Unleashing Data Dependency-based Query Optimization

Daniel Lindner
Hasso Plattner Institute
Potsdam, Germany
daniel.lindner@hpi.de

Daniel Ritter
SAP
Walldorf, Germany
daniel.ritter@sap.com

Felix Naumann
Hasso Plattner Institute
Potsdam, Germany
felix.naumann@hpi.de

Abstract

Primary key (PK) and foreign key (FK) constraints are widely used for query optimization. Knowledge about additional data dependencies, such as other unique columns or order dependencies, enables further substantial performance improvements. However, such dependencies are unknown or cannot even be specified by SQL. Thus, identifying and validating relevant dependencies automatically and efficiently remains an unsolved problem. This paper presents a system that (i) recognizes dependency candidates for optimization, (ii) efficiently checks their validity on the data, and (iii) optimizes query plans using valid dependencies.

While evaluating the performance impact of optimization techniques using data dependencies beyond PKs and FKs, we could empirically show that data dependencies improve performance for a wide range of analytical database systems and benchmarks. Thus, for the first time, we integrate data dependencies into a database system to use them without (i) manual declaration and maintenance or (ii) manual SQL rewrites. Our integrated and fully automated system matches the performance of dedicated SQL rewrites: compared to using only PKs and FKs, we achieve high geometric mean speedups, for instance 35 % for TPC-DS and 29 % for JOB. Individual query latencies improve by more than 90 %. The dependency discovery overhead is orders of magnitude lower than the latency improvement of a single workload execution.

Keywords

Data profiling, query optimization, data dependencies, subqueries

1 Introduction

Query optimization (QO) in database systems is crucial to find efficient execution plans and has been studied since the dawn of relational databases [35, 53]. Essential and well-known optimization techniques, such as predicate placement [70], join ordering [32], and subquery unnesting [37], substantially improve execution times.

A recent survey highlighted that optimizations using data dependencies have been proposed throughout the history of database research [40]. Data dependencies are "metadata that describe relationships among columns" [2, p. 561], and they formalize specific properties of datasets. Dependency discovery and use have been researched for decades [21] for different application areas, such as data cleaning [22, 69] and data integration [43, 48].

Among others, data dependencies include unique column combinations (UCCs), functional dependencies (FDs), inclusion dependencies (INDs), and order dependencies (ODs). For instance, a UCC states that tuples have no duplicate values in a given set of attributes, and an IND means that values for specific attributes are also present in another set of attributes, often in another table (see Section 3.1). In database systems, these two

EDBT '26, Tampere (Finland)

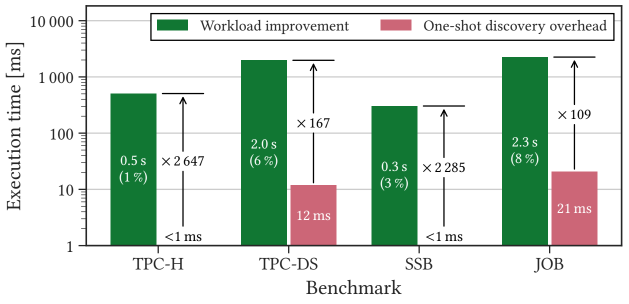

Figure 1: Workload improvement and discovery overhead when automatically discovering and exploiting data dependencies additionally to the schema (single-threaded). The improvement of a single workload execution is orders of magnitude higher than the overhead during data ingestion.

dependency types can be represented using unique/primary key (PK) and foreign key (FK) constraints, and database management systems (DBMSs) often apply optimizations if these constraints are present (see Section 7.2).

Many data dependencies are valid in real-world datasets due to relationships of real-world or modeled entities, application logic, or just by chance [9, 85]. Yet, DBMSs apply only a few dependency-based optimizations [40], mostly based on PKs and FKs. Previous research emphasized that valid dependencies are often unknown or not declared as constraints [8, 46, 85]. SQL cannot even express some relevant dependencies, such as FDs or ODs, as table constraints [34, p. 89]. Also, no column-wide constraints can be specified for standardized storage formats, such as CSV, Parquet, or ORC files, let alone constraints across multiple tables, e. g., FKs/INDs.

Thus, previous work outlined a system to automatically generate and validate interesting dependency candidates using workload information [39]. The evaluation showed to what extent selected optimizations using discovered dependencies benefit the performance of three benchmark workloads. However, that work provided only a proof of concept, and dependency-based optimizations were not fully integrated into the DBMS. Furthermore, the validation algorithms caused considerable overhead compared to the performance improvement per workload execution. Thus, that work raised two main questions to close the gap and make dependency-based QO practical:

Q-1 How can database systems efficiently validate data dependencies on stored data?
Q-2 How to thoroughly integrate data dependency-based optimization into a DBMS, e. g., by propagating valid dependencies along a plan to enable optimization decisions?

This paper addresses these questions by providing metadata-aware dependency validation algorithms and building blocks to integrate data dependencies as first-class citizens into a DBMS optimizer, further improving upon the architecture of [39]. We perceive dependencies as sole metadata and abolish the necessity to model, enforce, and maintain constraints explicitly (e. g., by creating indexes). We select three cost-independent dependency-based query rewrites for groupings and joins [40], which are expensive operations [18, 76], and apply them to different workloads and DBMSs.

Two rewrites rely on PKs and FKs, but Figure 1 shows that implementing optimizations using further dependencies notably improves the execution time of two benchmark workloads: 17 TPC-DS and 66 JOB queries improve with over 10× speedups for individual queries, and geometric mean speedups of 35 % and 29 %. Such improvements significantly benefit interactive analytical workloads with continuously recurring query patterns [4, 71, 82] comparable to other state-of-the-art optimization approaches [25, 36, 59, 64, 65]. While dependency-based QO techniques could also benefit transactional workloads, we focus on analytical scenarios: more complex queries exhibit more optimization potential.

Using our workload-driven architecture (Section 4), we discover relevant dependencies as part of an *extract, transform, load* (ETL) pipeline, re-iterating the discovery whenever data is ingested in batches. The discovery overhead is orders of magnitude smaller than the latency improvement of a single workload execution due to efficient validation algorithms optimized for databases (Q-1).

We integrate three rewrites from literature into the optimizer of an open-source DBMS as cost-independent transformations and evaluate them. To enable these optimizations, we propagate dependencies in the query plan. Furthermore, we modify the optimizer and the execution engine to handle predicates with subquery results, efficiently transforming joins into selections (Q-2). Adding optimization and candidate rules can extend our framework to further (also cost-based) techniques relying on UCCs, INDs, or ODs.

After discussing related work in Section 2, Section 3 introduces fundamental concepts of data dependencies, dependency-based query optimization, and database systems. Section 4 presents our architecture. In particular, we make the following contributions:

- C-1 *Metadata-aware dependency validation (Q-1).* We propose algorithms that exploit data layout, encoding, and statistics to drastically reduce the overhead of validating four dependency types (Section 5).
- C-2 *Dependency propagation (Q-2).* We enable the application of dependency-based QO techniques by representing and propagating dependencies in the query plan (Section 6.1).
- C-3 *Handling predicates with subquery results (Q-2).* One selected rewrite introduces predicates with values that are determined by subqueries. We handle these predicates in the DBMS, estimate their cardinality, and use them for dynamic pruning even though the concrete values are unknown during optimization (Section 6.2).
- C-4 *Evaluation.* We evaluate the potential of dependency-based optimization even beyond using PKs and FKs for five state-of-the-art systems. Furthermore, we analyze the benefits of system integration and discuss the overhead of additional dependency discovery (Section 7). Finally, we discuss which workloads benefit most.

We conclude and discuss how future work can extend our approach, e. g., with dependency maintenance and versioning to support frequently changing datasets, in Section 8. Our open-source implementation is available online.[1]

[1]See https://github.com/HPI-Information-Systems/dependency-based-qo.

2 Related Work

We identify two main research fields connected to metadata-based query optimization. First, data *constraint- and dependency-based query optimization techniques* have been proposed for various systems and applied to research prototypes. Second, there has been research on *semantic query optimization*, including systems for the automatic discovery of semantic constraints.

Constraints and dependencies for query optimization. Query optimization using data dependencies has been proposed since the 1970s. In a survey, Kossmann et al. [40] collected more than 60 such optimization techniques, grouping them by the type of exploited dependency, the affected operator of the relational algebra, and the optimization category. Section 3.2 presents three logical query rewrites [1, 13, 17, 58, 78] in detail. To discover the required dependencies, traditional data dependency mining algorithms aim to find all valid dependencies in a given dataset. Efficient algorithms have been proposed for different dependency types in single-node and distributed environments [12, 20, 60]. However, (i) finding all dependencies in a dataset is still expensive, and (ii) query optimization can exploit further metadata besides data dependencies. Based on the survey, our previous work [39] presented the automatic dependency discovery approach discussed in Section 1. Liu et al. [46] performed static code analysis to identify various constraints that applications guarantee, such as INDs, regular expressions for strings, or nullability. They used this information for query preprocessing, stating that constraints were not declared, optimizers did not perform their rewrites, and there were no specialized discovery tools [46, p. 1209–1210]. In contrast, we efficiently discover, maintain, and make use of data dependencies within the DBMS.

Semantic query optimization. Various publications focus on semantic query optimization (SQO) [15, 29, 30, 38, 51, 74, 86]. SQO aims to transform query plans into more efficient ones of the same semantics [29, 73]. As this definition applies to *any* query rewrite, a stricter formulation uses only semantic constraints [38, 72, 74, 86]. Semantic constraints restrict the domain of attribute values, often with a condition that makes them specific to a subset of tuples. An example of a semantic constraint is that every employee in a managing role is paid a bonus of at least \$1 000. Various systems exist to discover and use semantic constraints for cost-based query optimization. Some of these systems also use dependencies, such as FDs, only to derive transitive closures or for propagation, e. g., after joins. Yu and Sun [86] and Hsu and Knoblock [30] compared query results to derive valid constraints. Thus, they could only transform queries if relevant query reformulations were also part of the workload. To overcome this shortcoming, Shekhar et al. [72] and Pena et al. [63] derived constraints from the data first and used them for optimization in the second step. These systems also discover semantic constraints that cannot be used for query optimization, leading to avoidable overhead. Furthermore, they add another optimization layer on top of the DBMS. Siegel et al. [75] generated constraint candidates during optimization and validated them later, coupling constraint discovery tightly with query optimization.

In contrast, we present a system to integrate optimization techniques and the discovery of beneficial dependencies into database systems. Instead of semantic constraints, we exploit data dependencies, which are valid for all tuples. Furthermore, our approach to collecting relevant metadata is decoupled from the core query execution and uses specialized validation algorithms.

3 Data Dependencies for Query Optimization

This section describes the basic concepts we build upon in our work, similar to Kossmann et al. [39]. After we define different data dependency types in Section 3.1, Section 3.2 illustrates three dependency-based logical query rewrites using an example query. Finally, Section 3.3 introduces relevant features of relational DBMSs.

3.1 Data Dependencies

Data dependencies are dedicated metadata that describe how data is interrelated. Specific relationships are formalized to prove and compute whether a dataset fulfills a dependency's requirements, i. e., whether the dependency is *valid* or not. In the following, we define four types of data dependencies.

Unique column combination (UCC). Let R be a relation. The subset of attributes $X \subseteq R$ is a UCC iff there are no tuples whose projection on X is equal [47]. UCCs can occur by chance or stem from real-world identifiers or surrogate keys. Thus, they are also referred to as *candidate keys* [16]. Relational databases can enforce UCCs via unique or primary key constraints.

Functional dependency (FD). An FD $X \rightarrow Y$ is valid iff all tuples with the same values for $X \subseteq R$ also have the same values for $Y \subseteq R$ [16, 81]. In particular, the FD $X \rightarrow R \setminus X$ always holds if X is a UCC. Real-world relationships often cause FDs, e. g., `zip` $\rightarrow$ `city`.

Order dependency (OD). If ordering the tuples of R by $\mathbf{X}$ also orders the tuples by $\mathbf{Y}$, then $\mathbf{X} \mapsto \mathbf{Y}$ is a valid OD [77]. In this case, $\mathbf{X}$ and $\mathbf{Y}$ are lists of attributes in R, i. e., the attribute order is relevant. ODs often occur in data that includes a time component [77, 78].

Inclusion dependency (IND). The IND $\mathbf{X} \subseteq \mathbf{Y}$ is valid iff all distinct values of $R[\mathbf{X}]$ are also present in $S[\mathbf{Y}]$ [14]. As a special case, R and S might refer to the same relation. INDs often represent membership or ownership and can be enforced by FK constraints.

3.2 Data Dependency-based Query Rewrites

Corresponding to previous work [39], we picked three techniques from the survey [40] that (i) target expensive join and group-by operations in analytical workloads, (ii) rely on four well-known dependency types, requiring validation algorithms for them, and (iii) are logical rewrites that promise to always improve performance. To illustrate the rewrites, we use an example query inspired by TPC-DS data and dependencies that selects each customer's ID, name, and the sum spent on purchases for a specific period:

```
  SELECT c_sk, c_name, sum(s_sales_price)
    FROM date_dim
         INNER JOIN sales ON d_sk = s_sold_date
         INNER JOIN customer ON s_customer = c_sk
   WHERE d_date = '2000-01-01'
GROUP BY c_sk, c_name;
```

Figure 2 shows query plans resulting from the three query rewrites, where Figure 2a is the original query plan. Dependencies exploited by individual optimization techniques are highlighted.

O-1 Dependent group-by reduction [13, 17]. Grouping by multiple attributes can be avoided if an FD's determinant *and* dependent attributes are part of the group-by list. We remove all dependent attributes from the grouping set and select any value of dependent attributes, as they are uniform within the group. In the example query, the customer's name `c_name` is unique for their ID `c_sk`. Thus, we only group by `c_sk`.

O-2 Join-to-semi-join rewrite [58]. The second query rewrite transforms an inner equi-join $R \bowtie S$ to a semi-join $R \ltimes S$. Many DBMSs implement semi-joins as they execute them efficiently [3, 6, 27, 49, 58]. This rewrite is possible if S's join key is unique and subsequent operators or the final projection require no further attributes of S. In fact, the semi-join acts as a filter for R by the values of S's join key(s). Figure 2c shows that we perform a semi-join to replace *sales* $\bowtie$ *date_dim*. Here, *date_dim*'s join key is the primary key, and no attribute of *date_dim* is selected later.

O-3 Join-to-predicate rewrite [1, 78]. If joins are merely used to filter relations, we might even replace them with a selection. In our example query, the *date_dim* table represents each day. Thus, the `d_date` column is unique. Selecting a single day results in a single value for the join key. Instead of joining *sales* with *date_dim*, the query plan of Figure 2d turns the join into a selection to filter *sales* for the single join key, which is determined by a scalar subquery.

Similarly, the rewrite can be applied to range predicates. Adapting our example query, we change the temporal filter from `d_date = '2000-01-01'` to `d_year = 2000`. The OD `d_sk` $\mapsto$ `d_date` ensures that the minimal and maximal join keys within the selected `d_date` values are fed into the join, and the combination of the IND `s_sold_date` $\subseteq$ `d_sk` and the UCC `d_sk` guarantees that all tuples of *sales* have exactly one join partner. Thus, we can rewrite the join to a selection with a predicate value between the minimum and maximum of the join key in Figure 2e.

3.3 Relevant Database Concepts

Dependency-based query optimization is applicable to any DBMS. However, columnar, partitioned, and encoded data with statistics is the basis of our novel dependency validation algorithms and dynamic partition pruning. We integrate the dependency discovery system as a plug-in decoupled from the DBMS core. Thus, we explain these DBMS concepts in the following paragraphs.

Storage layout. Many commercial, open-source, and research DBMSs support columnar storage [19, 23, 33, 41, 46, 52, 66, 67] to improve performance for analytical workloads [1, 11]. Standardized storage formats, such as Apache Parquet and ORC, also build upon this layout [87]. Columns are usually split into horizontal *partitions* (also called *chunks* or *row groups)* to enable parallelization and distribution of large data. Each partition contains one *segment* for each column in the table, storing a fraction of the attribute's fields. Immutable segments can be encoded to improve space and execution efficiency using light- or heavy-weight compression. Dictionary encoding is often the default for real-world data [24, 45, 87]: the (often sorted) dictionary stores all unique values locally for each segment or globally for the entire column [11], and the attribute vector references the dictionary offset for each tuple.

Statistics. Databases use statistics to refine access to partitions. Segments' minimal and maximal values (*zone maps* [88]) or value ranges (*range sets* [59]) enable *partition pruning*, i. e., skipping partitions if they cannot match selection predicates. Pruning is effective if data is partitioned by attributes that are frequently filtered, where tuples within the same value range are stored in the same partition. Statistics are also available for ORC and Parquet files [45, 87].

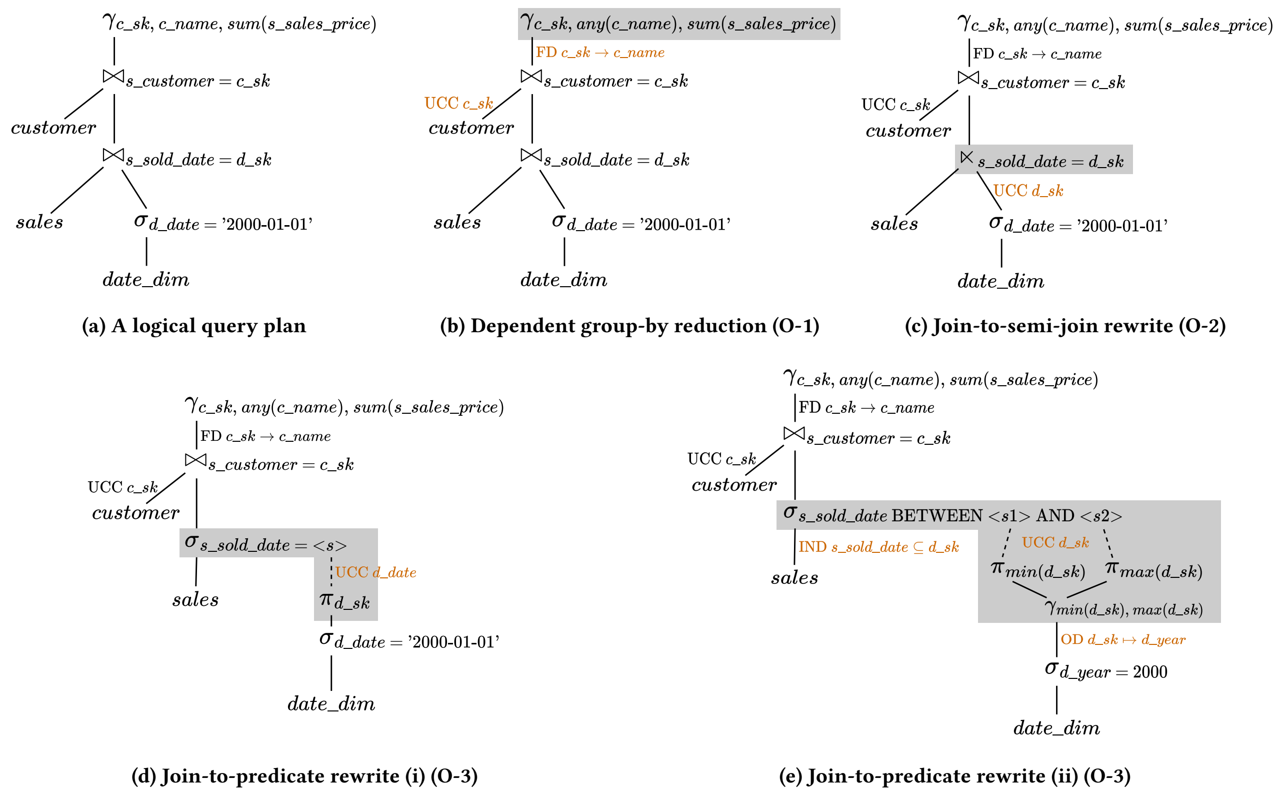

Figure 2: A query plan and versions successively rewritten for O-1, O-2, and O-3. Edges are annotated with the data dependencies that enable the specific rewrites. Rewritten parts and dependencies used are highlighted. Note that the selection predicate on *date_dim* was changed to showcase the OD-based version of O-3 in Figure 2e.

Plug-in interface. Plug-in interfaces allow functionality to be added without changing the core database code [5, 19]. Plug-ins are shared libraries that can be dynamically loaded and unloaded.

4 Workload-driven Data Dependency Discovery

This section describes the general approach of workload-driven dependency discovery, including details of our implementation for an open-source DBMS. The discovery system's architecture is inspired by previous work [39], but we (i) replace the dependency validation, (ii) integrate all three optimization techniques from Section 3.2 in the optimizer, (iii) extend the optimizer by dependency propagation and estimations for predicates with subquery results, and (iv) adapt the execution engine to enable dynamic pruning. Our ultimate goal is an automated system that exhibits the full potential of dependency-based QO with minimal discovery overhead.

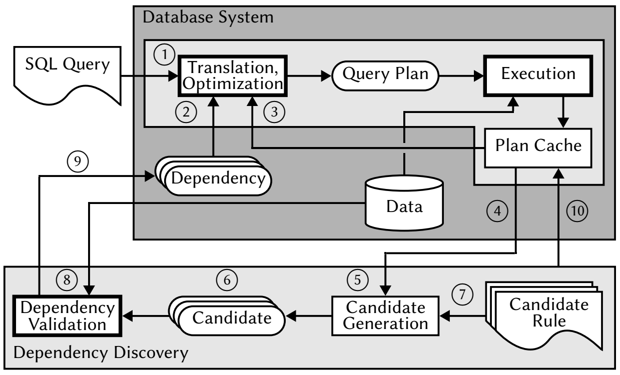

Figure 3: Automatic dependency discovery architecture, based on [39]. We replace crucial components (bold outlines).

4.1 Discovery Framework

Figure 3 gives an architectural overview of our automatic dependency discovery system. During regular workload execution, the DBMS translates a SQL query into a query plan and optimizes it ①. Optimizer rules can use metadata, such as data dependencies ②, in the optimization step. If the same query has been issued before, the query plan is obtained from the plan cache ③.

Automatic dependency discovery is triggered during an ETL process. Our discovery system obtains the workload's collected query plans from the plan cache ④, parses these plans ⑤, and obtains a set of dependency candidates ⑥. These candidates are determined by an extensible set of candidate rules ⑦. Candidate rules anticipate the behavior of dependency-based optimizer rules by searching for operator patterns that could be rewritten and suggest dependencies that are required to perform these rewrites. For instance, the join key of relation R fed into an inner equi-join is a UCC candidate if R's attributes are not used later. This UCC enables O-2 (join-to-semi-join rewrite).

Candidates are validated ⑧ on the stored data instance, skipping already validated candidates. We describe the tailored dependency validation algorithms and the order of candidate validation in Section 5. Valid dependencies are persisted ⑨ as table metadata. After execution, the plug-in invalidates the SQL plan cache entries of affected queries ⑩. Thus, future queries are optimized again, this time with optimization techniques using the persisted dependencies ②. Each new optimization rule requires only a new candidate generation rule to seamlessly integrate into the system. A meta table allows querying the discovered dependencies.

4.2 Implementation

We implemented three rewrites and dependency discovery for a modifiable DBMS with relevant features from Section 3.3, allowing reproducibility. We chose *Hyrise* [19] because it is an analytical open-source system designed for processing high-load workloads and serving multiple concurrent users, which is a common requirement of real-world interactive applications [4, 71, 82]. Hyrise is a columnar in-memory DBMS with horizontal partitioning into fixed-sized chunks of 65 535 tuples, featuring different encodings (the default is dictionary encoding), zone maps or range sets per column segment, a plug-in interface, and a rule-based optimizer with an extensible set of both heuristic and cost-based transformations.

We contribute multiple components based on the architecture of previous work [39]. First, we facilitate using discovered data dependencies in the core DBMS. Section 6.1 explains how we propagate data dependencies in query plans to derive valid dependencies for each operator in the optimization phase. Second, Section 6.2 provides the adaptations required to support subqueries introduced by dependency-based rewrites. These adaptations include improved cardinality estimation and extensions to query scheduling and execution to enable dynamic partition pruning. Third, we present our highly optimized dependency validation algorithms in Section 5.

The dependency discovery's design as a plug-in decouples it from the DBMS core, making it completely optional if all dependencies are known in advance. Triggering the discovery process is controllable and asynchronous, avoiding overhead in the running system. Thus, dependency discovery can be performed as a one-shot overhead for static datasets, regularly as part of an ETL process, or continuously for evolving workloads, where different query templates are queried over time. We perceive (discovered) data dependencies as additional metadata useful for optimization rather than as (SQL) data constraints. Because dependencies, as opposed to constraints, are not enforced by the DBMS, we avoid the overhead of, e. g., additional index structures and data checks regarding memory consumption and insertion latency.

5 Metadata-aware Data Dependency Validation

Reducing the overhead of additional dependency discovery requires efficient dependency validation strategies. This section presents tailored dependency validation algorithms we designate as *metadata-aware validation* (C-1). First, we motivate the need for fast dependency validation for query optimization and explain how it differs from traditional data profiling. Subsequently, we provide details on how we tailored algorithms to validate four types of data dependencies specifically inside a database. Finally, we explain how we order dependency candidates to minimize validation overhead.

We can validate specific dependency types using SQL [2, 9]. For instance, the following query validates the UCC *R*.a:

```
SELECT count(DISTINCT a) = count(a) FROM R;
```

However, specialized validation algorithms outperform such validation using general-purpose database operators [2, 20]. Contrary to state-of-the-art data profiling algorithms, we do not need to discover and validate *all* dependencies of a particular type, which is an NP-hard problem [2]. Rather, we focus on the efficient validation of individual dependency candidates instead of optimizing the traversal of the possible dependencies' search space (lattice) by aggressive pruning and data structures to combine already computed results. Furthermore, we can exploit metadata and encoding characteristics provided by the database system. Though we implemented our approach for an in-memory database system mainly relying on dictionary encoding, our algorithms apply to any system that uses common, accurate statistics for (horizontal partitions of) columns. These statistics are standardized by storage open formats (see Section 3.3), can be refreshed during the ETL processes, and remain unchanged for immutable partitions of append-optimized systems or data stored in open storage formats. As the applied query rewrites mostly target joins, our tailored validation algorithms provide specializations for numeric key candidates.

In the following subsections, R denotes a relation, a an attribute of R, and $\mathcal{S}_\mathsf{a}$ the set of a's segments, i. e., partitions of a. We denote the minimum and maximum attribute value present in a segment $s \in \mathcal{S}_a$ with $\min(s)$ and $\max(s)$. The cardinality (number of distinct values) of s is $\mathrm{dist}(s)$, whereas the number of tuples in s is $\mathrm{size}(s)$. The notions of cardinality and size also apply to attributes and relations. $|\mathcal{S}_\mathsf{a}|$ is the number of column a's segments.

5.1 Unique Column Combinations

State-of-the-art UCC discovery algorithms intersect so-called position list indexes (PLIs) [31] to combine the values of multiple columns and traverse the lattice efficiently [12, 61]. However, we can simplify the validation to construct a hash set containing a column's values for a single unary UCC candidate. As soon as we add a value to this set without increasing the set size, the column is not unique, and we can invalidate the candidate. If we added all fields of a column without aborting, the column is a UCC.

We further optimize the validation logic by incorporating metadata known by the database: a segment's minimal and maximal value, size, and cardinality. By accessing the first and the last elements of segments s's local dictionary, we can obtain $\min(s)$ and $\max(s)$. The dictionary size is $\mathrm{dist}(s)$, and the length of the attribute vector is $\mathrm{size}(s)$. If the segment is not dictionary-encoded or the dictionary is not sorted, zone maps, range sets, or other statistics provide this information even for open storage formats.

For metadata-aware UCC validation, we iterate over the dictionaries or data statistics. Figure 4 shows a running example of our UCC validation approach. If a single segment is not unique, neither is the column. Thus, we compare the distinct value count of each segment with its size, i. e., the number of stored tuples. We can immediately terminate the validation and reject the candidate if $\mathrm{dist}(s) \neq \mathrm{size}(s)$. This is the case for Segment 17: it has six unique values, where the segment size is seven.

Furthermore, a column is unique if all segments are unique and the segment values do not overlap. Thus, we build a segment index containing chunk IDs on the fly. We continuously insert each segment's chunk ID with both $\min(s)$ and $\max(s)$ as keys into a tree-based index. Thus, we can iterate and access the entries in a sorted fashion. For each chunk ID at key $\min(s)$, the chunk ID at the following key must reference the same segment,

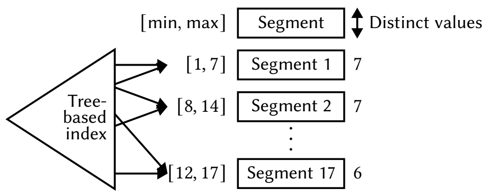

Figure 4: Metadata-aware UCC validation using the on-the-fly segment index. Segment 17 invalidates the UCC: it is not unique and its domain overlaps with Segment 2.

i. e., the same ID. Segments 1 and 2 in Figure 4 have no overlapping domains, but the minimal value of Segment 17 is between Segment 2's minimal and maximal values. Thus, the chunk ID at key 8 differs from that at the subsequent key 12.

However, the column can still be unique if all segments are unique, but their domains overlap. Then, we must fall back to constructing the hash set of all values. However, segments of range-partitioned (especially sorted) primary keys do not overlap by design, speeding up their validation (see Section 7.4). For such a range-partitioned column a of relation R, checking each segment's uniqueness and building the index has a complexity in $\mathcal{O}(|\mathcal{S}_\mathsf{a}| \cdot \log |\mathcal{S}_\mathsf{a}|)$ rather than $\mathcal{O}(\text{size}(R))$ for hash set construction,[2] where $|\mathcal{S}_\mathsf{a}| \ll \text{size}(R)$.

5.2 Functional Dependencies

Our approach to validate FD candidates uses a simplified strategy exploiting that $\mathsf{a} \rightarrow R \setminus \mathsf{a}$ is a valid FD if a is a UCC, e. g., if a is a primary key. Instead of searching for FDs in all combinations of the candidate columns (i. e., the *lattice*), we only check if one of the columns is unique. This simplification comes with the downside of falsely rejecting valid n-ary FD candidates with more than one determinant column. Indeed, we miss query optimization opportunities with these candidates, but we avoid the expensive lattice traversal. Also, the anticipated query rewrite benefits most when we can reduce to a single grouping attribute.

5.3 Order Dependencies

We cannot avoid sorting when validating an OD candidate $\mathsf{a} \mapsto \mathsf{b}$. The basic approach is to sort by a using the sort operator and to verify whether b is also sorted. If this is not the case, we reject the candidate. To reject invalid ODs faster, we first sort and check on a small sample. A sample size of 100 tuples is sufficient to reject all invalid ODs in our benchmark data (Section 7.4).

For tables with multiple chunks/partitions, we construct one segment index for each a and b. If we iterate both indexes simultaneously and the chunk IDs have the same order, we can sort each chunk individually and only fall back to sorting the entire column if there are overlaps. For segments of b, overlaps of one value, i. e., $\max(s_i) = \min(s_j)$, are allowed. In this way, we can reduce the complexity from $\mathcal{O}(\text{size}(R) \cdot \log \text{size}(R))$ to $\mathcal{O}(|\mathcal{S}_\mathsf{a}| \cdot \log |\mathcal{S}_\mathsf{a}| + |\mathcal{S}_\mathsf{a}| \cdot c \cdot \log c) \approx \mathcal{O}(|\mathcal{S}_\mathsf{a}| \cdot \log |\mathcal{S}_\mathsf{a}| + \text{size}(R) \cdot \log c)$, where c is the chunk size (e. g., fixed size of 65 535 for Hyrise) and $|\mathcal{S}_\mathsf{a}| \ll \text{size}(R)$. Sorting can be omitted if the partitions are already sorted by a, reducing the complexity to $\mathcal{O}(|\mathcal{S}_\mathsf{a}| \cdot \log |\mathcal{S}_\mathsf{a}| + \text{size}(R))$ for ordered relations.

[2]We assume an amortized complexity of $\mathcal{O}(1)$ for accessing the next element in the index and hash set insertion in $\mathcal{O}(1)$. In particular, incrementing the tree's iterator is constant if a is sorted, as we always insert keys at the tree's leaf with the highest value. We pre-allocate hash tables and guarantee enough buckets to prevent hash collisions or hash table resizing.

5.4 Inclusion Dependencies

In general, we can validate a single IND candidate $R.\mathsf{a} \subseteq S.\mathsf{x}$ by building a hash set of x's values and checking if each value of a is contained in this set. Multiple encoding characteristics and statistics can be exploited to accelerate validation. First, we often observe that $\text{size}(R) \gg \text{size}(S)$ when R is a fact table and S is a dimension table. Thus, the set of x's values is relatively small, and many tuples in R reference the same key in S. If a is dictionary-encoded, we do not have to probe each tuple for containment, but only the dictionary entries for each segment.

Second, we can use minimum and maximum values and continuity for further optimization. For instance, the IND $R.\mathsf{a} \subseteq S.\mathsf{x}$ cannot hold if $\min(\mathsf{a}) < \min(\mathsf{x})$ or $\max(\mathsf{a}) > \max(\mathsf{x})$, which can easily be derived from the segment statistics or the dictionaries. Furthermore, the IND must hold if $\min(\mathsf{a}) \geq \min(\mathsf{x})$, $\max(\mathsf{a}) \leq \max(\mathsf{x})$, and $\forall v \in [\min(\mathsf{x}), \max(\mathsf{x})] : v \in \mathsf{x}$, i. e., x contains continuous values. This rather straightforward reformulation allows us to drastically improve the validation performance for integer data types, e. g., numeric keys: if $\max(\mathsf{x}) - \min(\mathsf{x}) = \text{dist}(\mathsf{x}) + 1$, x must contain all values in $[\min(\mathsf{x}), \max(\mathsf{x})]$. For unique columns, $\text{dist}(\mathsf{x}) = \text{size}(\mathsf{x})$. Thus, if we know that x is a UCC, we can check for continuousness and ensure that x's minimal and maximal values match a.

The uniqueness property can either be given by an already validated UCC or derived by applying the same techniques as for UCC validation (set construction with index optimization for range-partitioned keys, see Section 5.1). In the latter case, we also detect a valid UCC on x, which we store as well and do not need to validate again if requested. We only fall back to probing a's values to the hash set if x is not continuous.

The general validation strategy of building a set for S.x's values and probing R.a's values has a complexity in $\mathcal{O}(\text{size}(S) + \text{size}(R))$. If x is continuous but unsorted, we can omit the probing step and reduce the complexity to $\mathcal{O}(|\mathcal{S}_{S.\mathsf{x}}| \cdot \log |\mathcal{S}_{S.\mathsf{x}}| + \text{size}(S) + |\mathcal{S}_{R.\mathsf{a}}|)$. For a range-partitioned and continuous integer key x, the complexity further decreases to $\mathcal{O}(|\mathcal{S}_{S.\mathsf{x}}| \cdot \log |\mathcal{S}_{S.\mathsf{x}}| + |\mathcal{S}_{R.\mathsf{a}}|)$ using the segment index or $\mathcal{O}(|\mathcal{S}_{S.\mathsf{x}}| + |\mathcal{S}_{R.\mathsf{a}}|)$ if we already validated that x is unique. Identifying a's minimum and maximum value in $\mathcal{O}(|\mathcal{S}_{R.\mathsf{a}}|)$ is always required. However, $|\mathcal{S}_{R.\mathsf{a}}| \ll \text{size}(R)$ and $|\mathcal{S}_{S.\mathsf{x}}| \ll \text{size}(S)$. For foreign keys in fact and dimension tables, $\text{size}(S) \ll \text{size}(R)$.

5.5 Ordering Dependency Candidates

From the previous description of validation techniques, we observe characteristics that yield rules for a beneficial order to validate dependency candidates. First, validating an IND $R.\mathsf{a} \subseteq S.\mathsf{x}$ always confirms a possible UCC S.x. If this UCC is also a candidate, we can skip its validation later. Second, we can skip the validation of an FD if any of the candidate columns is a UCC.

Furthermore, the rules generating dependency candidates can provide additional information. The rule for O-3 is an example of *candidate dependence*. Section 3.2 explains that we need an OD, an IND, and a UCC to apply this rewrite based on a range predicate. If it is invalid, an OD candidate $S.\mathsf{x} \mapsto S.\mathsf{y}$ can be rejected early using sampling. An IND candidate $R.\mathsf{a} \subseteq S.\mathsf{x}$ cannot be rejected before constructing S.x's value set if S.x does not contain all values present in R.a. Thus, the need to validate the IND candidate depends on the validation result of the OD candidate. We track this dependence and only validate the IND if the OD has not been rejected before. Combining all these observations, we obtain a clear candidate order by dependency type: we validate ODs first, INDs second, UCCs third, and FDs last.

6 Building Blocks for System Integration

This section presents techniques required to leverage dependency-based optimization. We describe how we propagate valid dependencies for each operator in the query plan during optimization. This knowledge is crucial to return correct and complete query results. Dependency-based optimizations rewrite joins into selections using the results of uncorrelated scalar subqueries (see join-to-predicate rewrite O-3 in Section 3.2), which require adjusted treatment in the query plan and enable further optimization during execution. Thus, we propose dedicated subquery handling concepts to leverage dynamic pruning using subquery results at execution.

6.1 Data Dependency Propagation

Evaluating the validity of dependencies for a specific logical operator is cumbersome, as operators can modify the required properties of a relation. For instance, UCCs are not valid after operators that duplicate tuples. INDs can be invalid after operators that remove tuples. Thus, we must successively propagate and adapt dependencies in the query plan for logical operators (C-2). Starting from a relation's declared or validated dependencies (see Section 4.1), each operator adds or removes dependencies. As plans can change after each optimization step, operators do not persist dependencies but recursively compute them on the fly based on their input operators' dependencies. Liu et al. [46] gave anecdotal evidence that this propagation is not trivial stating that implementing UCC propagation in Postgres was abandoned after two years.[3] The following paragraphs explain how we propagate dependencies as displayed in Table 1.

Unique column combinations. UCCs do not change if all required columns are part of the output and no functions modify the values. Initially unique columns can contain duplicates after (1) inner equi-joins $R \bowtie S$ where the join partner is not unique, (2) outer and (3) theta-joins, and (4) unions. However, new UCCs arise (5) for grouping columns/distinct selections and (6) for ungrouped aggregates.

Functional dependencies. FDs can always be derived from existing UCCs and ODs. Even (1) after joins where the join partner is not unique and after (2) outer or (3) theta-joins, a former UCC becomes the determinant of an FD with the remaining attributes of the input relation as dependents. These FDs remain unchanged as long as the involved attributes are part of the operator output.

Order dependencies. Per default, ODs do not change. Both join keys (7) of an equi-join form an OD with the respective join partner on the right-hand side. For such joins, existing ODs with the join key(s) on the left-hand side form transitive ODs with the other relation's join key(s). As we derive FDs from ODs, this behavior reflects transitive FDs for the join keys. ODs are invalidated (8) by union operators or if their attributes are not part of the output.

Inclusion dependencies. For an IND $R.\mathsf{a} \subseteq S.\mathsf{x}$, it is not obvious whether it should be propagated starting from R or S. Furthermore, INDs are also the most volatile dependency type: (9) a single selection on the right-hand side can invalidate the IND. Thus, we propagate INDs starting at the right hand-side and reduce the recursion depth: selections only propagate an input IND with x as the referenced column (10) for $\sigma_{\mathsf{x\ IS\ NOT\ NULL}}(S)$. In most cases, selections (9) and filtering joins (11) return an empty set of

[3]See https://commitfest.postgresql.org/35/2433/ (accessed Sep. 19, 2025).

Table 1: Dependency propagation rules for relations $R(\mathsf{a}, \mathsf{b}, \ldots)$, $S(\mathsf{x}, \mathsf{y}, \ldots)$, and $T(\mathsf{u}, \mathsf{v}, \ldots)$. By default, operators do not modify their inputs' data dependencies.

Operator	Input deps.	Output deps.
Unique column combinations		
(1) $R \bowtie_{R.\mathsf{a}=S.\mathsf{x}} S$	UCC b, UCC y	FD b $\rightarrow R \setminus$ b, FD y $\rightarrow S \setminus$ y
(2) $R ⟕_{R.\mathsf{a}=S.\mathsf{x}} S$	UCC b, UCC x	FD b $\rightarrow R \setminus$ b, FD y $\rightarrow S \setminus$ y
(3) $R \bowtie_{R.\mathsf{a}\,\theta\,S.\mathsf{x}} S$	UCC b, UCC y	FD b $\rightarrow R \setminus$ b, FD y $\rightarrow S \setminus$ y
(4) $R \cup R'$	UCC R.a, UCC R'.a	—
(5) $\gamma_{\mathsf{a,\,b,\,sum(c)}}(R)$	—	UCC {a, b}
(6) $\gamma_{\mathsf{avg(a)}}(R)$	—	UCC avg(a)
Order dependencies		
(7) $R \bowtie_{R.\mathsf{a}=S.\mathsf{x}} S$	—	OD a $\mapsto$ x, OD x $\mapsto$ a
(8) $R \cup R'$	OD R.a $\mapsto R$.b, OD R'.a $\mapsto R'$.b	—
Inclusion dependencies		
(9) $\sigma_{\mathsf{y}\,\theta\,c}(S)$	IND a $\subseteq$ x	—
(10) $\sigma_{\mathsf{x\ IS\ NOT\ NULL}}(S)$	IND a $\subseteq$ x	IND a $\subseteq$ x
(11) $S \ltimes T$	IND a $\subseteq$ x	—
(12) $S ⟕ T$	IND a $\subseteq$ x	IND a $\subseteq$ x
(13) $S \bowtie_{S.\mathsf{y}=T.\mathsf{u}} T$	IND a $\subseteq$ x, IND y $\subseteq$ u	IND a $\subseteq$ x

INDs and do not recurse further. Outer joins (12) and joins with an IND to the other relation's join key (13) preserve INDs. Other operators forward INDs if all columns are part of their output.

By propagating dependencies in the query plan, we enable dependency-based optimization out of the box and move further to making data dependencies first-class citizens of the DBMS.

6.2 Handling Predicates With Subquery Results

Usually, database optimizers rewrite subqueries to (semi-)joins (*subquery unnesting*) to avoid evaluating the subquery for each row [10, 26, 28, 37]. Optimization O-3 further rewrites joins to predicates containing results of uncorrelated scalar subqueries. We execute these predicates efficiently by evaluating the subqueries once and using their results like regular constants. However, we must estimate the predicate cardinalities even though the exact predicate values are unknown until execution (C-3). During execution, these predicates uncover potential for partition pruning.

Cardinality estimation. Estimating the cardinality of predicates using subquery results is necessary to place them well in the query plan. However, the results of scalar subqueries are unknown before execution. While simply calculating the cardinality of equality predicates using the column's distinct value count is a well-known coarse estimate [70], this approach does not work for range predicates with unknown lower and upper bounds (Figure 5).

We leverage the knowledge that the predicates generated by O-3 originated as (semi-)joins: whenever the pattern of a subquery predicate matches the rewrite, we perform the same estimation as for the original semi-join. The optimizer can use its regular estimation techniques, e. g., histogram-based estimation [83]. Without knowing the exact predicate values, the estimate for the semi-join is a proper approximation the cardinality estimator can provide. In particular, it matches the estimation without applying O-3, leading to similar placement in the plan. Different placements can effectively alter the join order, causing rather different, probably less beneficial query plans [68].

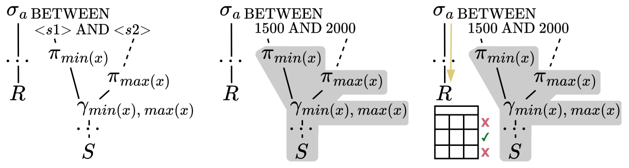
σ_a BETWEEN <s1> AND <s2>
$\pi_{min(x)}$
$\pi_{max(x)}$
$\gamma_{min(x), max(x)}$
R
S
σ_a BETWEEN 1500 AND 2000
σ_a BETWEEN 1500 AND 2000

Figure 5: Pruning using a predicate with subquery results. Logical plan, executed subtree, and dynamic pruning of *R*.

Partition pruning using subquery results. Commonly, horizontally partitioned databases prune partitions based on statistics, such as zone maps or range sets [55, 59, 88]. The DBMS skips partitions without tuples that match selection predicates, reducing the amount of data processed by all operators in the plan. Clearly, we cannot determine pruning criteria during optimization when predicate values have yet to be determined by query execution. However, we keep track of predicates that could enable pruning and shift from static partition pruning during optimization to *dynamic* pruning using subquery results during execution.

Figure 5 shows a plan that contains a range predicate with subquery results on *R*.a. Further operators, e. g., other selections or filtering semi-joins, can be below the predicate. During optimization, we only create a link between the predicate with scalar subquery results and the operator that first accesses relation *R* by collecting the predicate from subsequent operators. When scheduling physical operators for execution, we ensure to execute the subtree that determines the predicate values before any operator accessing relation *R*. Thus, the subquery is executed first, and we can perform dynamic partition pruning with concrete predicate values when executing the operator loading *R*.

Our technique for dynamic partition pruning using subquery results is generally applicable to predicates using results of uncorrelated scalar subqueries, e. g., data-induced predicates [59]. While we developed the approach to support dependency-based QO, it can be used to implement more general pruning techniques, which have been shown great performance benefits in distributed settings [89].

7 Evaluation

In this section, we evaluate the impact of dependency-based optimization techniques and the efficiency of metadata-aware data dependency validation (C-4). After briefly describing four standard benchmarks and characteristics of our experimental environment, we study the impact of dependency-based optimizations on five different DBMSs through SQL rewrites and compare the performance to optimization integrated into a DBMS. Then, we evaluate the performance impact per optimization technique and benchmark in the context of the additional dependency discovery overhead for Hyrise. Finally, we analyze the benefits of metadata-aware dependency validation algorithms and discuss the experimental results.

7.1 Experimental Setup

We evaluate our approach using four benchmarks: TPC-H [80], TPC-DS [79] (limited to 48 queries supported by Hyrise), the star schema benchmark (SSB) [56, 57], and the join order benchmark (JOB) [42]. TPC-H, TPC-DS, and SSB allow controlling the amount of data using a scale factor (SF). If not stated differently, this SF is 10. JOB is based on the real-world IMDB dataset and does not provide scaling. We conducted the experiments on one non-uniform memory access (NUMA) region of an Ubuntu 24.04 LTS server with an Intel Xeon Platinum 8180 CPU (28 cores/56 threads) and 378 GiB of local memory. Our Hyrise plug-in was implemented in C++ and compiled using LLVM-17. To ease interpretation, we use *symmetric logarithmic axes* [84] in Figures 7 and 10, which are linear close to 0 and logarithmic for larger values.

7.2 Optimization for Different DBMSs

First, we quantify the end-to-end impact of dependency-based QO for a production-like scenario that puts systems under load. Similar to real-world applications [4, 71, 82], we consider a multi-threaded, high-load scenario with 32 concurrent clients for five analytical DBMSs: *DuckDB* [66] (1.1.3), *MonetDB* [33] (11.51.3), *Umbra* [54] (24.11), *Hyrise* [19], and the commercial DBMS *SAP HANA Cloud* [50]. We conducted the experiments for SAP HANA on a cloud instance with 128 vCores (Intel Xeon Platinum 8260) and 1 008 GiB of RAM. Our goal is to optimize analytical workloads featuring large joins and group-bys [71, 76]. We excluded row-stores because they are not primarily designed to handle such workloads. Postgres 16, e. g., cannot execute individual TPC-H and TPC-DS queries within hours, even with key indexes.

Setup. We want to compare improvements achieved by dependency-based query optimization, not system performance. Thus, we report only relative runtime improvements per system. Each client executes permutations of all TPC-H, TPC-DS, SSB, and JOB queries for two hours. We measure the median runtime of all complete workload executions using four configurations: first, no primary and foreign keys are specified as a baseline. Second, we provide PKs and FKs via create or alter table statements. The systems can utilize index-based operators and own dependency-based optimizations. Third, we reformulate the SQL queries with optimizations O-1 (dependent group-by reduction) and O-3 (join-to-predicate rewrite) without PKs/FKs. We split subqueries generated by O-3 into separate statements and insert the results as concrete values into the subquery predicates to prevent the DBMSs from (i) unnesting the subqueries to the original joins and (ii) disadvantageous predicate placement. We omit O-2 (join-to-semi-join rewrite) because standard SQL cannot request semi-joins. Finally, we combine the latter two configurations. This configuration demonstrates the optimizations' potential as an upper bound, where systems can apply the best predicate orders.

Results. Figure 6 shows the relative runtime improvements per configuration over the systems' baselines. Rewriting joins to predicates (O-3) in SQL is highly beneficial for operator-at-a-time execution engines, such as MonetDB [33] Hyrise [19], because a faster operation blocks subsequent operators for a shorter time. The effect is smaller for pipelined engines, such as DuckDB [66], Umbra [54], and SAP HANA [50], that can start to operate already on partial outputs of pipelined joins.

MonetDB, SAP HANA, and Hyrise perform optimizations when PKs and FKs are known, such as join-to-semi-join (SAP HANA and Hyrise) or index joins (MonetDB and SAP HANA). DuckDB does not feature such optimizations and shows no improvements when providing PKs and FKs. For Umbra, JOB queries with PK-FK joins even cause performance degradations: FK indexes combined with severe cardinality underestimations tempt the optimizer to prefer index nested-loop joins over hash joins, deteriorating individual queries by two orders of magnitude. Replacing some of these joins with predicates (O-3) by additional

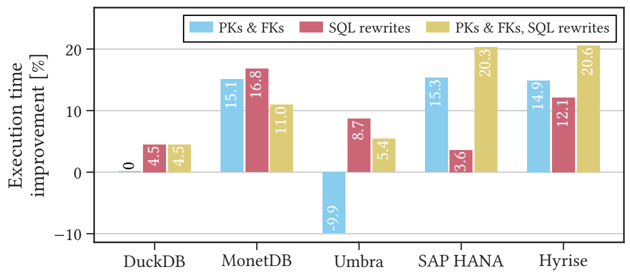

Figure 6: End-to-end workload execution time improvement of dependency-based QO by (i) PKs/FKs, (ii) SQL queries (O-1, O-3), and (iii) their combination over baselines without schema constraints under load (32 clients, MT). All systems benefit from using more than PKs/FKs.

SQL rewrites improves Umbra's performance, but using only these rewrites performs best. For MonetDB, the performance of individual (mostly JOB) queries when using either keys or rewrites is better than the combination: the cardinality estimations differ, resulting in worse choices of join orders and algorithms. Cost-based choices of different plan alternatives are not trivial, and some systems, such as DuckDB, do not implement dependency-based techniques at all.

However, the techniques themselves are beneficial for any system, and SAP HANA and Hyrise demonstrate that combining the techniques achieves the best performance. We also executed our integrated and automatic solution for Hyrise. Its performance closely resembles the upper bound of PK/FK and SQL rewrites providing concrete subquery predicate values with an improvement of 18.3 %, illustrating that our system integration performs well. For all systems, the runtime with additional rewrites further improves compared to providing (only) PKs and FKs.

7.3 Performance Impact and Dependency Discovery Overhead

Second, we analyze the three rewrites' impact on individual query latencies for Hyrise and compare it to the dependency discovery overhead. We perform single-threaded (ST) and multi-threaded (MT) benchmark executions with one client. The single-threaded setup allows us to assess the efficiency of optimized query plans without hiding latency by parallelism. Multi-threaded experiments are limited to the NUMA region's 28 physical cores to ensure stable measurements. We report the average latency of 100 repetitions within a time limit of 60 s per query. For the baseline execution (*Without deps.* in Table 2), we do not provide any schema-defined PKs and FKs. We expect the rewrites to have a varying impact based on the benchmark characteristics.

Improvement per workload. Table 2 depicts the latency impact of O-1 to O-3 individually and in combination for four benchmarks compared to the dependency discovery overhead (number of candidates, valid dependencies, and combined candidate generation and validation runtime). We achieve an average latency improvement of at least 9 % (6 % MT) through all benchmarks by combining all optimization techniques, where the dependency discovery overhead is at least one order of magnitude smaller than the saved execution for a single benchmark execution. Table 2 also compares the execution time when the system is aware of PKs/FKs and the additional improvement enabled by discovered UCCs and ODs.

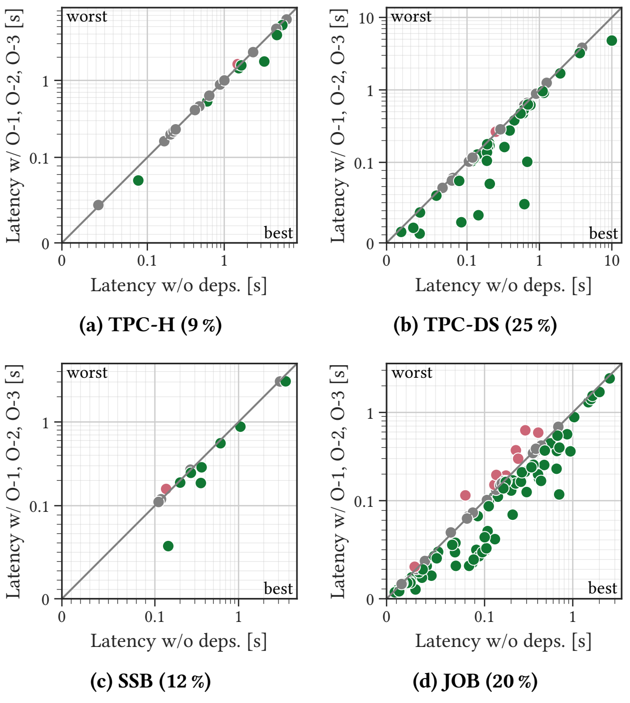

Figure 7: Individual queries' latencies with and without dependency-based optimization per benchmark query (ST). Average relative latency improvement in parentheses. Queries that change at least by ±5 % are colored (green/red). Note the bi-symmetric logarithmic axes (linear <0.1 s).

The impact of O-2 (join-to-semi-join rewrite) and O-3 (join-to-predicate rewrite) is high for TPC-DS and JOB. These benchmarks have snowflake schemas, which result in many joins of fact tables and dimension tables that can be rewritten. Each join rewritten to a predicate by O3 can also be turned into a semi-join by O-2. Thus, the impact of the two rewrites does not add up when all optimizations are applied. O-1 (dependent group-by reduction) is most beneficial for TPC-H, where aggregates are also dominant [18].

Exploiting dependencies beyond the schema (*+ UCCs, ODs*) yields further improvements for TPC-DS and JOB, as seen in the last row of Table 2. 17 TPC-DS and 66 JOB queries improve with geomean speedups of 35 % and 29 %, respectively. O-3 is the only technique requiring more than schema-defined dependencies, and benchmarks where O-3 is beneficial profit most from their discovery. For instance, TPC-DS's Q37 has a latency improvement over 90 %. The time to discover additional dependencies does not exceed a few milliseconds for all workloads. All rewrites improve performance, and the discovery overhead is always lower than the runtime saved for a single workload execution if there are valid dependencies.

Figure 7 visualizes the ST performance impact on individual queries. Each query is represented as a dot and placed with the baseline latency on the x-axis and the latency when applying the optimizations on the y-axis. The optimizations improve query latency if the query is below the diagonal line. For TPC-H, shown in Figure 7a, six out of 22 queries improve by at least 5 %. Q10 benefits most from reducing seven group-by columns to one and decreases its latency by 47 %. 36 out of 48 TPC-DS queries improve by up to 92 % (Figure 7b). We observe high relative latency

Table 2: Performance impact and dependency discovery overhead for four benchmarks and three individual rewrites (see Section 3.2), all rewrites combined, with schema-defined keys, and additional UCCs/ODs. Overall single- (ST) and multi-threaded (MT) execution time in seconds [s] and relative latency change [%] (one client). For dependency discovery, # is the number of dependency candidates, ✓ is the number of valid candidates, and ms is the total discovery time in milliseconds.

	Performance Impact							
	TPC-H (22 queries)		TPC-DS (48 queries)		SSB (13 queries)		JOB (113 queries)	
	ST [s (%)]	MT [s (%)]	ST [s (%)]	MT [s (%)]	ST [s (%)]	MT [s (%)]	ST [s (%)]	MT [s (%)]
Without deps.	37.5	18.1	38.1	16.8	10.4	4.0	33.3	20.6
Dep. group-by O-1	−1.9 (−5)	−0.6 (−3)	−1.0 (−3)	±0.0 (±0)	±0.0 (±0)	±0.0 (±0)	±0.0 (±0)	±0.0 (±0)
Join-to-semi-join O-2	−1.8 (−5)	−0.3 (−1)	−7.0 (−18)	−1.9 (−11)	−1.2 (−11)	−0.5 (−13)	−4.2 (−13)	−3.2 (−15)
Join-to-predicate O-3	−0.6 (−1)	±0.0 (±0)	−7.1 (−19)	−2.0 (−12)	−0.1 (−1)	−0.2 (−5)	−5.3 (−16)	−3.2 (−15)
Combined	−3.6 (−9)	−1.0 (−6)	−9.5 (−25)	−2.6 (−15)	−1.2 (−12)	−0.7 (−17)	−6.6 (−20)	−4.5 (−22)
PKs & FKs	34.5	17.0	30.6	14.7	9.5	3.5	29.0	17.4
+ UCCs, ODs	−0.5 (−1)	±0.0 (±0)	−2.0 (−6)	−0.5 (−3)	−0.3 (−3)	−0.1 (−4)	−2.3 (−8)	−1.3 (−7)

	Dependency Discovery											
	TPC-H			TPC-DS			SSB			JOB		
	#	✓	ms	#	✓	ms	#	✓	ms	#	✓	ms
Without deps.												
Dep. group-by O-1	9	4	<1	32	2	<1	7	0	<1	0	0	<1
Join-to-semi-join O-2	6	6	<1	15	11	<1	4	4	<1	10	10	260
Join-to-predicate O-3	22	9	<1	46	16	13	15	6	10	38	15	33
Combined	31	13	<1	85	25	13	22	7	10	40	17	285
PKs & FKs												
+ UCCs, ODs	24	3	<1	74	4	12	18	2	<1	30	7	21

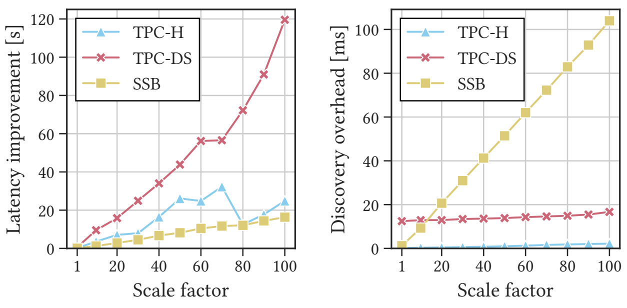

Figure 8: Latency improvement (seconds) and discovery overhead (milliseconds) for increasing scale factors (ST).

improvements when joins between fact tables and the *date* dimension are rewritten to range predicates and the physical order of tuples correlates to the date. In this case, we can dynamically prune large parts of the fact table. O-2 also achieves high absolute improvements and reduces the latency of Q95 by ≈4.9 s (51 %). However, Q1 degrades by 8 %: the optimizer does not place all semi-joins beneficially because of Hyrise's simple cost model. We do not observe performance degradations from O-3 because our adapted subquery handling provides stable query plans compared to the original joins, i. e., no join reordering. Eight out of 13 SSB queries improve by up to 24 %, whereas Q1.3 degrades (Figure 7c). For JOB, 83 out of 113 queries improve up to 83 %, as shown in Figure 7d. The UCC-based version of O-3 achieves high relative improvements, and dynamic subquery pruning (Section 6.2) contributes with latency improvements up to 31 % and a 13 % geometric mean speedup for 21 queries. Placing semi-joins is one of Hyrise's cost-based decisions. Due to the simple cost model and estimation errors, the placement is not always beneficial for this benchmark, as ten queries degrade.

Scalability. We execute the workloads and dependency discovery using a scale factor (SF) of 1 and SFs from 10 to 100 for TPC-H, TPC-DS, and SSB. The improvement in ST latency and to the ST dependency discovery overhead are shown in Figure 8. For all scale factors, the dependency discovery overhead is orders of magnitude smaller than the latency improvement and does not exceed 104 ms for SSB, 3 ms for TPC-H, and 17 ms for TPC-DS (all SF 100). The discovery scales linearly for SSB and TPC-H, and sub-linearly for TPC-DS: all tables with candidates grow linearly with the scale factor for TPC-H and SSB, whereas some dimension tables of TPC-DS grow slower than linearly. The latency improvement, i. e., the saved execution time, has the highest growth rate for TPC-DS and the lowest growth rate for SSB. For TPC-H, we observe a reduced latency improvement from SF 80 on, caused by disadvantageous placement decisions, most severely in Q21. Here, rewritten semi-joins with large build sides are pushed below selections.

Our experiments demonstrate that dependency-based QO improves performance. Datasets with normalized schemas benefit more than datasets with few dimension tables because of the higher potential for join rewrites. While exploiting schema-provided dependencies already has an impact, discovering and using additional dependencies can turn joins into selections, further improving performance. Integration of subqueries used for these selections yields stable query plans compared to the baseline. Discovering additional dependencies is amortized after a single benchmark execution.

7.4 Metadata-aware Dependency Validation

We investigate the efficiency of the metadata-aware data dependency validation algorithms presented in Section 5 and report the validation times of candidates generated for the benchmarks.

Performance impact of tailored validation. In an ablation study, we activate all optimizations described in Section 5 one by one and measure their impact. As a baseline, we use the fallback validation strategies: first, we always build hash sets for UCCs and referenced columns of INDs. Second, we probe all fields of foreign key columns for INDs. Third, we always sort by the entire column for ODs. Ultimately, we do not track candidate dependence and validate INDs regardless of the validity of ODs.

Figure 9 shows the overall validation times per benchmark (average of 100 executions) using naïve (left bar) and optimized validation techniques (right bar) as well as the impact of individual optimizations (middle bar). We observe a 2.5× speedup for JOB and improvements of at least two orders of magnitude for the other benchmarks. The impact of individual optimizations varies between datasets, highlighting that their combination is necessary. However, optimizations that are independent of data layout and encoding (candidate dependence, sampling for ODs, statistics-based invalidation for UCCs) greatly reduce validation times. Exploiting dictionary encoding (probe dictionary for INDs, insert entire segment to hash set for UCCs) or partitioning (segment index for UCCs) further improves the performance. The benefit of tailored validation techniques is most noticeable for TPC-H. Here, our system generates IND candidates for *lineitem*

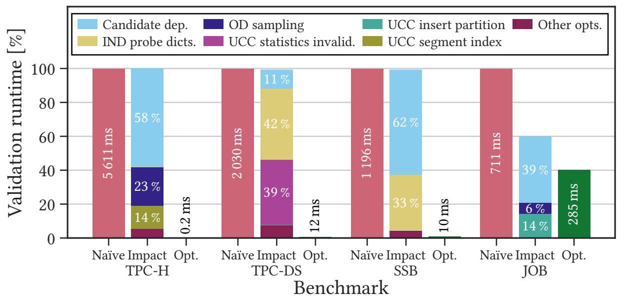

Figure 9: Dependency validation runtime improvement of metadata-aware, optimized strategies based on impact of individual optimizations relative to the naïve baseline. The combination of optimizations speeds up validation.

and OD and UCC candidates on *orders*, which are costly to validate by sorting a large column or adding it to a hash set. We observe similar behavior for TPC-DS and SSB. The validation improves less for JOB, as we often fall back to set-based validation. However, optimizations not relying on statistics improve runtime even if statistics are outdated and cannot be used.

Detailed validation performance per benchmark. Figure 10 shows the average validation time per candidate type and benchmark (average of 1 000 executions). For TPC-H, the system generates 31 candidates and validates each in less than 100 µs (Figure 10a). Five OD candidates are rejected by sampling, and only *region*.r_regionkey ↦ *region*.r_name is valid. We skip five INDs that depend on invalid ODs (see Section 5.5) even though they are FKs. One of them is *lineitem*.l_orderkey ⊆ *orders*.o_orderkey, which would dominate the discovery with a validation time of ≈1 s. The o_orderkey column is not continuous (only 25 % of the key range is populated [80, p. 86]), falling back to hash set-based validation. We validate the remaining IND *nation*.n_regionkey ⊆ *region*.r_regionkey within microseconds. We reject four UCCs using metadata and skip one on a PK that was already confirmed as a byproduct of IND validation. The remaining eight UCCs are range-partitioned PKs that we confirm using metadata and the name columns of *nation* and *region*, which consist of a single chunk. The UCC on *order*'s PK consumes the most time (≈70 µs) because we index 229 segments. Three FDs contain a UCC and are valid; the remaining four are rejected by metadata.

Figure 10b shows the validation times of 85 candidates generated for TPC-DS. Eight ODs are rejected by sampling, and the remaining four have *date_dim*'s sequential key on the left-hand side. Two candidates on string columns are confirmed in ≈5 ms, the others in less than 1 ms. All IND candidates represent valid FKs. We skip five INDs that depend on invalid ODs and confirm the remaining ones in under 1 ms each by exploiting sorted PKs. Due to traversing *inventory*'s 2 032 chunks, *inventory*.inv_date_sk ⊆ *date_dim*.d_date_sk takes ≈300 µs. Sorted PKs also help to confirm eleven UCCs within microseconds. IND validation already confirmed one UCC, and the remaining 27 are rejected by metadata. We confirm one and reject 20 FDs by metadata.

The validation times for SSB show a larger variance (Figure 10c). Two ODs are rejected by sampling, and two ODs on *date* are confirmed in ≈25 µs. The two invalid ODs lead to skipping two INDs, but *lineorder*.lo_orderdate ⊆ *date*.d_datekey falls back to the hash set-based check because d_datekey is not continuous [56, p. 4]. Thus, confirming this IND takes ≈10 ms. We confirm three UCC candidates candidates and reject eleven by

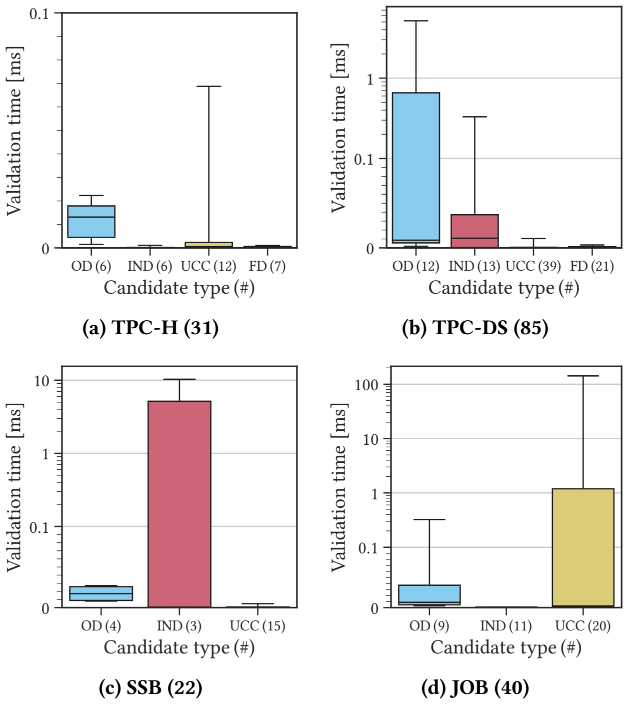

Figure 10: Average candidate validation runtimes for four benchmarks (number of candidates in parentheses). Note the symmetric logarithmic y-axis (linear <0.1 ms). Whiskers cover the entire value range.

metadata within a few microseconds. The UCC *date*.d_datekey was already confirmed by IND validation. No FD candidates were generated as no query groups by multiple columns of one table.

Validation takes the most time for JOB (Figure 10d). All ODs are rejected by sampling in less than 1 ms, allowing to skip all IND candidates. Three UCCs are rejected by metadata, and twelve are confirmed by metadata in microseconds. The remaining five candidates are valid UCCs but are not range-partitioned. Thus, we must use hash set construction for them, taking up to 105 ms for *char_name*.id and 142 ms for *name*.id. The JOB queries have no group-by statements; thus, there are no FD candidates.

Our experiments confirm that metadata-aware validation is efficient for rejecting and confirming candidates. We observe the longest validation times for valid candidates that fall back to default validation, i. e., sorting for ODs or hash set construction for UCCs and INDs. Metadata-aware validation scales well with relation size and number of partitions. Ordering the candidates by type allows skipping candidates known to be valid, and exploiting candidate dependence reduces the validation overhead.

7.5 Discussion

Our experiments highlight that dependency-based QO improves performance. Degradations of individual queries arise from estimation errors during costing, which can be further tuned.

Beneficial dependencies hold on dimension tables. Most data dependencies required for the selected query rewrites are genuine dependencies on dimension tables, which are unlikely to be rendered invalid by insertions or updates. However, the need to re-iterate dependency validation as part of an ETL process whenever data changes is a limitation of our current solution. We discuss a corresponding extension in the following section.

Dimension table modeling uncovers potential. If a dimension table's join key orders columns with selections, joins with the fact table can be reformulated to a scan of the fact table for both point and range predicates on the dimension. Thus, the disadvantage of performing many costly joins when using normalized schemas compared to flat tables [7, 44] can be reduced. When the dimension table's key is sequential and range-partitions the table, dependency validation can work solely on metadata, validating UCCs and INDs immediately without traversing entire columns.

Schema normalization facilitates optimization. The schema design influences the potential of dependency-based optimization and the efficiency of dependency discovery. Normalized snowflake schemas with fact and dimension tables lead to more joins that can be reformulated, especially in combination with valid dependencies additional to PKs and FKs (O-3). We observe high relative improvements for short-running queries with few rewritten joins. Long-running queries benefit if we rewrite many joins or a single dominating join (e. g., TPC-DS's Q95).

8 Conclusion and Future Work

We evaluated three data dependency-based query optimization techniques that substantially reduce workload latencies and presented methods to discover relevant data dependencies within milliseconds. We developed metadata-aware dependency validation algorithms to confirm or reject data dependency candidates purely based on database statistics and described how to adapt a DBMS for integration of dependency-based optimization. Our methods to handle predicates with scalar subqueries in cardinality estimation and pruning are not limited to dependency-based QO and can facilitate other pruning approaches. The experiments confirm the benefit of dependency-based QO for four benchmarks and five DBMSs. Compared to using only known primary/foreign keys, 17 TPC-DS and 66 JOB queries improve with 35 % and 29 % geometric mean speedups, respectively. Our approach allows to perceive data dependencies as pure metadata. While our approach requires adding optimization rules, we eliminate the need to specify constraints only for the purpose of QO by providing automatic, workload-driven dependency discovery, combining the concepts of autonomous databases [62] and data profiling. Discovered dependencies enable elaborate rewrites, and the ability to query them makes the additional metadata transparent.

We considered dependency discovery as part of ETL processes. Thus, a promising next step is to maintain dependencies of frequently changing datasets with an online approach that does not need to collect workload information. Besides leveraging indexes to track dependency validity, dependency versioning could even enable the combination of optimized and unoptimized plans operating on different partitions of append-optimized storage. Another direction of future work is extending dependency validation, e. g., to fully support FDs, n-ary dependencies, or additional dependency types.

Acknowledgments

We thank Taehyung Lee for detailed insights into the SAP HANA optimizer, Philipp Fent and Tobias Schmidt for assistance with the Umbra setup, and Martin Boissier for his feedback on the draft. This work was partially funded by SAP SE.

Artifacts

The source code, reproducibility scripts, and links to the datasets have been made available at https://github.com/HPI-Information-Systems/dependency-based-qo.

References

[1] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 967–980.

[2] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling relational data: a survey. *The VLDB Journal* 24, 4 (2015), 557–581.

[3] Rafi Ahmed, Allison W. Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zaït, and Thierry Cruanes. 2006. Cost-Based Query Transformation in Oracle. In *Proceedings of the International Conference on Very Large Databases (VLDB)*. 1026–1036.

[4] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2205–2217.

[5] RJ Atwal, Peter Boncz, Ryan Boyd, Antony Courtney, Till Döhmen, Florian Gerlinghoff, Jeff Huang, Joseph Hwang, Raphael Hyde, Elena Felder, Jacob Lacouture, Yves LeMaout, Boaz Leskes, Yao Liu, Alex Monahan, Dan Perkins, Tino Tereshko, Jordan Tigani, Nick Ursa, Stephanie Wang, and Yannick Welsch. 2024. MotherDuck: DuckDB in the cloud and in the client. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. 7 pages.

[6] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 168–180.

[7] Yuanzhe Bei, Thao Pham, Akshay Aggarwal, Nga Tran, Jaimin Dave, Chuck Bear, and Michael Leuchtenburg. 2019. Vertica Flattened Tables and Live Aggregate Projections: A Column-based Alternative to Materialized Views for Analytics. In *Proceedings of the International Conference on Big Data (BigData)*. 1749–1758.

[8] Siegfried Bell. 1997. Dependency Mining in Relational Databases. In *Proceedings of the International Joint Conference on Qualitative and Quantitative Practical Reasoning (ECSQARU-FAPR)*. 16–29.

[9] Siegfried Bell and Peter Brockhausen. 1995. *Discovery of Data Dependencies in Relational Databases*. Technical Report. University Dortmund. 6 pages.

[10] Srikanth Bellamkonda, Rafi Ahmed, Andrew Witkowski, Angela Amor, Mohamed Zaït, and Chun Chieh Lin. 2009. Enhanced Subquery Optimizations in Oracle. *PVLDB* 2, 2 (2009), 1366–1377.

[11] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 283–296.

[12] Johann Birnick, Thomas Bläsius, Tobias Friedrich, Felix Naumann, Thorsten Papenbrock, and Martin Schirneck. 2020. Hitting Set Enumeration with Partial Information for Unique Column Combination Discovery. *PVLDB* 13, 11 (2020), 2270–2283.

[13] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Proceedings of the TPC Technology Conference (TPCTC)*. 61–76.

[14] Marco A. Casanova, Luiz Tucherman, and Antonio L. Furtado. 1988. Enforcing Inclusion Dependencies and Referencial Integrity. In *Proceedings of the International Conference on Very Large Databases (VLDB)*. 38–49.

[15] Upen S. Chakravarthy, John Grant, and Jack Minker. 1990. Logic-Based Approach to Semantic Query Optimization. *ACM Transactions on Database Systems (TODS)* 15, 2 (1990), 162–207.

[16] Edgar F. Codd. 1971. *Further Normalization of the Data Base Relational Model.* Research Report RJ909. IBM. 33 pages.

[17] 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.

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

[20] Falco Dürsch, Axel Stebner, Fabian Windheuser, Maxi Fischer, Tim Friedrich, Nils Strelow, Tobias Bleifuß, Hazar Harmouch, Lan Jiang, Thorsten Papenbrock, and Felix Naumann. 2019. Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen Algorithms. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. 219–228.

[21] Ronald Fagin and Moshe Y. Vardi. 1984. The Theory of Data Dependencies - An Overview. In *Proceedings of the International Colloquium on Automata,*

Languages and Programming (ICALP). 1–22.
[22] Wenfei Fan, Floris Geerts, and Xibei Jia. 2008. Semandaq: a data quality system based on conditional functional dependencies. *PVLDB* 1, 2 (2008), 1460–1463.
[23] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2011. SAP HANA database: data management for modern business applications. *SIGMOD Record* 40, 4 (2011), 45–51.
[24] 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 Engineering Bulletin* 35, 1 (2012), 28–33.
[25] Philipp Fent, Altan Birler, and Thomas Neumann. 2023. Practical planning and execution of groupjoin and nested aggregates. *The VLDB Journal* 32, 6 (2023), 1165–1190.
[26] Richard A. Ganski and Harry K. T. Wong. 1987. Optimization of Nested SQL Queries Revisited. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 23–33.
[27] Goetz Graefe, Ross Bunker, and Shaun Cooper. 1998. Hash Joins and Hash Teams in Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Databases (VLDB).* 86–97.
[28] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. 1989. Extensible Query Processing in Starburst. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 377–388.
[29] Michael Hammer and Stanley B. Zdonik. 1980. Knowledge-Based Query Processing. In *Proceedings of the International Conference on Very Large Databases (VLDB).* 137–147.
[30] Chun-Nan Hsu and Craig A. Knoblock. 1996. Using Inductive Learning To Generate Rules for Semantic Query Optimization. In *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press, 425–445.
[31] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Comput. J.* 42, 2 (1999), 100–111.
[32] Toshihide Ibaraki and Tiko Kameda. 1984. On the Optimal Nesting Order for Computing N-Relational Joins. *ACM Transactions on Database Systems (TODS)* 9, 3 (1984), 482–502.
[33] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin* 35, 1 (2012), 40–45.
[34] International Organization for Standardization. 2023. *Information technology – Database languages SQL – Part 2: Foundation (SQL/Foundation).* Standard Specification ISO/IEC 9075-2:2023(E).
[35] Yannis E. Ioannidis. 1996. Query Optimization. *Comput. Surveys* 28, 1 (1996), 121–123.
[36] David Justen, Daniel Ritter, Campbell Fraser, Andrew Lamb, Nga Tran, Allison Lee, Thomas Bodner, Mhd Yamen Haddad, Steffen Zeuch, Volker Markl, and Matthias Boehm. 2024. POLAR: Adaptive and Non-invasive Join Order Selection via Plans of Least Resistance. *PVLDB* 17, 6 (2024), 1350–1363.
[37] Won Kim. 1982. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems (TODS)* 7, 3 (1982), 443–469.
[38] Jonathan J. King. 1980. Modelling Concepts for Reasoning About Access to Knowledge. In *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling.* 138–140.
[39] Jan Kossmann, Daniel Lindner, Felix Naumann, and Thorsten Papenbrock. 2022. Workload-driven, Lazy Discovery of Data Dependencies for Query Optimization. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 7 pages.
[40] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data dependencies for query optimization: a survey. *The VLDB Journal* 31, 1 (2022), 1–22.
[41] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-Time Analytical Processing with SQL Server. *PVLDB* 8, 12 (2015), 1740–1751.
[42] 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.
[43] Maurizio Lenzerini. 2002. Data Integration: A Theoretical Perspective. In *Proceedings of the Symposium on Principles of Database Systems (PODS).* 233–246.
[44] Mark Levene and George Loizou. 2003. Why is the snowflake schema a good data warehouse design? *Information Systems (IS)* 28, 3 (2003), 225–240.
[45] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2025. Data formats in analytical DBMSs: performance trade-offs and future directions. *The VLDB Journal* 34, 3 (2025), 30.
[46] Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sicheng Pan, Ge Li, Siddharth Jha, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. 2023. Leveraging Application Data Constraints to Optimize Database-Backed Web Applications. *PVLDB* 16, 6 (2023), 1208–1221.
[47] Claudio L. Lucchesi and Sylvia L. Osborn. 1978. Candidate Keys for Relations. *J. Comput. System Sci.* 17, 2 (1978), 270–279.
[48] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. 2001. Generic Schema Matching with Cupid. In *Proceedings of the International Conference on Very Large Databases (VLDB).* 49–58.
[49] Norman May, Alexander Böhm, and Wolfgang Lehner. 2017. SAP HANA - The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads. In *Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW).* 545–563.
[50] Norman May, Alexander Böhm, Daniel Ritter, Frank Renkes, Mihnea Andrei, and Wolfgang Lehner. 2025. SAP HANA Cloud: Data Management for Modern Enterprise Applications. In *Companion of the International Conference on Management of Data (SIGMOD).* 580–592.
[51] Michael Meier, Michael Schmidt, Fang Wei, and Georg Lausen. 2013. Semantic query optimization in the presence of types. *J. Comput. System Sci.* 79, 6 (2013), 937–957.
[52] Niloy Mukherjee, Shasank Chavan, Maria Colgan, Dinesh Das, Mike Gleeson, Sanket Hase, Allison Holloway, Hui Jin, Jesse Kamp, Kartik Kulkarni, Tirthankar Lahiri, Juan Loaiza, Neil MacNaughton, Vineet Marwah, Atrayee Mullick, Andy Witkowski, Jiaqi Yan, and Mohamed Zaït. 2015. Distributed Architecture of Oracle Database In-memory. *PVLDB* 8, 12 (2015), 1630–1641.
[53] Thomas Neumann. 2014. Engineering High-Performance Database Engines. *PVLDB* 7, 13 (2014), 1734–1741.
[54] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 7 pages.
[55] Anisoara Nica, Reza Sherkat, Mihnea Andrei, Xun Chen, Martin Heidel, Christian Bensberg, and Heiko Gerwens. 2017. Statisticum: Data Statistics Management in SAP HANA. *PVLDB* 10, 12 (2017), 1658–1669.
[56] Patrick E. O'Neil, Elizabeth J. O'Neil, and Xuedong Chen. 2009. *Star Schema Benchmark.* Standard Specification Revision 3. https://www.cs.umb.edu/~poneil/StarSchemaB.PDF (accessed Sep. 19, 2025).
[57] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Proceedings of the TPC Technology Conference (TPCTC).* 237–252.
[58] Oracle. [n. d.]. *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 (accessed Sep. 19, 2025).
[59] Laurel J. Orr, Srikanth Kandula, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates Through Joins in Big-Data Clusters. *PVLDB* 13, 3 (2019), 252–265.
[60] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *PVLDB* 8, 10 (2015), 1082–1093.
[61] Thorsten Papenbrock and Felix Naumann. 2017. A Hybrid Approach for Efficient Unique Column Combination Discovery. In *Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW).* 195–204.
[62] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 6 pages.
[63] 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 (JIDM)* 9, 1 (2018), 3–19.
[64] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. 2019. How I Learned to Stop Worrying and Love Re-optimization. In *Proceedings of the International Conference on Data Engineering (ICDE).* 1758–1761.
[65] Yiming Qiao and Huanchen Zhang. 2025. Data Chunk Compaction in Vectorized Execution. *Proceedings of the ACM on Management of Data (PACMMOD)* 3, 1 (2025), 26:1–26:25.
[66] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 1981–1984.
[67] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB* 6, 11 (2013), 1080–1091.
[68] Naveen Reddy and Jayant R. Haritsa. 2005. Analyzing Plan Diagrams of Database Query Optimizers. In *Proceedings of the International Conference on Very Large Databases (VLDB).* 1228–1240.
[69] El Kindi Rezig, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, Ahmed R. Mahmood, and Michael Stonebraker. 2021. Horizon: Scalable Dependency-driven Data Cleaning. *PVLDB* 14, 11 (2021), 2546–2554.
[70] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 23–34.
[71] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *Proceedings of the International Conference on Data Engineering (ICDE).* 1802–1813.
[72] Shashi Shekhar, Babak Hamidzadeh, Ashim Kohli, and Mark Coyle. 1993. Learning Transformation Rules for Semantic Query Optimization: A Data-Driven Approach. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 5, 6 (1993), 950–964.
[73] Shashi Shekhar, Jaideep Srivastava, and Soumitra Dutta. 1988. A Formal Model of Trade-off between Optimization and Execution Costs in Semantic Query

Optimization. In *Proceedings of the International Conference on Very Large Databases (VLDB)*. 457–467.

[74] Sreekumar T. Shenoy and Z. Meral Özsoyoglu. 1987. A System for Semantic Query Optimization. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 181–195.

[75] Michael D. Siegel, Edward Sciore, and Sharon C. Salveter. 1992. A Method for Automatic Rule Derivation to Support Semantic Query Optimization. *ACM Transactions on Database Systems (TODS)* 17, 4 (1992), 563–600.

[76] Jan Vincent Szlang, Sebastian Breß, Sebastian Cattes, Jonathan Dees, Florian Funke, Max Heimel, Michel Oleynik, Ismail Oukid, and Tobias Maltenberger. 2025. Workload Insights From the Snowflake Data Cloud: What Do Production Analytic Queries Really Look Like? *PVLDB* 18, 12 (2025), 5126–5138.

[77] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. 2012. Fundamentals of Order Dependencies. *PVLDB* 5, 11 (2012), 1220–1231.

[78] 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.

[79] Transaction Processing Performance Council. 2021. *TPC Benchmark DS*. Standard Specification Version 3.2.0. http://tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v3.2.0.pdf (accessed Sep. 19, 2025).

[80] Transaction Processing Performance Council. 2022. *TPC Benchmark H*. Standard Specification Revision 3.0.1. http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf (accessed Sep. 19, 2025).

[81] Jeffrey D. Ullman. 1988. *Principles of Database and Knowledge-Base Systems, Volume I*. Principles of computer science series, Vol. 14. Computer Science Press.

[82] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. *PVLDB* 17, 11 (2024), 3694–3706.

[83] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *PVLDB* 14, 9 (2021), 1640–1654.

[84] J. Beau W. Webber. 2013. A bi-symmetric log transformation for wide-range data. *Measurement Science and Technology* 24, 2 (2013), 3 pages.

[85] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. 2020. Managing data constraints in database-backed web applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 1098–1109.

[86] Clement T. Yu and Wei Sun. 1989. Automatic Knowledge Acquisition and Maintenance for Semantic Query Optimization. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 1, 3 (1989), 362–375.

[87] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *PVLDB* 17, 2 (2023), 148–161.

[88] Mohamed Ziauddin, Andrew Witkowski, You Jung Kim, Janaki Lahorani, Dmitry Potapov, and Murali Krishna. 2017. Dimensions Based Data Clustering and Zone Maps. *PVLDB* 10, 12 (2017), 1622–1633.

[89] Andreas Zimmerer, Damien Dam, Jan Kossmann, Juliane Waack, Ismail Oukid, and Andreas Kipf. 2025. Pruning in Snowflake: Working Smarter, Not Harder. In *Companion of the International Conference on Management of Data (SIGMOD)*. 757–770.