Workload-driven, Lazy Discovery of Data Dependencies for Query Optimization

Jan Kossmann
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
Jan.Kossmann@hpi.de

Daniel Lindner
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
Daniel.Lindner@student.hpi.de

Felix Naumann
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
Felix.Naumann@hpi.de

Thorsten Papenbrock
Philipps University of Marburg
Marburg, Germany
Papenbrock@informatik.uni-marburg.de

ABSTRACT

Effective query optimization is the backbone of relational database systems; query optimizers use all manners of tricks, from specialized auxiliary data structures over caching and batch processing to the use of secondary metadata. Nevertheless, these systems do not fully exploit the potential of data dependencies, such as functional dependencies, although dozens of dependency-based query optimization techniques exist. This disregard occurs because the required data dependencies are, in practice, hard to find and hard to maintain.

This paper presents our vision of a workload-driven, lazy dependency discovery system for query optimization. We propose a lightweight approach that identifies relevant data dependency candidates based on actually executed query plans, validates the candidates dynamically against the database, and maintains the results using different strategies that also exploit concepts of columnar DBMSs. Our evaluation demonstrates the feasibility of this approach and the potential of dependency-based optimizations, using a prototypical implementation in Hyrise that implements three exemplary dependency-based optimization techniques. For example, after automatic dependency discovery, these optimizations reduce the Join Order Benchmark's execution time by 27 %.

KEYWORDS

query optimization, data dependencies, unique column combinations, functional dependencies, order dependencies, data profiling

1 DATA DEPENDENCIES FOR OPTIMIZATION

The utilization of metadata is an important pillar of relational query optimization. We propose an efficient application for the largely untapped query optimization potential of *data dependencies*, such as functional dependencies or order dependencies. Commonly, for any given dataset, many data dependencies [24] of different types are valid, due to natural correlations in the data, but also due to schema denormalization, and certain data generation patterns. Furthermore, dozens of data dependency-based query optimization techniques exist [11], most of which have been known for years. For example, data dependencies can be used to obtain improved selectivity estimates, to simplify `GROUP BY` statements, or to convert expensive joins into local predicates [11]. Yet, data dependencies and their query optimization strategies remain largely underutilized, because the database systems are not aware of these dependencies. The reason for this are three challenges, namely dependency *discovery*, *selection*, and *mutation*, that are still practically unsolved in the context of database management systems.

Considering the *discovery* challenge, dependencies derived from manually or schema-defined constraints, such as primary or foreign keys, make up only a fraction of all existing dependencies. Some dependencies, such as order dependencies, cannot be defined as constraints and, in some use cases, no manually defined constraints and keys exist. Many scientific datasets, for example, are provided as CSV files that may include column headers but no further metadata. Automatically determining data dependencies via data profiling algorithms is feasible, but expensive and can take hours of processing [1].

Systematically profiled dependencies, then, however, lead to the second challenge, which is dependency *selection*. Because data profiling algorithms discover *all* technically valid dependencies on a given relation instance, the result sets can become large enough that even storing and efficiently accessing them might exhaust a database. The 1m rows NCVoter dataset, for instance, contains a remarkable amount of 5m minimal FDs (and many further dependencies of other types) [19]. If the database can actually store all discovered dependencies, then it may still not be possible to find a certain dependency during query optimization fast enough, because their retrieval involves not only dependency lookup but also inference: discovered dependencies are usually minimal, but the dependencies required for query optimization may not be. A strategy to select only dependencies relevant for query optimization from discovered dependency sets does not exist yet.

Assuming that such a strategy would exist, *mutation* is still an open issue, because every `INSERT`, `UPDATE`, or `DELETE` statement can invalidate expensively mined dependencies, which makes them unsafe for query optimization; also, new and helpful dependencies might appear. Thus, dependency discovery, selection, and mutation, i.e., the entire metadata maintenance needs to be part of any such query optimization efforts.

We present an integrated and autonomous query optimization system that uses lazily discovered data dependencies for query rewriting and plan optimization. Based on observed workloads, the system discovers on-demand only such data dependencies that enable certain optimizations and are, therefore, relevant to the database system. It also incrementally maintains the discovered dependencies by dynamically adding new dependencies and re-evaluating existing dependencies over time. In this way, our system solves all three challenges: The *discovery* is integrated into the optimization process and, in this way, validates fewer and more suitable (i.e., also non-minimal) dependencies; the *selection* of dependencies is workload-driven and, hence, both effective and well tuned to actual needs; and an efficient *mutation* is possible due to the tight integration into and exploitation of database operations. In summary, this paper makes the following contributions:

(1) A workload-driven, lazy discovery system that serves relevant data dependencies to a query optimizer.
(2) A collection of incremental dependency maintenance techniques that keeps the query optimizer's metadata valid.
(3) A practical integration of the proposed discovery system with three data dependency-based query optimization techniques into the open-source DBMS Hyrise [6].
(4) An evaluation that demonstrates the performance of our system (and dependency-based query optimization): 27 % overall execution time improvement in the join order benchmark and more than 60 × speed-up for certain TPC-DS queries.

2 BACKGROUND

In this section, we provide necessary background information on data dependencies, dependency-based query optimization techniques, and the database system Hyrise.

2.1 Data dependencies

Data dependencies are well-defined relational properties. They express relationships between attributes usually within a table, but some dependency types can also span multiple tables. We now briefly introduce the three data dependencies that we use for query optimization in this paper. For further details, we refer the interested reader to [1]. The following definitions are largely adopted from [11]:

Unique column combinations (UCCs). The attribute set `X` forms a unique column combination, if a projection on `X` does not contain any duplicate tuples. More formally, given a relational instance r over a relation R, $X \subseteq R$ is *unique*, i.e., a *column combination* (UCC) for R, iff $\forall r_i, r_j \in R, i \neq j : r_i[X] \neq r_j[X]$. A popular example for UCCs are all relational database keys, such as a synthetic `id` column or a combination of a `customer_id` and a `timestamp` in an `orders` table.

Functional dependencies (FDs). According to a functional dependency $X \longrightarrow Y$, all tuples that agree on their X values also agree on their Y values. Again, more formally, a *functional dependency* (FD) $X \longrightarrow Y$ of a relation R holds in a relational instance r over R, iff $\forall s, t \in r : s[X] = t[X] \Rightarrow s[Y] = t[Y]$. For example, the attributes `zip, street_name, latitude` usually functionally determine `city` in an `address` table.

Order dependencies (ODs). An order dependency $\boldsymbol{X} \mapsto \boldsymbol{Y}$ denotes that sorting the tuples of a table by $\boldsymbol{X}$ also orders the records by $\boldsymbol{Y}$. In formal terms, for two lists of attributes $\boldsymbol{X}$ and $\boldsymbol{Y}$ of a relation R, the *order dependency* (OD) $\boldsymbol{X} \mapsto \boldsymbol{Y}$ holds in relation instance r over R, iff $\forall s, t \in r : s[\boldsymbol{X}] \preceq t[\boldsymbol{X}] \Rightarrow s[\boldsymbol{Y}] \preceq t[\boldsymbol{Y}]$. Note that the comparison operator $\preceq$ compares the $\boldsymbol{X}$ and $\boldsymbol{Y}$ values attribute-wise, i.e., lexicographically via $\leq$ with the first attribute in each list being the most significant one. An example OD from an `employees` table might be `salary` $\mapsto$ `taxrate`.

2.2 Dependency-based query optimizations

Since the creation of relational theory, the database community has developed a plethora of query optimization techniques based on various data dependencies. A recent scientific study [11] described and categorized 59 optimizations, which are widely underutilized in practice. Hence, in this study, we exemplarily selected and implemented three of these techniques; Section 4 shows their potential for query optimization:

O-1 Join to semijoin [16]. A UCC on a joined column allows simplified join executions. Consider the following query:

```
SELECT r.A, r.B FROM r, s WHERE r.ID = s.ID
```

If `ID` is a UCC on `s`, above's query can be executed by more efficient semijoin strategies. This optimization works because the UCC guarantees that for any tuple of `r`, there will only be a single matching tuple from `s`.

O-2 Reduce `GROUP BY` attributes [3, 5]. Functional dependencies can be used to execute grouping operations more efficiently by reducing the number of grouping attributes. Consider the statement `GROUP BY X, A` and the FD $X \longrightarrow A$. By the definition of FDs, tuples that agree on `X` will also agree on `A`. Hence, during grouping, such tuples will fall into the same groups so that calculating `GROUP BY X` is equivalent.

O-3 Join avoidance [26]. In some cases, joins can be avoided with the knowledge of order dependencies or UCCs. Consider the following simplified query excerpt and an OD $\texttt{date}_{\texttt{sk}} \overset{\leq}{\mapsto} \texttt{date}$:

```
SELECT ... FROM fact, dim WHERE
  fact.date_sk = dim.date_sk AND
  dim.date BETWEEN '20210816' AND '20210820'
```

The join between fact and dimension tables can be replaced with a local predicate if no columns of the dimension table are needed for the final result: `fact.date_sk BETWEEN min_date_sk AND max_date_sk` where `min_date_sk` (and `max_date_sk` accordingly) are computed as:

```
MIN(date_sk) min_date_sk FROM dim WHERE
  date >= '20210816'.
```

Dimension tables are often small compared to fact tables [9]. Thus, `min_date_sk` and `max_date_sk` can be determined quickly. The resulting `BETWEEN` predicate is usually more efficient than the original join. Without the aforementioned OD, it would not be guaranteed that, e.g., `MIN(date_sk)`, determines the correct value.

Similarly to the OD-based optimization, joins can be converted to cheap local predicates if the dimension table is filtered on a UCC. Instead of a `BETWEEN` predicate comparing for `MIN` and `MAX` of the join columns, a simple equals predicate is sufficient. The application of these OD- and UCC-based join avoidance optimizations, and in particular their knowledge during the query optimization phase, permits another downstream optimization: Introducing a local predicate on the fact table enables pruning opportunities that often show significant performance impacts in practice. These impacts are included in the reported performance numbers for O-3.

2.3 Hyrise

To demonstrate our dependency-based query optimization system, we implemented [10] it into the research DBMS Hyrise[1]. In the following, we briefly introduce the architecture and relevant components of Hyrise [6] since some of its concepts are beneficial to our approach.

Hyrise is a main memory, column-oriented database system with an implicitly horizontally partitioned storage layout. The partitions are called *chunks* and have a fixed maximum size (default: 65 535 tuples). By default, dictionary encoding is applied to all chunks. Furthermore, Hyrise follows an insert-only approach utilizing multiversion concurrency control (MVCC) where deletes and updates do not physically delete rows. Instead, rows are marked invisible and new versions are appended to the table's last chunk. In addition, Hyrise offers a so-called *plugin interface* that enables the implementation of autonomous components with access to internal resources. Such components can read and write internal data structures without being tightly coupled to the database's core (cf. [6, p. 321]), which simplifies the implementation of our approach.

3 AUTONOMOUS DISCOVERY, SELECTION, AND MUTATION OF DATA DEPENDENCIES

In this section, we first explain our workload-driven, lazy data dependency discovery system that determines and validates data dependency candidates for specific query optimization objectives, considering only such dependencies that are actually relevant for the observed workload (Section 3.1). We then explain how changes to the data, which potentially invalidate dependencies, can be handled (Section 3.2). While we implemented our data dependency-based query optimization system for Hyrise, the presented concepts are based on generic database concepts and not technically coupled to this particular database system; hence, they should be applicable to other database systems as well.

3.1 Workload-driven, lazy data dependency discovery and selection

Data dependencies are usually not known, and determining all dependencies on a table or dataset without limiting the search space is extremely expensive. The overall goal of our approach is to efficiently provide data dependencies that can be used for query optimization. For this reason, and in contrast to state-of-the-art data profiling algorithms, we approach this problem by not considering all possible attribute combinations as potential data dependency candidates but only specifically promising ones.

For this purpose, our approach consists of two phases: First, we determine which data dependency candidates are *relevant*, i.e., which dependencies could be beneficial for processing the system's workload when applied during query optimization. Second, the previously determined candidates are validated.

The concrete procedure that determines and validates relevant dependency candidates is depicted in Figure 1. The figure contains two main components: (i) A database system (Hyrise in our case) that is capable of applying dependency-based optimizations and (ii) the *dependency discovery plugin.*

Database System. ① All queries are passed as SQL strings to the database system for processing. ② The SQL strings are then translated to logical query plans, optimized, and finally translated to physical query plans. If a query was already processed in the past, its plan might be retrievable from the plan cache (2A) to avoid unnecessary retranslation and -optimization. Also, during optimization, dependencies might be retrieved from the *data dependency store* (2B) to create more efficient query plans – this extension enables the application of data dependency-based optimizations, such as the three optimizations discussed in Section 2.2. ③ Afterward, the query plan gets executed and is stored with runtime and cardinality information in the query plan cache.

Dependency Discovery. The actual dependency discovery extension, which is implemented as a Hyrise plugin [10] (cf. Section 2.3), is executed periodically with a configurable frequency. ④ The basis for this procedure is a set of user-defined *rules* that provide the logic to derive specific dependency candidates from queries in the plan cache, from executed physical and their corresponding logical operators. These rules subscribe to certain operator types, e.g., scans or joins, and define under what circumstances which dependency candidates are created. For instance, for O-3: *if two relations are joined, one relation does not contribute attributes to the final result, and this relation is filtered, then create an OD candidate* `join_column` $\mapsto$ `filter_column`. In other words, the rules determine which dependencies need to exist to enable certain optimizations (O-1, O-2, O-*n* in Figure 1). Note, it depends on the user-defined rules whether or not the dependency selection promotes useful dependencies.

⑤ The dependency discovery procedure accesses the database system's plan cache that serves as a representation of the processed workload. The *PlanParser* handles the cache's entries, i.e., the physical and logical query plans, in an operator by operator fashion. ⑥ The operators are passed to all *rules* that subscribed to the particular operator type (O-2 and O-*n* in Figure 1). ⑦ Afterward, the rules, which are active components in this architecture, check the user-defined requirements and return applicable, operator-specific dependency *candidates* (e.g., OD and UCC candidates in Figure 1).

⑧ Next, the dependency candidates are validated against the underlying data by the *dependency validator*. In our case, where the procedure is implemented into Hyrise, the validation process profits from memory-resident data, its column-oriented storage layout and dictionary encoding, see Section 3.2 for more details. For some dependency types, the dependency validator partly resorts to Hyrise's efficient operator implementations, for instance, the *Sort* operator for validating order dependencies. We also use techniques of existing validation algorithms, such as sampling [19], to quickly eliminate candidates. In a more generic setting, traditional SQL- or PLI-based validation algorithms could be used [1].

(8A) After validation, the available data dependencies are stored in the database system's *data dependency store* to be used during

[1]Hyrise source code and documentation: https://github.com/hyrise/hyrise

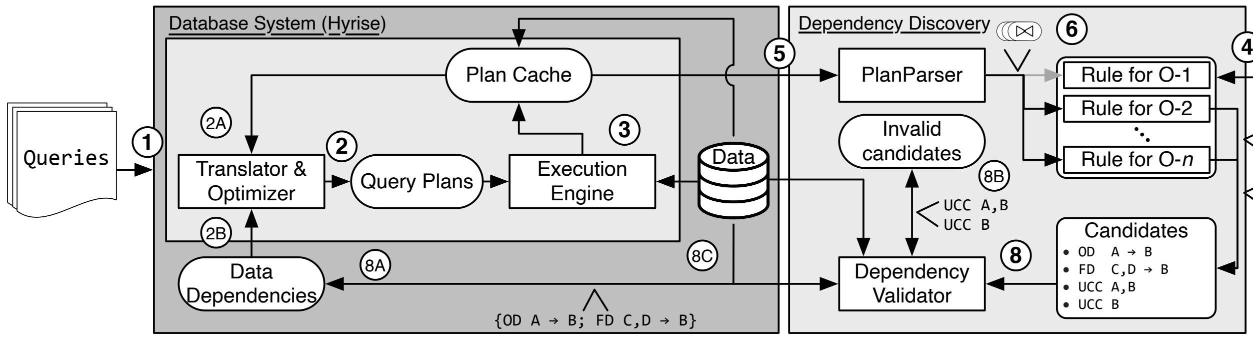

Figure 1: Schematic overview of the dependency discovery and selection procedure and its DBMS interaction.

query optimization. (8B) In addition, the system also keeps a list of unsuccessfully validated dependency candidates for efficiency reasons. Both stores are accessed during validation to avoid repeated unnecessary validations on unchanged underlying data. (8C) Furthermore, the corresponding query plans are removed from the plan cache to enable re-optimization, and, thereby, the use of the validated dependencies for optimization.

3.2 Efficient data dependency mutation

Data changes can invalidate dependencies, for instance, by introducing duplicates in certain columns. Then, the use of invalid dependencies for query optimization can lead to poor cost and cardinality estimations, inefficient query plans and, in the worst case, faulty results. While such mistakes might be acceptable in certain cases, such as approximate query processing [14, 18], they are, in general, not tolerable.

In this section, we go over three techniques that keep the dependencies up-to-date: (i) workload-driven discovery, (ii) incremental validation and maintenance, and (iii) the utilization of column store DBMS concepts. We now discuss each technique in more detail.

Workload-driven discovery. A general observation about data dependencies is that *spurious* dependencies may change very often, while *genuine* dependencies, i.e., semantically meaningful dependencies that model a real-world constraint, do not or only rarely change over the lifetime of a dataset. Because our system draws the dependencies from real-world, executed, and possibly human-crafted, SQL queries and datasets, a large portion of the discovered dependencies is expected to be genuine. The workload-driven discovery, hence, works as a semantic pre-filter, which significantly reduces the amount of change in the dependency store when compared to other mined dependency collections. Adding to this observation, our evaluations (cf. Section 4) have also revealed that many beneficial dependencies exist on dimension tables, e.g., date or type tables, that are very rarely updated [9, p. 141]; and if updates are actually required, the cost of validating dependencies is low due to the comparably small size of dimension tables [9], as we observe in Section 4.3.

Incremental validation and maintenance. If our dependency-based query optimization system is applied in some data warehouse scenarios, where the data is updated in specific, low-frequent cycles [25], keeping the dependencies up-to-date in batch jobs is easy. For the more general setup, though, we need to focus on more fine-grained updates: Given a concrete data change, it is not necessary to re-evaluate all dependencies, and it is also not necessary to re-evaluate dependencies entirely. Instead, we propose the use of efficient incremental validation and maintenance approaches that exist for FDs [4, 23] and ODs [27]. For UCC maintenance, we can adopt techniques that were originally designed for enforcing key constraints [15]. These approaches use clever techniques and index structures to selectively re-evaluate only affected cases. Because our system maintains only a very small subset of dependencies and does not progressively replace invalidated dependencies by new minimal dependencies (as the referenced incremental discovery systems do), the maintenance is much more efficient.

Utilization of column-store DBMS concepts. Apart from the previously discussed incremental validation techniques, certain column-store concepts are particularly beneficial for handling data change in terms of efficient dependency validation and the treatment of potentially invalid ones. Modern column-store systems, such as DuckDB [22], HyPer [8], or Hyrise [6], often store values in *chunks*, which are implicit horizontal partitions of a fixed size that are compressed and made *immutable* when their capacity is reached [12]. `UPDATE` statements are implemented as *append-only* operations, where the invalidation of an original tuple is succeeded by the insertion of a new tuple containing the updated values. This layout can be exploited in at least two ways: First, similar to some database operations, some dependency validation algorithms benefit directly from the columnar storage layout. Because the relational data dependencies express relationships between attributes, i.e., columns, typically, only very few columns need to be read for their validation. Since in columnar storage layouts relevant data can be accessed precisely, column stores, such as Hyrise, serve to (re-)validate the dependencies more efficiently. A second advantage of column stores is the fact that such systems often rely on compression techniques, such as dictionary encoding, which can further improve the validation, especially for incremental tests. For example, our implementation uses Hyrise's chunk-wise dictionaries to quickly determine non-unique column combinations during the validation of UCC candidates.

In summary, the challenges caused by data changes can largely be solved with our workload-driven architecture, recent incremental maintenance techniques, and concepts of modern column stores.

4 EVALUATION

To evaluate our workload-driven dependency discovery system for query optimization as well as the dependency-based query optimization techniques themselves, we implemented the system as a Hyrise plugin [10] that analyzes Hyrise's plan cache to determine and validate useful dependency candidates. We also integrated the optimization O-3 into Hyrise, while O-1 and O-2 had already been integrated in former releases.

4.1 Experimental setup

All the following experiments were conducted on an Intel Xeon 8180 Platinum CPU with 384 GB of main memory. For our evaluation, we use the TPC-H [21], TPC-DS[2] [17], and Join Order Benchmarks (JOB) [13]. While the first two use synthetic datasets and were specifically designed for benchmarking analytical systems, the latter operates on real-world data from the IMDB. Such real-world data is particularly interesting to evaluate the effects of dependency-driven optimizations in practice. For the TPC-H and TPC-DS, we use a scale factor of 10 and, for the JOB, we use the original paper's dataset[3].

To simulate a scenario as depicted in Section 1, where the system has to identify all dependency candidates by itself, no (foreign) keys or constraints are defined for the benchmarks. For the evaluation of a single benchmark, we first measure the baseline execution time as the average (mean) execution time of its entire workload over 100 executions. Then, we invoke the dependency discovery plugin, which in practice would run in the background during query processing. Afterwards, the same workloads are executed and measured again. In this way, the experiment measures the execution times of the plugin and the fully optimized workloads separately. All experiments are executed in a single threaded fashion.

4.2 Optimization performance

Table 1 shows the performance impact (*execution time*) of three data dependency-driven query optimization techniques and information of the entire dependency selection and discovery process (*candidates*). Apart from the impact on the execution time, Table 1 also shows how many queries showed improved or degraded performance. For the selection and discovery process, the number of identified and valid candidates as well as the total runtime of the dependency discovery plugin is depicted.

As a first observation, the realized benefits are substantial, which is particularly true for JOB and TPC-DS, where a combination of all three optimizations reduces the execution times by 27 % and 10 % respectively. While the observed performance benefits do not represent a formal verification of our approach, they indicate that the proposed selection process promotes useful dependencies. Also, for these two benchmarks the performance of the vast majority of all queries is affected positively; more than half of JOB's 113 queries are improved. In all cases, the number of improved queries clearly outweighs the degraded ones.

In addition to the summary statistics contained in the table, we want to point out that each of the optimizations causes significant improvements in some queries. For example, for O-1, we observed a 2.6 × speed-up for JOB's Q22d; for O-2, we observed a 1.6 × speed-up[4] for TPC-H's Q10; for O-3, we observed a 65 × speed-up of TPC-DS' Q32. Furthermore, the table demonstrates that the optimizations compete with each other in some cases. For instance, considering the individual impacts of O-1 and O-3, one could expect to see a larger combined improvement for the JOB, which is not the case because both optimizations target joins.

Discussion. The magnitude of the impact that dependency-based query optimization techniques have on the performance depends on the data model, the queries, the evaluated optimizations, and the underlying data. For example, O-1 and O-3 explicitly target joins, which occur frequently in the JOB and have a larger overall impact in the JOB and TPC-DS than aggregates, which are targeted by O-2. Also, the star schema-like data model of the TPC-DS and JOB is more suitable for such optimizations. Finally, the data itself determines whether dependencies exist: The ratio of valid and candidate dependencies is high for the real-world data based JOB (85 %), compared to the TPC-H (35 %) and -DS (19 %).

4.3 Discovery and selection overhead

Using dependencies for query optimization introduces a tradeoff between the realized performance improvements and the overhead for determining dependencies. Therefore, while the overhead introduced by the initial dependency discovery and selection process is visible in some cases, it must be judged under consideration of the achieved performance gains. Table 1's *Time* columns indicate the time necessary to determine dependency candidates and their validation.

For the scenario where all three optimizations are applied, the system searches for UCCs, FDs, and ODs. Note, the times of O-1 to O-3 do not necessarily result in the *Combined* time as some dependency candidates might be relevant for multiple optimizations. The necessary time and break-even rate are different for the three investigated benchmarks: For the JOB and TPC-DS not even an entire run of all queries is necessary to break even. For the TPC-H, all queries must be executed more than once.

There are multiple reasons for the differences in validation runtime. First, some types of dependency candidates are harder to validate or rule out than others. For example, determining or ruling out uniqueness is generally simpler than sorting large tables with dozens of millions of rows. The overhead of the candidate generation, i.e., query plan analysis and candidate extraction, is significantly lower compared to the candidate validation. For the presented experiments, the candidate generation consumed never more than 10 % of the total dependency discovery time. Lastly, the table's size and the nature of the data impact validation costs. In fact, validating a single UCC on `c_address` of TPC-H's `customer` table with $\approx$ 1.5m rows takes one second and is responsible for 36 % of the discovery and selection runtime. In contrast, the TPC-DS profits from an OD on a small dimension table that can be validated in only 41ms. The last observation conforms with the argumentation presented in Section 3.2: There are valuable data dependencies leading to performance improvements that can be (re-)validated quickly.

[2]Currently, Hyrise supports only 47 of the 99 TPC-DS queries.
[3]Dataset for Join Order Benchmark: http://homepages.cwi.nl/~boncz/job/imdb.tgz

[4]For the JCC-H [2] dataset the observed speedup was $\approx$2.5 ×.

Table 1: Performance (execution time) impact of optimizations (O-1 to O-3, see Section 2.2). ∑ combined execution time of all queries. ∅ indicates the mean execution time change across all of the workload's queries. #↓ indicates the number of improved queries (with lower execution time), #↑ degraded ones; only changes larger/smaller than or equal to +/-5% were considered for this metric. In addition, the number of dependency candidates, successfully validated dependencies and the combined time for candidate generation and validation is displayed.

	JOB (113 Queries)							TPC-DS (47 Queries)							TPC-H (22 Queries)						
	Execution time				Candidates			Execution time				Candidates			Execution time				Candidates		
	∑	∅ Change	#↓	#↑	#	Valid	Time	∑	∅ Change	#↓	#↑	#	Valid	Time	∑	∅ Change	#↓	#↑	#	Valid	Time
Baseline	40.7 s	–	–	–	–	–	–	29.8 s	–	–	–	–	–	–	39.5 s	–	–	–	–	–	–
O-1	-8.9 s	-22 %	44	24	10	10	474 ms	-1.5 s	-5 %	25	1	13	11	196 ms	-0.7 s	-2 %	3	2	5	5	1.4 s
O-2	0 s	0 %	0	0	0	0	8 ms	-0.7 s	-2 %	5	0	53	2	54 ms	-1.5 s	-4 %	1	0	19	5	2.7 s
O-3	-7.3 s	-18 %	70	7	18	15	97 ms	-1.1 s	-4 %	6	0	23	7	208 ms	-0.6 s	-2 %	1	0	12	7	1.4 s
Combined	-10.8 s	-27 %	73	15	20	17	498 ms	-3.0 s	-10 %	28	1	72	14	227 ms	-2.2 s	-6 %	4	2	26	9	2.8 s

Discussion. Above's experiments have shown that the effort for dependency discovery is usually amortized quickly in most scenarios. More importantly, the discovery process is not tied to query execution and optimization. Hence, it can be executed as asynchronous background task whose runtime is significantly less relevant. In addition, preliminary evaluations on larger TPC-H and TPC-DS datasets show that the performance benefits increase at least as quickly as the validation efforts.

5 RELATED WORK

In the following, we briefly discuss existing approaches that investigate how data dependencies can be used for query optimization. Some dependency-based query optimization techniques are used in commercial database systems. However, such systems do not autonomously determine the necessary dependencies. Thus, these optimizations are based on user-defined dependencies. Examples include UCCs and INDs (referential integrity constraints) to reduce the number of statistical views [7], ODs for join avoidance [26], or FDs (which are only validated if instructed by the user) for improved selectivity estimates [28].

Pena et al. [20] propose a system that automatically incorporates FDs for query rewriting. First, their approach needs to discover all functional dependencies. Then, their applicability is determined by comparing (i) attribute matrices representing the FDs and (ii) matrices of attribute occurrences in the workload's queries. The FDs used to optimize the queries by rewriting, are selected according to rankings by quality metrics or clustering. In contrast to our approach, their dependency discovery process is not limited to relevant candidates, leading to possibly high computational costs. Additionally, only query rewriting is performed. While that approach works without adjusting the database system (rewriting can solely work on SQL strings), not all optimizations can be achieved by rewriting and the full potential of plan level optimizations cannot be realized. Furthermore, our evaluation has shown that other dependency types than FDs show larger performance impacts.

6 CONCLUSION

We presented an approach that efficiently determines data dependencies based on concrete, given workloads and applies them during query optimization to generate more efficient query plans to improve performance. The three challenges when applying data dependencies for the purpose of query optimization are the *discovery*, *selection*, and *mutation* of relevant dependencies. We proposed an integrated solution that tackles these challenges with our workload-driven, lazy dependency discovery approach, incremental validation and maintenance techniques, and concepts of column store DBMSs.

An evaluation performed with the open-source DBMS Hyrise showed promising results for the analytical benchmarks TPC-H, TPC-DS, and JOB: The observed runtime improvements are substantial (up to 27 % of reduction in workload execution time and more than 60 × query speed up) and the overhead for determining and validating dependency candidates is reasonable: between 0.05 × and 1.3 × of the observed execution time reduction. Note, though, that the discovery efforts run as asynchronous background tasks and are, therefore, not a factor that needs to be considered to immediately impact query optimization costs. Furthermore, in the conducted experiments, we observed that those dependencies that turned out to be beneficial for query optimization could also be re-validated quickly, which mitigates the impact of dependency mutation.

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. 2018. *Data Profiling.* Morgan & Claypool Publishers.

[2] Peter A. Boncz, Angelos-Christos G. Anadiotis, and Steffen Kläbe. 2017. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *Performance Evaluation and Benchmarking for the Analytics Era - 9th TPC Technology Conference, TPCTC 2017, Munich, Germany, August 28, 2017, Revised Selected Papers.* 103–119.

[3] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC).* 61–76.

[4] Loredana Caruccio, Stefano Cirillo, Vincenzo Deufemia, and Giuseppe Polese. 2019. Incremental Discovery of Functional Dependencies with a Bit-vector Algorithm. In *Proceedings of the Italian Symposium on Advanced Database Systems.*

[5] C. J. Date and Hugh Darwen. 1992. *Relational Database Writings 1989-1991.* Addison-Wesley, Chapter The Role of functional Dependence in Query Decomposition, 133–150.

[6] 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 *Proceedings of the International Conference on Extending Database Technology (EDBT).* 313–324.

[7] IBM. 2019. *Referential integrity constraints help reduce the number of statistical views.* https://www.ibm.com/support/knowledgecenter/SSEPGG_11.5.0/com.ibm.db2.luw.admin.perf.doc/doc/c0059081.html

[8] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of*

the International Conference on Data Engineering (ICDE). 195–206.

[9] Ralph Kimball and Margy Ross. 2013. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling, 3rd Edition*. Wiley.

[10] Jan Kossmann, Daniel Lindner, Felix Naumann, and Thorsten Papenbrock. 2021. *Source Code for workload-driven, lazy discovery of data dependencies for query optimization*. https://git.io/WorkloadDrivenDependencyDiscoveryForQO

[11] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2021. Data Dependencies for Query Optimization: A Survey. *The VLDB Journal* (2021).

[12] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 311–326.

[13] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

[14] Qing Liu. 2018. Approximate Query Processing. In *Encyclopedia of Database Systems, Second Edition*. Springer.

[15] Microsoft. 2017. *SQL Server 2019 - Unique Constraints and Check Constraints*. https://docs.microsoft.com/en-us/sql/relational-databases/tables/unique-constraints-and-check-constraints?view=sql-server-ver15

[16] MySQL. 2021. *MySQL 8.0 Reference Manual - Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations*. https://dev.mysql.com/doc/refman/8.0/en/semijoins.html

[17] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the International Conference on Very Large Databases (VLDB)*. 1049–1058.

[18] Ullas Nambiar and Subbarao Kambhampati. 2004. Mining Approximate Functional Dependencies and Concept Similarities to Answer Imprecise Queries. In *Proceedings of the ACM Workshop on the Web and Databases (WebDB)*. 73–78.

[19] Thorsten Papenbrock and Felix Naumann. 2016. A Hybrid Approach to Functional Dependency Discovery. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 821–833.

[20] Eduardo H. M. Pena, Erik Falk, Jorge Augusto Meira, and Eduardo Cunha de Almeida. 2018. Mind Your Dependencies for Semantic Query Optimization. *Journal of Information and Data Management* 9, 1 (2018).

[21] Meikel Pöss and Chris Floyd. 2000. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Record* 29, 4 (2000), 64–71.

[22] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1981–1984.

[23] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Felix Naumann, Dennis Hempfing, Torben Mayer, and Daniel Neuschäfer-Rube. 2019. DynFD: Functional Dependency Discovery in Dynamic Datasets. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 253–264.

[24] Sebastian Schmidl and Thorsten Papenbrock. 2021. Efficient Distributed Discovery of Bidirectional Order Dependencies. *VLDB Journal* (2021).

[25] Il-Yeol Song. 2018. Data Warehousing Systems: Foundations and Architectures. In *Encyclopedia of Database Systems, Second Edition*. Springer.

[26] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, Wenbin Ma, Przemyslaw Pawluk, and Calisto Zuzarte. 2011. Queries on dates: fast yet not blind. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 497–502.

[27] Zijing Tan, Ai Ran, Shuai Ma, and Sheng Qin. 2020. Fast Incremental Discovery of Pointwise Order Dependencies. *PVLDB* 13, 10 (2020), 1669–1681.

[28] The PostgreSQL Global Development Group. 2021. *PostgreSQL: Documentation: 14.2.2.1.: Functional Dependencies*. https://www.postgresql.org/docs/current/planner-stats.html