuration while monitoring the runtime-constrained queries for such cases. First, we determine a runtime-constrained configuration with the minimal possible size. When applying this configuration, we first apply all changes to segments that are not accessed by the constrained queries. Then, we iteratively apply the remaining changes and monitor the runtime of the constrained queries. As soon as the constraints are violated, we reverse the last changes.

7 RELATED WORK

The topic of encoding in database systems has been studied for decades. On the one hand, encoding is applied to compress and lower costs by reducing the storage requirements [65]. On the other hand, research on encoding has been driven by limiting bandwidths of the primary storage device, i.e., hard-drive disks [52]. For main memory-optimized databases, the objective was no longer to reduce IO accesses as in disk-based database systems but rather to mitigate the effects of the main memory bottleneck (cf. [6]).

Abadi et al. presented an encoding system for the columnar C-Store system, which supports various encoding schemes allowing operations directly on compressed data [1]. The authors also proposed a decision tree to select encoding schemes, which uses both workload and data properties.

Lemke et al. presented a performance-optimized approach for TREX, the predecessor to SAP HANA [47]. Compression schemes are explicitly handled within most database operators at the cost of code complexity and maintenance efforts, allowing to fully exploit vectorization and other optimizations.

Damme et al. carried out a comprehensive study of light-weight integer encoding that strongly influenced this work [15]. The authors found that sophisticated compression techniques can significantly impact both performance and compression ratios, but one must consider multiple dimensions to determine which technique is best in which scenario. Moreover, the authors introduce a cost model to select appropriate techniques.

Mosaic is an approach using linear programming for data placement on multiple storage tiers [72]. The runtimes are predicted by estimating the table scan performance of each storage tier.

Two recent approaches for encoding advisors are CodecDB [39] and LEA [10]. CodecDB analyzes data characteristics to make encoding recommendations. It does not analyze the workload. LEA uses three prediction models, one for size estimation and two for estimating scan performance, to recommend encodings based on size and performance. The authors decided for regression trees and linear regressions, similar to our findings. Both approaches do not take memory budgets or robustness considerations into account.

To the best of our knowledge, no commercial database system uses an encoding selection mechanism that exploits the potential of automated workload-driven compressions.

8 DISCUSSION

The introduced framework of predicting and determining budget-constrained encoding configurations opens a wide array of possibilities for further investigations. We want to briefly discuss two issues related to data encoding and optimizing for a given workload and formulate recommendations.

8.1 Tiering to Secondary Storage

Several main memory-optimized database systems tier data to secondary storage to lower their memory footprint instead of encoding data with higher compression ratios (cf. [17, 20, 46, 56, 58]). Given the performance of recent solid-state drives, significant parts of the data can be tiered without severely impacting the runtime.

We argue that compressing data should be considered before evicting it. Most workloads include large parts of data that are

rarely accessed and can thus be easily moved to secondary storage. Nevertheless, the potential of reducing the footprint of even frequently accessed data is often not fully exploited. The gap between main memory and secondary storage is still significant. The tiering system *Mosaic* heavily compresses data using `zstd` before tiering it to secondary storage, underlining that current SSD-based systems still perform worse than main memory and continue to be the major bottleneck. Consequently, we argue that both tiering as well as encoding of main memory-resident data must be considered to achieve the best ratio of runtime performance and main memory consumption. How the mentioned aspects translate to the usage of non-volatile memory (NVM) or disaggregated memory needs to be evaluated in future research.

8.2 Changing Workloads

The shown approach optimizes for the most recently observed workload in the form of the physical query plan cache. Consequently, the yielded configurations do not necessarily perform well for changing workloads. Nevertheless, we do not consider changing workloads problematic for the following reasons: **(i)** Rerunning the entire prediction and selection pipeline is efficient and can be done within minutes, even for large systems, as no new models need to be trained. For the LP solution, models are stored and can be supplied as a starting solution to the new workload. **(ii)** Changing the encoding configuration is an asynchronous background job and is split in many small non-blocking jobs. This allows steadily adapting the system without downtimes (cf. Section 5.2.3). **(iii)** It is possible to incorporate reconfiguration costs into the LP to balance the runtime gained by a new configuration and the costs of changing the current configuration (cf. [5]). The ability to forecast (cf. [50]) and anticipate future workloads is nonetheless desirable.

8.3 Prediction & Selection Considerations

Physical Cost-Based Query Optimization: Hyrise's optimizer does not consider encodings when queries are optimized but solely uses a logical cost model with estimated cardinalities. We exploit this fact as encoding decisions do not change query plans. The more encodings will be integrated into the optimizer, e.g., when predicates are no longer sorted by selectivity but expected runtimes, the less accurate the current linear model will reflect the actually executed queries when encoding configurations change.

Static Encoding Selection: We have analyzed the best-performing configuration for TPC-H and found five general recommendations. **(i)** Integer columns that are frequently joined should not be compressed unless **(ii)** the system can profit from dictionary-encoding (e.g., `l_orderkey` in query 18; cf. choke point 1.3 in [7]). **(iii)** Less frequently accessed integer columns with high distinctness (e.g., primary keys) are best stored using a form of delta encoding. In the case of dictionary encoding, a high distinctness can lead to large dictionaries, which can significantly increase the cache miss rates (cf. [32, 63]). **(iv)** Strings are best packed in a fixed-width array when they are small (e.g., `l_shipmode`). **(v)** In case none of the former situations matches, dictionary encoding is a good default encoding, which also has the best performance for sequential operations.

Dynamic Encoding Selection: The results presented in this paper show that incorporating workload knowledge into the encoding decision provides significant advantages. Workload knowledge helps lower the memory footprint of seldomly accessed data, while faster encodings can be used for frequently accessed data. Whether simpler approaches than the one presented in this paper are sufficient depends on whether runtime predictions are desirable and if robustness should be considered. For simpler cases, light-weight access counters as used in Hyrise, SAHARA [9], or [45] can already reveal sufficient workload information and are comparatively easy to implement. In case such counters are stored per partition, they also allow incorporating and exploiting temporal skew in the workload. If memory consumption is less of a concern, static decision trees, as done by Abadi et al. [1], provide an efficient way to yield a good performance while being easy to build and maintain.

Heavy-weight compression should only be used when the workload is well known. In this case, the footprint reductions can be significant with no or only small performance penalties. LZ4 can be a good alternative even for regularly accessed columns as long as the operations are mostly sequentially. Random accesses, however, are slow due to LZ4's block-wise compression.

Automated approaches that select indexes or data tiers (e.g., [42] and [72]) are already non-trivial. As an increasing number of database components will be automated through some form of machine learning, optimizing multiple mutually dependent optimizations will be of increasing importance (cf. [44]).

9 CONCLUSION

We introduced an efficient operator-based prediction system using regression models, which can accurately predict both the runtime of a proposed encoding configuration and the resulting memory footprint without requiring the need to apply any actual encoding upfront. Using these prediction models, we presented a system that automatically determines encoding configurations for a given workload and memory budget. To find such configurations, we proposed a scalable and simple greedy heuristic and a linear programming-based approach that yields optimal configurations and increases robustness by incorporating query runtime constraints. The greedy heuristics are efficient and scalable and can yield configurations that are often on par with the optimal LP solutions. Both the LP-based solution as well as the greedy heuristic significantly reduce the memory footprint while at the same time improving performance compared to state-of-the-art dictionary encoding.

We argue that the automated selection of encoding configurations is an essential aspect of physical database design as it allows the database system to scale its own size depending on the given requirements, which can improve efficiency and lower costs.

ACKNOWLEDGMENTS

I want to thank various people for their contribution to this project. Markus Dreseler, Jan Kossmann, Moritz Eyssen, and all other Hyrise contributors for their outstanding work on Hyrise. Max Jendruk for his work on the encoding framework. Rainer Schlosser for introducing me to linear programming and scientific working. Furthermore, I want to thank Keven Richly, Daniel Ritter, Stefan Halfpap, and the anonymous reviewers for their valuable feedback.

REFERENCES

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 671–682.

[2] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *IEEE 28th International Conference on Data Engineering, ICDE*. 390–401.

[3] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proc. VLDB Endow.* 14, 7 (2021), 1241–1253.

[4] Martin Boissier and Max Jendruk. 2019. Workload-Driven and Robust Selection of Compression Schemes for Column Stores. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT*. 674–677.

[5] Martin Boissier, Rainer Schlosser, and Matthias Uflacker. 2018. Hybrid Data Layouts for Tiered HTAP Databases with Pareto-Optimal Data Placements. In *34th IEEE International Conference on Data Engineering, ICDE*. 209–220.

[6] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*. 54–65.

[7] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC, Revised Selected Papers*. 61–76.

[8] Peter A. Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *PVLDB* 13, 11 (2020), 2649–2661.

[9] Michael Brendle, Nick Weber, Mahammad Valiyev, Norman May, Robert Schulze, Alexander Böhm, Guido Moerkotte, and Michael Grossniklaus. 2022. SAHARA: Memory Footprint Reduction of Cloud Databases with Automated Table Partitioning. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT*. 13–26.

[10] Lujing Cen, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. LEA: A Learned Encoding Advisor for Column Stores. In *aiDM '21: Fourth Workshop in Exploiting AI Techniques for Data Management*. 32–35.

[11] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 785–794.

[12] Richard L. Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems, DBTest*. 8.

[13] Yann Collet. 2022. LZ4 compression algorithm. http://www.lz4.org Accessed: 2022-01-23.

[14] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT*. 72–83.

[15] 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.

[16] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *PVLDB* 4, 6 (2011), 362–372.

[17] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. 2013. Anti-Caching: A New Approach to Database Management System Architecture. *PVLDB* 6, 14 (2013), 1942–1953.

[18] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *PVLDB* 13, 8 (2020), 1206–1220.

[19] 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 *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT*. 313–324.

[20] Ahmed Eldawy, Justin J. Levandoski, and Per-Åke Larson. 2014. Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. *PVLDB* 7, 11 (2014), 931–942.

[21] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database Compression on Graphics Processors. *PVLDB* 3, 1 (2010), 670–680.

[22] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[23] Florian Funke, Alfons Kemper, and Thomas Neumann. 2012. Compacting Transactional Data in Hybrid OLTP & OLAP Databases. *PVLDB* 5, 11 (2012), 1424–1435.

[24] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Pierre Le Bodic, Stephen J. Maher, Frederic Matter, Matthias Miltenberger, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, and Jakob Witzig. 2020. The SCIP Optimization Suite 7.0. http://www.optimization-online.org/DB_HTML/2020/03/7705.html Accessed: 2022-01-23.

[25] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, and David A. Patterson. 2009. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *Proceedings of the 25th International Conference on Data Engineering, ICDE*. 592–603.

[26] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering, ICDE*. 370–379.

[27] Goetz Graefe and Leonard D Shapiro. 1991. Data compression and database performance. In *Proceedings of 1991 Symposium on Applied Computing*. IEEE, 22–27.

[28] LLC Gurobi Optimization. 2022. Gurobi Optimizer Reference Manual. http://www.gurobi.com Accessed: 2022-01-23.

[29] Stefan Halfpap and Rainer Schlosser. 2019. Workload-Driven Fragment Allocation for Partially Replicated Databases Using Linear Programming. In *35th IEEE International Conference on Data Engineering, ICDE*. 1746–1749.

[30] Theo Härder. 1978. Implementing a Generalized Access Path Structure for a Relational Database System. *ACM Trans. Database Syst.* 3, 3 (1978), 285–298.

[31] Jayant R. Haritsa. 2020. Robust Query Processing: Mission Possible. *PVLDB* 13, 12 (2020), 3425–3428.

[32] Linus Heinzl, Ben Hurdelhey, Martin Boissier, Michael Perscheid, and Hasso Plattner. 2021. Evaluating Lightweight Integer Compression Algorithms in Column-Oriented In-Memory DBMS. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB*. 26–36.

[33] Herodotos Herodotou and Shivnath Babu. 2010. Xplus: A SQL-Tuning-Aware Query Optimizer. *PVLDB* 3, 1 (2010), 1149–1160.

[34] Benjamin Hilprecht and Carsten Binnig. 2021. One Model to Rule them All: Towards Zero-Shot Learning for Databases. In *11th Conference on Innovative Data Systems Research, CIDR, Online Proceedings*.

[35] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD*. 143–157.

[36] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD*. 535–550.

[37] Intel. 2022. Intel Extension for Scikit-learn. https://intel.github.io/scikit-learn-intelex/ Accessed: 2022-01-23.

[38] Balakrishna R. Iyer and David Wilhite. 1994. Data Compression Support in Databases. In *Proc. VLDB*. 695–704.

[39] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A. Chien, Jihong Ma, and Aaron J. Elmore. 2021. Good to the Last Bit: Data-Driven Encoding with CodecDB. In *SIGMOD '21: International Conference on Management of Data*. 843–856.

[40] Alfons Kemper, Thomas Neumann, Florian Funke, Viktor Leis, and Henrik Mühe. 2012. HyPer: Adapting Columnar Main-Memory Data Management for Transactional AND Query Processing. *IEEE Data Eng. Bull.* 35, 1 (2012), 46–51.

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

[42] 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. *PVLDB* 13, 11 (2020), 2382–2395.

[43] Jan Kossmann, Alexander Kastius, and Rainer Schlosser. 2022. SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT*.

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

[45] Robert Lasch, Robert Schulze, Thomas Legler, and Kai-Uwe Sattler. 2021. Workload-Driven Placement of Column-Store Data Structures on DRAM and NVM. In *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN*. 5:1–5:8.

[46] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *34th IEEE International Conference on Data Engineering, ICDE*. 185–196.

[47] Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. 2010. Speeding Up Queries in Column Stores - A Case for Compression. In *Data Warehousing and Knowledge Discovery, 12th International Conference, DAWAK*. 117–129.

[48] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. 2012. Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques. *PVLDB* 5, 11 (2012), 1555–1566.

[49] Robin Lougee-Heimer. 2003. The common optimization interface for operations research: Promoting open-source software in the operations research community.

IBM Journal of Research and Development 47, 1 (2003), 57–66.

[50] Lin Ma et al. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proc. SIGMOD*. 631–645.

[51] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. 2021. MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems. In *SIGMOD '21: International Conference on Management of Data*. 1248–1261.

[52] Roger MacNicol and Blaine French. 2004. Sybase IQ Multiplex - Designed For Analytics. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB*. 1227–1230.

[53] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *PVLDB* 12, 11 (2019), 1705–1718.

[54] Ryan C. Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *PVLDB* 12, 11 (2019), 1733–1746.

[55] Subhash C. Narula and John F. Wellington. 1977. Prediction, Linear Regression and the Minimum Sum of Relative Errors. *Technometrics* 19, 2 (1977), 185–190.

[56] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR, Online Proceedings*.

[57] K. Nimkanjana, S. Vanichayobon, and W. Wettayaprasit. 2008. Auto-Indexing Selection Technique in Databases under Space Usage Constraint Using FP-Growth and Dynamic Programming. In *International Conference on Computer and Electrical Engineering*. 932–935.

[58] Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). 2016. *Proceedings of the 2016 International Conference on Management of Data, SIGMOD*. ACM.

[59] 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. *PVLDB* 11, 6 (2018), 663–676.

[60] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[61] Hasso Plattner. 2014. The Impact of Columnar In-Memory Databases on Enterprise Systems. *Proc. VLDB Endow.* 7, 13 (2014), 1722–1729.

[62] Meikel Pöss, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why You Should Run TPC-DS: A Workload Analysis. In *Proceedings of the 33rd International Conference on Very Large Data Bases*. 1138–1149.

[63] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *PVLDB* 11, 2 (2017), 230–242.

[64] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science - Towards Embedded Analytics. In *10th Conference on Innovative Data Systems Research, CIDR, Online Proceedings*.

[65] 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.

[66] Keven Richly. 2021. Memory-Efficient Storing of Timestamps for Spatio-Temporal Data Management in Columnar In-Memory Databases. In *Database Systems for Advanced Applications - 26th International Conference, DASFAA, Proceedings, Part I*. 542–557.

[67] Keven Richly, Rainer Schlosser, and Martin Boissier. 2021. Joint Index, Sorting, and Compression Optimization for Memory-Efficient Spatio-Temporal Data Management. In *37th IEEE International Conference on Data Engineering, ICDE*. 1901–1906.

[68] Mark A. Roth and Scott J. Van Horn. 1993. Database Compression. *SIGMOD Record* 22, 3 (1993), 31–39.

[69] David Schwalb, Martin Faust, Johannes Wust, Martin Grund, and Hasso Plattner. 2014. Efficient Transaction Processing for Hyrise in Mixed Workload Environments. In *Proceedings of the 2nd International Workshop on In Memory Data Management and Analytics, IMDM*. 16–29.

[70] Chris Tofallis. 2008. Least Squares Percentage Regression. *Journal of Modern Applied Statistical Methods* 7, 2 (2008), 526–534.

[71] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proceedings of the 16th International Conference on Data Engineering, ICDE*. 101–110.

[72] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems. *PVLDB* 13, 11 (2020), 2662–2675.

[73] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. 2000. The Implementation and Performance of Compressed Databases. *SIGMOD Rec.* 29, 3 (2000), 55–67.

[74] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB* 2, 1 (2009), 385–394.

[75] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD*. 1567–1581.