Learned What-If Cost Models for Autonomous Clustering

Daniel Lindner, Alexander Löser, and Jan Kossmann

Hasso Plattner Institute, University of Potsdam, Germany
{daniel.lindner,alexander.loeser}@student.hpi.de
jan.kossmann@hpi.de

Abstract. Clustering database tables, i.e., storing similar values in physical proximity, can be used to enhance the performance of a database system. Determining optimal clustering configurations manually is cumbersome due to the large solution space and because the performance impact depends on the workload at hand. Autonomous clustering approaches can support database administrators in this task. However, determining the optimal clustering by physically applying all possible clusterings is prohibitively expensive. We present a what-if-based learned cost model that *simulates* the effects of a hypothetical clustering and then *estimates* the resulting workload latency based on previously trained cost models. Optimal clustering configurations can be found with these estimations. Our model accurately estimates the latencies of the TPC-H workload for different clusterings with a relative error of at most 5 %.

Keywords: Physical Database Design · Database Clustering · Autonomous Database · Learned Cost Models.

1 Introduction

The physical data layout of a database table has a large impact on the database performance [7]. Database clustering is the process of reorganizing a table's physical layout in a way that tuples with similar values in certain columns are grouped and stored together. Thereby, clustering increases data locality and may enable further optimizations, such as partition pruning [9]. Database vendors such as Oracle [23], IBM [14], Snowflake [19], and Microsoft [2] have acknowledged the importance of clustering, and provide tools to cluster the stored data by one or multiple columns. However, real-world databases often contain wide tables with numerous columns [3,13], e.g., enterprise resource planning systems consist of tables with hundreds of columns. In such scenarios, a tremendous number of clustering options exists, which makes it challenging to determine the clustering that results in the best performance. Traditionally, this choice is made by database administrators [10] who must have detailed knowledge of the database and its workload.

Autonomous databases promise to simplify the work of database administrators [18]. Not only does the DBMS *itself* decide on its configuration, but it may

even identify settings superior to human decisions: The database has the most profound knowledge of workload and data and thus has also the best rationale for decisions [22]. In general, autonomous systems simulate multiple configuration options. To allow weighing these options, learned cost models are a fundamental part of autonomous databases [18]. They estimate the run time of a database operation based on particular features, and thus are crucial for the quality of the decisions. As they can be re-trained when the code-base or the underlying hardware changes, they are independent of implementation and hardware. Learned cost models have been used for various applications, e.g., query optimization, index selection, or transaction management [22].

Contributions In this work, we use a learned cost model to identify how a specific clustering impacts the database's performance. We use a what-if-based approach, i.e., we do not actually change the clustering and measure the resulting performance. Instead, we first perform a rule-based clustering simulation that predicts the effect of a clustering on certain database operations (e.g., whether the order of table scans changes). In the second step, a learned cost model estimates the workload's latency. Such a cost model can be integrated into a what-if-based clustering model, e.g., the clustering model by Löser [15].

The remainder of this paper is structured as follows: Section 2 introduces basic concepts and fundamentals for Hyrise, the DBMS we used for our investigations. In Section 3, we present our approach on what-if-based latency estimation. After describing how we obtain training data in Section 3.3, Section 4 evaluates both steps of our model, the clustering simulation in Section 4.1 and the learned cost model in Section 4.2. Section 5 discusses related work in the areas of autonomous clustering and learned cost models. Finally, Section 6 sums up our insights and concludes our work.

2 The Hyrise Database System

This work was developed for the research database Hyrise[1] [6]. This section provides an overview of the parts and concepts of the Hyrise database system that are necessary for understanding our approach presented in Section 3.

Storage layout Hyrise is a relational, non-distributed, memory-resident research database. Tables are stored in a column-oriented fashion but are partitioned into *chunks* with a fixed number of rows, 65 535 by default. Thus, each chunk contains a fraction of all columns. Those fractions are called *segments*. For each segment, Hyrise stores aggregated segment information [17], such as the minimum and maximum value present in the segment. During query optimization, the minimum and maximum values may be used to prune chunks, i.e., to exclude chunks from the query execution without scanning their actual segment data. Figure 1 visualizes Hyrise's storage layout.

[1] `https://github.com/hyrise/hyrise`

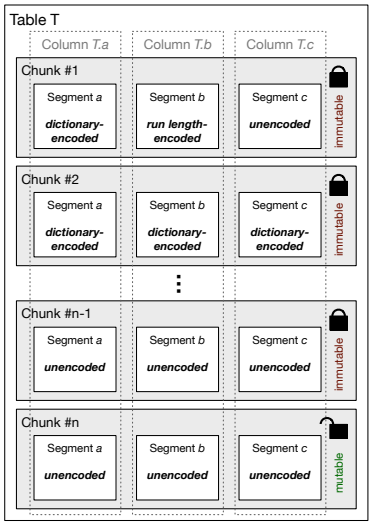

Fig. 1. Visualization of Hyrise's storage layout. Reprinted from *Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management* [6].

Within a segment, data may be stored in an encoded format. Hyrise supports a variety of encoding types, e.g., dictionary encoding [1] (default), run length encoding, or frame of reference encoding. All measurements in this work were performed with dictionary-encoded segments.

SQL Pipeline Hyrise can be queried via SQL. When Hyrise receives an SQL query, the *SQL pipeline* is triggered: First, the SQL query is translated into a logical query plan, called LQP. An LQP is a directed acyclic graph (DAG) whose nodes represent operations of the relational algebra, such as joins or aggregates. The LQP is then optimized by the query optimizer and finally translated into a physical query plan, called PQP. A PQP is a DAG whose nodes are specific implementations of database operations. For example, a join in the LQP may be translated to a hash join, a sort-merge join, or a nested loop join in the PQP. In Hyrise, those specific implementations are called *operators*.

An operator may receive none, one, or two other operators as input. The leaves of the PQP are `GetTable` operators, a particular Hyrise operator that receives only a table name as input and returns the table with the given name. To execute an SQL query, Hyrise triggers the SQL Pipeline to obtain a PQP and then executes the PQP bottom up. The result of the query is the output of the root PQP node. Hyrise maintains a cache of recently executed query plans, called *PQP cache*.

Reference segments In Hyrise, the output of an operator is a temporary table. Usually, these tables are *reference tables*, which contain *reference segments*. They

do not store the data, but only point to where the data is located in the original table. There is always at most one level of indirection, i.e., if an operator receives a table with reference segments as input and outputs a table with reference segments, the output segments still directly point to the original table. The process of accessing the actual values behind a reference segment is called *materialization*. Figure 2 visualizes the concept of reference segments.

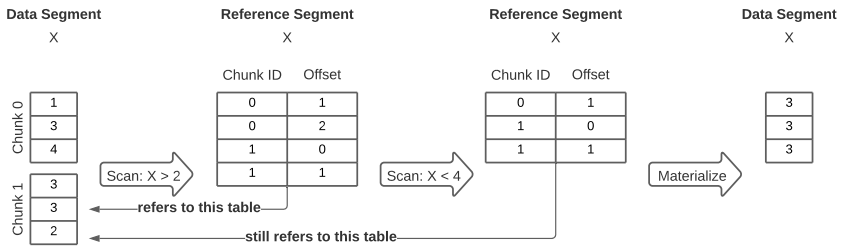

Fig. 2. Visualization of Hyrise's reference segments. Reprinted from *Automatic Clustering in Hyrise* [15].

Operator Performance Data In Hyrise, operators collect statistics about their execution. These statistics include, e.g., input and output row counts, the execution time, and the accessed columns and are available via the *PQP cache*. In our clustering model, we use the entirety of all executed operators and their performance data to represent workloads.

3 What-If Latency Estimation for Autonomous Clustering

This section presents our approach on what-if-based latency estimation for autonomous clustering. While we provide a brief overview of the entire clustering model in Section 3.1, this work focuses on our latency estimation technique and its training data, which we describe in Section 3.2 and Section 3.3, respectively.

3.1 Clustering Model

We use the clustering model that was developed during the master's thesis of Löser [15]. Given the current database workload, the model aims to find a clustering that maximizes the database's performance. There are different ways to define performance. Analogous to Löser, we choose the workload's latency, i.e., the sum of all individual query latencies, as our performance metric, so maximizing performance is equivalent to minimizing the workload's latency.

The model operates in two steps: First, the *candidate creator* analyzes the current workload and determines a list of possible clustering candidates. Second, for each clustering candidate, the *latency estimator* is used to estimate the impact of such a clustering on the current workload's performance. The database can then weight the expected performance gain against the expensive process of reclustering.

Candidate creator The first step towards creating clustering candidates is to identify *interesting columns*. In a nutshell, a column is considered interesting if a scan predicate or a join predicate operates on it. Table scans on a clustered column can benefit, e.g., from pruning [7]; hash-based joins on a clustered column may benefit through better cache hit rates.

An arbitrary number of these interesting columns can be chosen for clustering, thereby allowing for n-dimensional clusterings. However, in this work, we only consider one-dimensional clusterings, i.e., with exactly one clustering column.

Latency estimator The most precise way to determine a clustering's performance impact is to physically implement the clustering, and measure the latency. Since physically implementing a clustering is a very expensive process, we consider it an unfeasible approach for production databases, especially if a large number of clusterings has to be considered.

Consequently, the latency estimator works on a what-if basis: Instead of physically applying a certain clustering and measuring its actual performance impact, the latency estimator only assumes the clustering was implemented and provides a latency estimate.

Löser [15] uses a set of handcrafted rules to estimate the latency of table scans and hash joins. In this work, we replace the handcrafted rule system with a learned cost model that can predict latencies for table scans, hash joins, and hash aggregates.

3.2 Latency Estimation

Determining how a certain clustering will affect the current workload's latency is a crucial part of our clustering model. The latency estimation is conducted per operator for aggregates, scans, and joins. For all other operators, we assume that their latency is unaffected by the clustering. This simplifying assumption is justified by the fact that the aforementioned three operators are responsible for more than 90 % of the execution time of the TPC-H benchmark [7].

Our latency estimation approach consists of two steps: First, we perform a *clustering simulation*. In a clustering simulation, we apply a set of rule-based transformations to the operators. The aim is to predict how a hypothetical clustering will affect the operators' clustering-dependent features, such as the number of input rows they receive. For example, when estimating the latency for clustering by column X, the simulation would consider an input reduction

through pruning for filter predicates that operate on column X. The clustering simulation is described in more detail in Section 3.2.

In the second step, we perform the actual latency estimation using learned physical cost models. The cost models receive the clustering simulation's output as input and yield latency estimates as output. Figure 3 displays the latency estimation process.

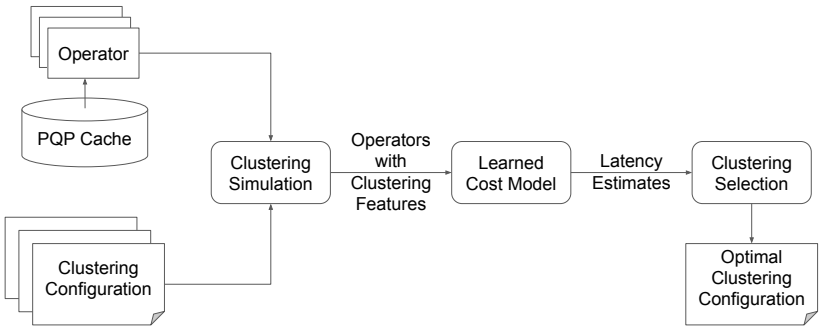

Fig. 3. Architecture of the latency estimation.

Clustering Simulation In Hyrise, operators track certain performance data, such as the number of input rows, output rows, or the input data's sort order. Given a workload (i.e., a set of operators), our clustering simulation applies a predefined set of rules to identify the operators impacted by the clustering and estimates how their input rows, output rows, sortedness, and operator-specific features will change. We identified two aspects that need to be considered for a precise clustering simulation: PQP awareness and predicate reordering.

Awareness of surrounding operators For some operators, it might be sufficient to consider only the operator itself to determine whether the clustering will influence its performance. For example, if a filter predicate occurs on a particular column, clustering by that column reduces the input size of scans that operate on it through chunk pruning. In general, however, we found that it is crucial to be aware of an operator's position in the physical query plan, and, thereby, its surrounding operators.

Consider the following excerpt from Hyrise's query plan for query 20 of the analytical TPC-H benchmark [5,21], which is is visualized in Figure 4: Hyrise first performs semi joins on the `l_partkey` and `l_suppkey` columns of the `lineitem` table, before performing a scan on the `l_shipdate` column. By default, the `lineitem` table is sorted by the `l_orderkey` column. If we assume that `lineitem` was clustered by `l_shipdate` instead, it is obvious that the scan on `l_shipdate` will benefit from pruning. The semi joins are executed before the scan; however, the scan enables chunk pruning, which Hyrise applies during query optimization,

i.e., even before executing the joins. Thus, despite the semi joins neither operate on `l_orderkey` nor on `l_shipdate`, clustering by `l_shipdate` has a significant impact on their input size, reducing it by factor 6. Embedding these operators into the surrounding PQP is required to obtain precise information about run time-critical properties.

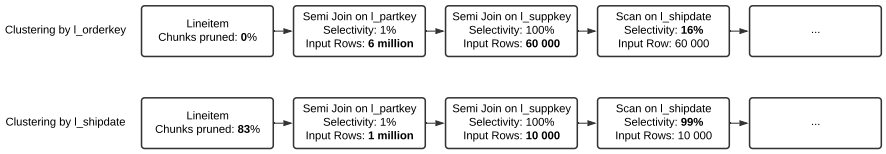

Fig. 4. Partial PQP of TPC-H Query 20 at scale factor 1, with `l_orderkey` and `l_shipdate` clustering. Reprinted from *Automatic Clustering in Hyrise* [15].

Predicate Reordering There is another crucial aspect to consider: predicate reordering. In Hyrise, an operator is considered a *predicate* when it filters its input table. The most important types of predicates are table scans and semi joins.

One of the steps performed by Hyrise's query optimizer is the reordering of predicates: Predicates are re-arranged so that the predicate with the highest selectivity is executed last. If a scan operates on a clustered column, large parts of the non-matching values may have been pruned. Consequently, the scan will have a selectivity close to 1 and be executed last, i.e., its input size will be further reduced by other predicates. Additionally, the reordering may cause another predicate to operate on data segments instead of reference segments and vice versa, which should be considered in the clustering simulation. Figure 5 visualizes a predicate reordering with a change of segment types at the example of query 6 of the analytical TPC-H benchmark.

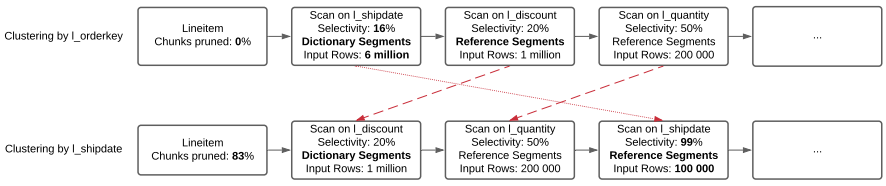

Fig. 5. Partial PQP of TPC-H Query 6 at scale factor 1, with `l_orderkey` and `l_shipdate` clustering. Reprinted from *Automatic Clustering in Hyrise* [15].

Learned Cost Model We perform the actual latency estimation using a learned cost model, i.e., a machine learning model is used to predict the operator run times. In this work, we use a gradient boosting regressor [8] with a Huber loss function [11] provided by scikit-learn[2]. Boosting allows for different types of dependencies within variables, e.g., linear, quadratic, or logistic relationships, without explicitly modeling these types [12, chapter 8].

In Hyrise, operators can have multiple implementations. E.g., the `TableScan` has five specialized implementations for different types of predicates. For each operator implementation, we use a dedicated regressor, i.e., we train five models for these five implementations. Furthermore, we observed that the type of input tables, i.e., if a table contains data or is an intermediate reference table, has a strong influence on the performance of `JoinHash`. Thus,we add models for each combination of table types of the build and probe input table for this operator. A complete overview of the features used to predict the operator latency can be found on GitHub[3].

3.3 Obtaining Data for Training

When Hyrise executes a given workload, i.e., a set of SQL queries, the resulting PQPs are stored in the PQP cache. Combined with information about the current clustering, we use the performance data of the PQPs' operators as our training data.

Specifically, we use the current workload in combination with the currently implemented clustering as a part of the training data. In our case, this is the workload defined by the TPC-H benchmark, with all tables being left as generated. By default, the `lineitem` table is sorted by `l_orderkey`.

Clustering may lead to changes of the PQPs, e.g., we have seen that both input sizes and input table types of a `TableScan` may change due to the clustering in Figure 5. For operators that are not affected by clustering, the execution data of the current workload is sufficient to predict their run time. However, if an operator is affected by the clustering, its features (e.g., input size) may be unlike any the model has seen during training. Consequently, additional training data is required.

Thus, we add dedicated *calibration data*. Calibration data contain execution information of synthetic PQPs, covering variations of the cost model's input features. For example, we add table scans on different input sizes, i.e., with a variable amount of pruned chunks, to our training data. The calibration data also covers different input sizes for semi joins and aggregates. Overall, we include the execution information of more than 90 000 dedicated PQPs.

Additionally, our calibration data contain performance data for the TPC-H workload executed on a randomly shuffled `l_lineitem` table. These data are useful for, e.g., joins that operate on sorted data in the current clustering, but would not receive sorted data in a hypothetical new clustering.

[2] `https://scikit-learn.org/stable/modules/ensemble.html`
[3] `https://github.com/aloeser/hyrise/blob/whatif/cost_model_features.md`

3.4 Limitations

Our latency estimation process has two limitations. First and foremost, we currently consider only one-dimensional clusterings. Second, our learned cost model can hardly interpolate observations, and thus does not yield reasonable estimates for previously unseen operators. However, this is only a limitation our chosen machine learning model type.

4 Results and Discussion

In this section, we evaluate our latency estimation approach. For that purpose, we use our model to predict how the latencies of the TPC-H benchmark will change, assuming that the `lineitem` table was sorted, i.e., clustered, by another column than `l_orderkey` (the default). We operate on the `lineitem` table because it is the largest of the TPC-H tables, and responsible for the major share of execution time. For the clustering columns, we consider a column mostly used in filter predicates, `l_shipdate`, and a column mostly used for joins, `l_partkey`.

Our model was developed based on TPC-H data with a scale factor of 1. For that reason, the evaluation is focused on data with a scale factor of 1, too, but we also evaluate data with a scale factor of 10 to determine how well our model scales. Our implementation is publicly available on GitHub[4].

Like the approach itself, our evaluation is divided into two steps: We evaluate the clustering simulation's precision in Section 4.1 and the latency estimates' precision in Section 4.2.

4.1 Clustering Simulation

For scans and joins, we evaluate our predictions of input rows, output rows, and segment types, i.e., whether the operator receives dictionary or reference segments. For the input and output rows, we consider the relative errors, i.e., the ratio between predicted and actual size; for the segment type, we consider the ratio of correctly predicted segment types.

Table Scans By default, Hyrise stores data in dictionary encoded segments. The dictionaries of those segments contain a list of all unique values present in the segment, which enables an early-out optimization for scans: For range predicates such as the SQL `BETWEEN` keyword, the dictionaries can be used to determine whether none, all, or some of the segment's values will match. In particular, if both the minimum and the maximum value of the dictionary are within the range specified by the predicate, all values in the segment will match. Consequently, Hyrise only has to write the output, without actually scanning (and materializing) the segment. In addition to input rows, output rows, and the segment type, our clustering simulation also predicts whether the early-out optimization is applicable. Table 1 shows the precision of our clustering

[4] `https://github.com/aloeser/hyrise/tree/db73923`

simulation for scans that operate on `lineitem`. First of all, we observe that the simulation yields precise results for a large majority of the table scans. We further observe that the simulation is almost perfect for the `l_partkey` clustering. This can be expected, as neither clustering by `l_partkey` nor by `l_orderkey` allows any pruning or sorted searches; i.e., the features of the table scans remain unaffected.

Clustering	Input rows		Output rows		Segment type	Early-out applicable
	≤ 10%	≤ 20%	≤ 10%	≤ 20%		
l_shipdate	75%	86%	83%	93%	97%	90%
l_partkey	99%	99%	99%	99%	100%	100%

Table 1. Aggregated statistics for the clustering simulation of scans on `lineitem`. For the input and output rows, it lists the share of operators for which the prediction is within 10 % (or 20 %, respectively) of the actual value. For the segment type and early-out optimization (binary values), it displays the share of correctly predicted values.

The simulation achieves similar performance on data with a scale factor of 10. We conclude that our clustering simulation is sufficiently precise for table scans, even for higher scale factors.

Hash joins For the joins, we evaluate the input rows, output rows, and the segment type of the input tables. If a join operates on reference segments, we additionally predict (and evaluate) the number of referenced chunks, i.e., the number of chunks of the original table that the join's input refers to. Intuitively, a join which receives 1 000 input rows that refer to 1 000 different chunks should be slower than a join whose 1 000 input rows refer to only two chunks, as we need to access fewer dictionaries when materializing the reference segments. Hyrise's operators already track their input chunk count; however, this metric is highly misleading: Depending on the build table's size, hash joins in Hyrise may apply an r-Bit radix partitioning [4], which moves data across chunk boundaries. The output of a hash join with radix partitioning contains at most 2^r chunks, independent of the number of referenced segments. Table 2 shows the precision of our clustering simulation for joins that operate on `lineitem`. We observe that the clustering simulations yields precise results for the vast majority of the hash joins. Analogous to the table scans, changes in the input size of joins only occur through pruning; which explains why simulation performs better for the `l_partkey` clustering.

The simulation achieves similar performance on data with a scale factor of 10. We conclude that our clustering simulation yields precise results for hash joins, even for higher scale factors.

Evaluation of Sortedness An essential part of the clustering simulation is to determine whether the original sort order of a table is still maintained. Hyrise does

Clustering	Input rows		Output rows		Segment type	Referenced chunks	
	≤ 10%	≤ 20%	≤ 10%	≤ 20%		≤ 10%	≤ 20%
l_shipdate	96%	96%	99%	99%	100%	92%	92%
l_partkey	100%	100%	100%	100%	100%	100%	100%

Table 2. Aggregated statistics for the clustering simulation of hash joins that operate on `lineitem` (as probe or build side). For the input rows, output rows, and the number of referenced chunks, it lists the share of operators for which the prediction is within 10 % (or 20 %, respectively) of the actual value. For the segment type it displays the share of correctly predicted values.

track whether a segment is flagged as sorted; however, this flag is set very conservatively. Only certain operators, such as table scans, forward the sortedness flags, while other operators, such as hash joins, do not forward the flag, even if they do maintain the original sort order. As a consequence, we cannot provide a meaningful evaluation for our sort order predictions.

Aggregates In Hyrise, aggregates usually occur in the PQP after scans and joins are executed. As a consequence, the input and output size of aggregates cannot be modified by the clustering. This implies that we only need to predict whether an aggregate's input table still has the same sort order as the original table. However, as explained in the previous paragraph, such an evaluation is not possible.

4.2 Learned Cost Model

After evaluating the quality of the learned cost model's input data, we investigate its run time estimates. To identify the most suitable clustering regarding a workload scenario, we focus on the aggregated latencies for each clustering candidate. Since we considered three affected operators, `TableScan`, `JoinHash`, and `AggregateHash`, we limit our examination to them. Furthermore, we only take operators on the `lineitem` table into account, as they are the only ones affected by one-dimensional clusterings of this table.

We have already introduced the `l_shipdate` and the `l_partkey` columns as clustering candidates and the `l_orderkey` column as the original clustering column. The accumulated run times for the three operators are displayed in Figure 6.

For all three clustering candidates, the learned cost model estimates the overall latencies with a relative error of at most 5 %. Though the joins are consistently over-estimated, the aggregates' run times are estimated too low for two clusterings. This is also the case for the `l_orderkey` clustering, which was part of the training data. Though we have both under- and over-estimations, the most important information statement is precise: The order of the clustering candidates' run times is preserved, and our clustering model suggests the optimal one-dimensional clustering correctly.

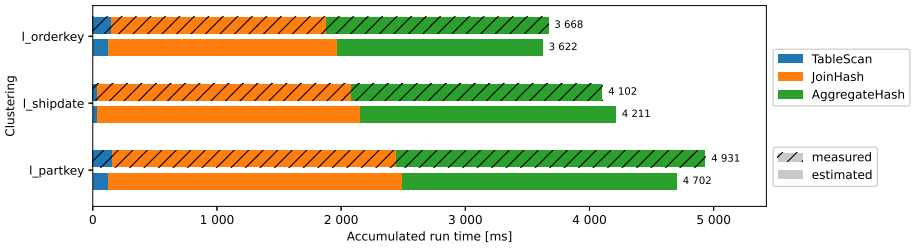

Fig. 6. Accumulated run times of selected operators on `lineitem`, estimated vs. measured, for three clusterings.

Table 3 shows the mean squared error (MSE) of the accumulated latencies for each clustering and operator. Though these MSEs are high compared to the average run times of the operators, we have already seen that the accumulated latencies are accurate.

Clustering	TableScan	JoinHash	AggregateHash
l_orderkey	11 / 9	39 / 40	1 085 / 94
l_shipdate	1 / 2	346 / 48	1 574 / 106
l_partkey	14 / 10	167 / 54	2 258 / 131

Table 3. Mean squared error (MSE) in ms^2 of latency estimations / average run time in ms for chosen operators and clusterings.

For a scale factor of 10, the results are comparable: The estimations of the accumulated run times of the three operators are exceeded by maximal 7 %. This small overall relative error is achieved by a consistent under-estimation of aggregates and over-estimation of joins. Though we focused on modeling calibration data suitable for a scale factor of 1, this is a satisfying result.

Overall, our model predicts the latency of the workload for given clustering scenarios with an adequate accuracy: We obtain the correct order for clustering candidates and the estimations are precise enough to make a deliberate decision regarding the application of a clustering.

5 Related Work

Recently, there have been advancements in both research areas, learned cost models and database clustering.

Zhou et al. [22] give an overview of current applications of learned cost models in databases. These cost models are used to tackle various problems, e.g., query optimization, index selection, or transaction management. However, none of the

cited authors use machine learning techniques for latency estimations of different clusterings.

Marcus et al. [16] and Sun et al. [20] present cost models to estimate query latencies with tree-structured deep neural networks. These models have a high precision, but require a lot of training time, and they infer knowledge about the current data layout, which hinders the simulation of hypothetical data layouts.

Hilprecht et al. [10] present a clustering model for cloud databases. They argue that exact latency estimates are essential for identifying an optimal clustering. However, latency estimation is a difficult task; even more so for cloud databases, which often use heterogeneous hardware. As a consequence, they suggest a model based on deep reinforcement learning: the model learns the latencies of different clusterings by implementing a clustering and measuring its latency. However, when considering a large number of clusterings, the training process becomes very expensive. To speed up the training, Hilprecht et al. argue that – being in a cloud environment – the latency of transferring data between different nodes accounts for the dominating part of the query latencies. Thus, for an approximate latency estimate, clusterings need not be physically implemented; it is sufficient to estimate the amount of data that needs to be transferred. However, this cost model cannot be applied to non-distributed database systems like Hyrise, as no data is transferred over the network.

6 Conclusion and Future Work

This paper presented a what-if model for clustering-specific latency estimation, consisting of a clustering simulation and a learned cost model for latency estimation. The resulting system is a hybrid approach: The results of the rule-based clustering simulation are the input of a gradient boosting regressor to estimate the latencies of a given workload. These latencies are estimated with a relative error of at most 5 % for the TPC-H workload with a scale factor of 1, and our model correctly identifies the optimal one-dimensional clustering. Compared to rule-based approaches, the learned cost model is more flexible and can be retrained if the implementations of the database operators change or if the system runs on different hardware.

Nevertheless, there are still aspects that are subject to future work. First, the model only considers one-dimensional clusterings. Second, we focused on one workload and one specific type of learned cost model. Additional workloads and calibration data covering a more extensive range of characteristics can generalize the stated observations. Other types of learned cost models, which support interpolation and fine-tuning in a more sophisticated manner, could further improve latency estimation. Third, the rule-based clustering simulation could also be automated, e.g., by incorporating the database optimizer.

References

1. Abadi, D.J., et al.: Integrating compression and execution in column-oriented database systems. In: SIGMOD. pp. 671–682 (2006)

2. Agrawal, S., Narasayya, V.R., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: SIGMOD. pp. 359–370 (2004)
3. Bian, H., et al.: Wide table layout optimization based on column ordering and duplication. In: SIGMOD. pp. 299–314 (2017)
4. Boncz, P.A., et al.: Database architecture optimized for the new bottleneck: Memory access. In: VLDB. pp. 54–65 (1999)
5. Boncz, P.A., et al.: TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In: Performance Characterization and Benchmarking - TPC Technology Conference. vol. 8391, pp. 61–76 (2013)
6. Dreseler, M., et al.: Hyrise re-engineered: An extensible database system for research in relational in-memory data management. In: EDBT. pp. 313–324 (2019)
7. Dreseler, M., et al.: Quantifying TPC-H choke points and their optimizations. PVLDB **13**(8), 1206–1220 (2020)
8. Friedman, J.H.: Greedy function approximation: A gradient boosting machine. The Annals of Statistics **29**(5), 1189–1232 (2001)
9. Herodotou, H., et al.: Query optimization techniques for partitioned tables. In: SIGMOD. pp. 49–60 (2011)
10. Hilprecht, B., et al.: Learning a partitioning advisor for cloud databases. In: SIGMOD. pp. 143–157 (2020)
11. Huber, P.J.: Robust Estimation of a Location Parameter. The Annals of Mathematical Statistics **35**(1), 73 – 101 (1964)
12. James, G., et al.: An Introduction to Statistical Learning with Applications in R. Springer (2013)
13. Krüger, J., et al.: Fast updates on read-optimized databases using multi-core cpus. PVLDB **5**(1), 61–72 (2011)
14. Lightstone, S., Bhattacharjee, B.: Automated design of multidimensional clustering tables for relational databases. In: VLDB. pp. 1170–1181 (2004)
15. Löser, A.: Automatic Clustering in Hyrise. Master's thesis, Hasso-Plattner-Institute, University of Potsdam (2020)
16. Marcus, R.C., Papaemmanouil, O.: Plan-structured deep neural network models for query performance prediction. PVLDB **12**(11), 1733–1746 (2019)
17. Moerkotte, G.: Small materialized aggregates: A light weight index structure for data warehousing. In: VLDB. pp. 476–487 (1998)
18. Pavlo, A., et al.: Self-driving database management systems. In: 8th Biennial Conference on Innovative Data Systems Research, CIDR Online Proceedings (2017)
19. Snowflake: Clustering Keys & Clustered Tables (2020), `https://docs.snowflake.com/en/user-guide/tables-clustering-keys.html`
20. Sun, J., Li, G.: An end-to-end learning-based cost estimator. PVLDB **13**(3), 307–319 (2019)
21. Transaction Processing Performance Council: TPC-H Specification (2014), `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf`
22. Zhou, X., et al.: Database meets artificial intelligence: A survey. TKDE (2020)
23. Ziauddin, M., et al.: Dimensions based data clustering and zone maps. PVLDB **10**(12), 1622–1633 (2017)