The VLDB Journal
https://doi.org/10.1007/s00778-021-00676-3

REGULAR PAPER

Data dependencies for query optimization: a survey

Jan Kossmann[1] · **Thorsten Papenbrock**[1] · **Felix Naumann**[1]

Received: 7 May 2020 / Revised: 12 March 2021 / Accepted: 26 March 2021

Abstract
Effective query optimization is a core feature of any database management system. While most query optimization techniques make use of simple metadata, such as cardinalities and other basic statistics, other optimization techniques are based on more advanced metadata including data dependencies, such as functional, uniqueness, order, or inclusion dependencies. This survey provides an overview, intuitive descriptions, and classifications of query optimization and execution strategies that are enabled by data dependencies. We consider the most popular types of data dependencies and focus on optimization strategies that target the optimization of relational database queries. The survey supports database vendors to identify optimization opportunities as well as DBMS researchers to find related work and open research questions.

Keywords Query optimization · Query execution · Data dependencies · Data profiling · Unique column combinations · Functional dependencies · Order dependencies · Inclusion dependencies · Relational data · SQL

1 Efficient querying with data dependencies

Increasing the performance of modern database management systems is a major objective of database research. In this context, research has accelerated the processing of queries through advances in different areas, such as utilization of new hardware technologies, improved implementations of database operators, or sophisticated query plan modifications as part of the query optimization process. In this survey, we consider *data dependencies* and how they can be utilized for more efficient query processing. More specifically, we provide a comprehensive survey of methods that exploit data dependencies during query optimization and execution in order to process relational database queries faster.

Most of the data dependency-driven methods discussed in this survey have been known for years, but many of them are rarely implemented in actual database systems. To make the techniques more accessible, we present them in a clear and concise way with intuitive descriptions and links to detailed material. A reason for data dependencies not being used in modern query processing engines might be that the required data dependencies of a given dataset are often unknown: they have never been specified, are difficult to discover, and expensive to maintain. However, the latest developments in the field of data profiling tackle these issues for various types of data dependencies and thus increase the feasibility of discovering and thereby the availability of data dependencies [3]. Note that data profiling algorithms discover all valid dependencies in a given dataset instance. In many use cases, it is important to distinguish semantically meaningful dependencies from accidentally valid, i.e., spurious ones. For query optimization, however, this distinction of *genuine* and *spurious* dependencies is irrelevant: If a data dependency is valid it can be used for optimization, regardless of its genuineness. Thus, query optimization techniques based on discovered data dependencies become more viable; their implementation and further research in this area can, therefore, be reconsidered.

Contributions With this survey, we give an overview of about 60 query optimization techniques that are based on data dependencies. We provide brief and intuitive descriptions of all techniques and classify them into different types of application areas, data dependencies, and optimization phases. Most dependency-driven optimization techniques have been published or at least mentioned in scientific literature; these techniques are usually rather sophisticated optimizations.

✉ Jan Kossmann
jan.kossmann@hpi.de

Thorsten Papenbrock
thorsten.papenbrock@hpi.de

Felix Naumann
felix.naumann@hpi.de

[1] Hasso Plattner Institute, University of Potsdam, Potsdam, Germany

Published online: 14 June 2021

Other techniques have not been scientifically published and constitute more straightforward optimizations. This survey captures all techniques, regardless of their origin.

No survey has yet been published that provides an overview of the query optimization techniques based on functional, uniqueness, order, and inclusion dependencies. An extensive examination of order and uniqueness properties in the context of operators of the relational algebra has, though, been given by Paulley's dissertation [101]. Our survey is based in parts on his findings but incorporates many further and more recent research: We add many additional optimization techniques as well as additional types of dependencies. We also provide a reference matrix in Table 1, which summarizes the optimizations for different types of data dependencies in different areas of application with regard to the query optimization process.

Focus This survey focusses solely on utilizing data dependencies for effective query optimization and, therefore, does not consider optimizations that are not related to data dependencies. While we briefly discuss data dependency discovery algorithms as part of Sect. 3, we do not cover these algorithms thoroughly and refer to [3] for detailed explanations of the algorithms and data profiling in general. Because this is *not* an experimental survey, we do not judge the optimizations' effectiveness, ease of application, or relevance in terms of how likely they get triggered in real-world workloads; instead, we focus on classifications and intuitive explanations.

Overview The following two sections summarize the foundations in the area of query optimization (Sect. 2) and the area of data dependencies (Sect. 3); the common terminology and technical background are necessary to understand the techniques and classifications presented thereafter. When we introduce the four data dependency types considered in this survey in Sect. 3, we also discuss automatic discovery and maintenance algorithms of data dependencies, as these make data dependencies more generally available and, hence, can be seen as an enabler for the dependency-based query optimization techniques. Section 4 provides an overview and classifications for the presented optimization techniques in form of an optimization technique matrix. This matrix serves as a reference for all of the following descriptions of optimization techniques. Sections 5, 6, 7, and 8, then, describe the proposed optimizations, including examples where appropriate, for all four dependencies. Section 9 discusses further optimization opportunities before Sect. 10 provides a short summary of the paper, concluding remarks, open research questions, and ideas for future work.

2 Query optimization

Relational database management systems (DBMSs) are usually queried with SQL in a declarative way. The query engine of a DBMS, then, transforms these queries into physical query execution plans. In this process, *query optimization* is the task of finding an optimal (or at least very good) physical execution plan with respect to the plan's execution time. It is crucial to find efficient query plans, because the execution times of different physical plans (that yield identical results) for the same query can vary by orders of magnitude [61] and "the runtime system alone could never get that good performance without an optimizer" [92]. In most systems, the query optimization process is handled by an interplay of three main activities: (i) *cost-independent transformation* (also referred to as *query rewriting*), (ii) *cardinality and cost estimation*, (iii) and *cost-based transformation*. In the following, we briefly outline how these three components operate and how the optimization techniques presented in this survey relate to them. Later, in Sect. 4 we use the three optimization activities to classify the various dependency-driven optimization techniques.

Note that not all systems follow this division explicitly. Cascades-style optimizers [50], for example, take a combined approach. The data dependency-based query optimization rules presented in this paper can still apply to such systems.

Cost-independent transformation describes the process of rewriting a query into a semantically equivalent but presumably more efficient query via static, rule-based transformations [102]. The required rules are characterized by being generally applicable instead of being cost dependent, that is, they generally[1] produce superior, more efficient query formulations [103]. Examples for such rewrites are the resolution of views, the removal of `DISTINCT` clauses and predicate push-downs. For several rewrites, the presence of certain data dependencies, such as information on keys, is a prerequisite to produce semantically equivalent plans. Hence, many of the dependency-based optimization techniques presented in this survey address query cost-independent transformations.

Cardinality and cost estimation serve to estimate the cost of a query plan a priori. Cost is an indispensable metric to compare different plan alternatives for cost-based transformations. Via cost models, a query optimizer estimates the expected cost for a certain query plan and its individual operators based on cardinality information, logical operator specification, and hardware-specific costs [80]. Cardinality information is usually derived from statistics, e.g., histograms or samples. According to Leis et al. [75], esti-

[1] In theory, downstream *cost-dependent* transformations *could* produce more efficient plans based on the unmodified version. This behavior is unlikely and depends on the concrete system and query.

mation errors are often responsible for suboptimal plans. Data dependencies can be applied during cardinality and cost estimation to obtain more accurate cardinality estimates or estimates for otherwise missing cardinalities; in this way, data dependencies serve to mitigate estimation errors.

Cost-based transformation is the process of improving physical query plans via cost-driven transformations that depend on the database instance at hand [63]. Based on transformation rules, the optimizer repeatedly generates different plan alternatives, then requests estimates for the plans' costs from the aforementioned cost models based on concrete cardinality information, and finally chooses the most efficient plan based on these estimated costs. Examples for query plan optimizations are predicate reordering, the ordering of join operators, the selection of execution strategies for operators with different implementations (e.g., hash-, sort-, or index-based joins), and statistic-based partition pruning. It is apparent that the success of such plan optimizations strongly depends on the underlying data, the specific query, and the query's parameter values.

Data dependency information can be applied also in the cost-based transformation phase to generate more efficient plans. A table scan operation, for instance, usually scans the entire table sequentially. With the information that the data is sorted and in-memory, however, the optimizer can instruct the execution engine to execute it as a binary search; and, with functional dependency information, it can substitute the scan attributes with other attributes that are potentially more efficient to scan. Please note that some of the techniques presented in this survey might require the extension of existing operators or even the implementation of alternative operators, such as binary scans, early-aborting joins, or caching subquery executions.

3 Data dependencies

Data dependencies usually provide information about multiple attributes, sometimes even across different relations, and how they relate to each other. The four most popular types of data dependencies for query optimization according to the number of optimizations that use data dependencies are *unique column combinations*, *functional dependencies*, *order dependencies*, and *inclusion dependencies*. In this section, we give a brief overview of these four types of data dependencies and provide their formal definitions. Traditionally, data dependencies stem from data modeling and schema design, e.g., 3NF synthesis, or BCNF decomposition, but data profiling identifies data dependencies from the data themselves, independently of such processes. Because the discovery of data dependencies (of any type) is NP-complete [5] and sometimes even W[2]- to W[3]-complete [35], mining data dependencies is challenging. For this reason, we also provide pointers to the most recent automatic discovery and maintenance algorithms, which in practice are sufficiently fast to be useful in the context of query optimization on real-world datasets. An introductory overview of data profiling techniques can be found in [5], while a comprehensive survey is given in [3].

3.1 Unique column combinations (UCCs)

A *unique column combination* is a set of attributes whose projection on some relational instance has no duplicate entry—all entries are unique [53]. Thus, UCCs functionally determine all other attributes and, hence, are sometimes denoted as *candidate keys* [105].

Examples for UCCs are the combined attributes first-name, lastname, address, date-of-birth in a person table, or an auto-incremented id column that is by its definition a UCC. It is worth noting that most relational database management systems recommend the existence of at least one key per relation, i.e., for such systems, we can expect at least one UCC in every relational instance that can potentially be utilized to optimize queries.

In query optimization, UCCs serve to avoid unnecessary duplicate eliminations, i.e., `DISTINCT` calls, obtain improved cardinality estimations, and optimize joins.

Due to the relevance of keys in the relational model, unique column combinations are an old and well-established concept in database theory [79].

Definition 1 (*Unique column combination*) Given a relational instance r over a relation R, we formally say that a *column combination* $X \subseteq R$ is *unique* (UCC) for R, iff $\forall r_i, r_j \in R, i \neq j : r_i[X] \neq r_j[X]$. A UCC is said to be *minimal*, if no subset of that UCC exists for which the above constraint also holds; hence, $\forall X' \subset X : \exists r_i, r_j \in R, i \neq j : r_i[X] = r_j[X]$.

Origin of UCCs In general, there are four ways to introduce uniqueness in relational databases: (i) Database users and applications can explicitly produce uniqueness when processing existing data by utilizing SQL's `DISTINCT` operation; other SQL clauses, such as `EXCEPT`, `UNION`, `INTERSECT`, or `GROUP BY` may also create uniqueness during query execution if they remove duplicate values. In such cases, query optimizers can infer that the data will be unique at a certain point of the query plan. (ii) Most relational database systems allow specifying unique constraints via SQL DDL, such as `UNIQUE` or `KEY` that automatically enforce uniqueness on certain attributes or attribute sets. (iii) Columns or column combinations can be unique by their very nature, such as UUIDs in computer systems or passport_number for the citizens of a particular country. (iv)

UCCs can occur by chance,[2] especially for column combinations containing many columns.

Discovery and maintenance of UCCs Unique column combinations, especially minimal ones, are neither obvious nor simple to determine, as UCC discovery is a problem in $\mathcal{O}(2^m)$ for datasets with m attributes [5,77]. Therefore, efficient, automatic UCC discovery algorithms exist, which can serve these unique column combinations to a query optimizer. Examples of such algorithms are DUCC [53], and HPIVALID [13]. With the incremental profiling algorithm SWAN [4], unique column combinations can be incrementally maintained.

While UCCs are a special case of functional dependencies, which we discuss below, specialized UCC discovery algorithms are more efficient. Thus, we discuss the use in query optimization separately.

3.2 Functional dependencies (FDs)

Real-world data often follow semantics according to which values in certain attributes functionally determine values in other attributes. Hence, *functional dependencies* indicate relationships between database attributes [27]. An FD is a statement $X \rightarrow Y$, expressing that any two records in a relational instance r that have the same values in the attributes $X \subseteq R$ also have the same values in the attributes $Y \in R$; the attributes in X *functionally determine* the attributes in Y.

For example, for a relation with address data, the functional dependency city, street address $\rightarrow$ zip code could hold; another relation about planets should feature the functional dependency diameter $\rightarrow$ circumference.

This dependency is a popular property, because it serves many use cases, including schema normalization, consistency checking, and data exploration. In query optimization, one can use FDs mainly to remove unnecessary attributes in various SQL operations and to improve cardinality estimates by transferring these estimates via FDs between sets of attributes.

Definition 2 (Functional dependency) A *functional dependency* (FD) $X \rightarrow A$ of a relation R holds in a relational instance r over R, iff $\forall s, t \in r : s[X] = t[X] \Rightarrow s[A] = t[A]$. The left-hand-side (LHS) attributes X are called *determinant* and the right-hand-side (RHS) attribute A is called *dependent*. Multiple FDs with the same determinant attributes X can be grouped and written as $X \rightarrow Y$, with $Y = \bigcup A_i$.

Some FDs have special properties. An FD $X \rightarrow A$ is called *minimal* if no attribute $B \in X$ exists such that $X \setminus B \rightarrow A$ is still a valid FD. An FD is called *non-trivial* if $A \notin X$. Although minimality and non-triviality play an important role in data profiling, these two properties have no special meaning for query optimization.

A UCC $X \subseteq R$ induces FDs on all attributes that are not part of the UCC: $\forall A \in R \setminus X : X \rightarrow A$. Thus, all optimizations that we introduce for functional dependencies later on are also applicable to unique column combinations.

Origin of FDs In general, functional dependencies exist by the very nature of the underlying data: they represent real-world constraints, semantic relationships, and physical laws that are reflected in the data. Functional dependencies can also be artificially introduced to datasets, e.g., with surrogate keys, which by definition functionally determine all other attributes. In addition, after filtering a relation with an equality predicate on A, the FD $X \rightarrow A$ holds for every X [101, p. 68]. Paulley explains how scalar functions can generate FDs and discusses for all relational operators which FDs are valid on their output, given a set of FDs on their input [101, pp. 71–104]. Very many further FDs exist coincidentally on a given dataset. While they carry no semantic meaning, they can nevertheless be exploited by query optimization techniques.

Discovery and maintenance of FDs Functional dependencies are usually not provided with the data and determining them manually is hard. More specifically, the FD discovery problem is in $\mathcal{O}(2^m \cdot (\frac{m}{2})^2)$ for m attribute sized datasets [5,77]. Therefore, a variety of automatic FD discovery algorithms, such as TANE [55], FDEP [41], and HYFD [96] have been proposed. The surveys by Liu et al. [77] and Papenbrock et al. [99] present and compare different FD profiling techniques from both a theoretical and a practical perspective, respectively. To discover FDs incrementally or maintain the FDs of a dynamic dataset over time, various incremental profiling algorithms exist [20,108,121].

3.3 Order dependencies (ODs)

An *order dependency* (OD) is a statement of the form $\boldsymbol{X} \mapsto \boldsymbol{Y}$[3] specifying that ordering a relational instance r by the attribute list $\boldsymbol{X} \subset R$ also orders r by the attributes $\boldsymbol{Y} \subset R$. Given an OD, we thus know how order decisions on certain attributes propagate to orders of other attributes; this knowledge can be used to optimize order decisions.

To give an intuition, in a date table, the OD month $\mapsto$ quarter holds, but the inverse OD quarter $\mapsto$ month is not valid. Another typical example is salary $\mapsto$ taxrate. One can find many ODs in real-world datasets. For example, the ncvoter[4] dataset and data from a real-world ERP system con-

[2] In this context, *chance* means that dependencies do not origin from data modeling or semantics, but occur rather randomly due to a large value codomain.

[3] Bold symbols $\boldsymbol{X}$ indicate lists, whereas standard symbols X indicate sets.

[4] North Carolina election data: www.ncsbe.gov/Data.

tain many order dependencies. Order dependencies play an important role in query optimization, because many relational operators use sorting or exploit already sorted data for their execution. More specifically, ODs help in selecting suitable operator implementations and support finding good query plans. They can also be used to, for instance, effectively rewrite, remove, or inject `ORDER BY` clauses.

Ginsburg and Hull were the first to formally introduce the concept of order dependencies [46,47]. In this paper, we use the notation of Szlichta et al. [115].

Definition 3 (*Order dependency*) For two lists of attributes $\boldsymbol{X}$ and $\boldsymbol{Y}$ of a relation R, the *order dependency* (OD) $\boldsymbol{X} \mapsto \boldsymbol{Y}$ holds in relation instance r over R, iff $\forall s, t \in r : s[\boldsymbol{X}] \preceq t[\boldsymbol{X}] \Rightarrow s[\boldsymbol{Y}] \preceq t[\boldsymbol{Y}]$.

Note that the comparison operator $\preceq$ compares the $\boldsymbol{X}$ and $\boldsymbol{Y}$ values attribute-wise, i.e., lexicographically via $\leq$ with the first attribute in each list being the most significant one. Following SQL semantics, the comparison is data-type specific, which means that it is numerical for numbers and lexicographical for strings. In principle, ODs support different comparators including $\prec$, $\preceq$, $=$, $\succeq$, and $\succ$ as well as combinations of these comparators. An OD with the comparator $=$, for example, is equivalent to a functional dependency [115] and $\forall s, t \in r : s[\boldsymbol{X}] \preceq t[\boldsymbol{X}] \Rightarrow s[\boldsymbol{Y}] \succeq t[\boldsymbol{Y}]$ means that an ascending $\boldsymbol{X}$ order implies a descending $\boldsymbol{Y}$ order. While the optimizations presented in this paper are often extendable to other comparator types, we assume the comparator to be $\preceq$ ($\overset{\preceq}{\mapsto}$), if not stated differently. Please note that $\preceq$ induces a total ordering.

Origin of ODs There are three ways to introduce order in relational data (*order decisions*): First, users can explicitly create order by specifying an `ORDER BY` clause; second, some database operations produce ordered results as a side-effect of their implementation, e.g., sort-merge joins or sort-based aggregates; third, data can also be naturally ordered by, for instance, timestamp or auto-incremented surrogate key attributes during data ingestion.

Discovery and maintenance of ODs The discovery problem for ODs (in set-based notation) is in $\mathcal{O}(2^m)$ with m being the number of attributes in the dataset [117]. However, automatic discovery algorithms, such as FastOD by Szlichta et al. [117], are efficient in practice, because they use clever search space pruning and most ODs in real-world datasets actually appear relatively early in the discovery process. Similar to the other types of data dependencies, the incremental discovery has also been studied for ODs. For example, a recent incremental discovery algorithm for pointwise ODs is IncPOD [119].

3.4 Inclusion dependencies (INDs)

If all values in the projection of some attribute combination $\boldsymbol{X}$ also occur in the projection of some attribute combination $\boldsymbol{Y}$ (of the same or a different relation), then an *inclusion dependency* exists between $\boldsymbol{X}$ and $\boldsymbol{Y}$. If, furthermore, the referenced attribute combination is a key, i.e., a UCC, for its relation, the inclusion dependency is a *foreign-key* candidate—in other words, INDs are prerequisites for a foreign-key relationships. For example, the three inclusion dependencies

$$\text{click.website} \subseteq \text{website.url}$$
$$\text{sales.item} \subseteq \text{items.id}$$
$$\{\text{ship.lname, ship.bdate}\} \subseteq \{\text{addr.name, addr.dob}\}$$

might represent foreign-key relationships. Such inclusion dependencies can be used in data integration and data linkage scenarios to connect tables across multiple data sources by suggesting join paths. Inclusion dependencies are, however, also useful if they do not describe foreign-key relationships. The inclusion dependency nails.supplier $\subseteq$ screws.supplier, for example, asserts that all nail suppliers also supply screws without supplier being a key in screws. Such general INDs can be used for data exploration and, as we will show in this section, query optimization.

Our formal definition of inclusion dependencies follows the syntax of Casanova et al. [21] and De Marchi [81]:

Definition 4 (*Inclusion dependency*) An *inclusion dependency* (IND) $R_i[\boldsymbol{X}] \subseteq R_j[\boldsymbol{Y}]$ is valid for the two relational instances r_i and r_j of schemata R_i and R_j and the attribute lists $\boldsymbol{X}$ and $\boldsymbol{Y}$ with cardinalities $n = |\boldsymbol{X}| = |\boldsymbol{Y}|$ iff $\forall t_i \in r_i, \exists t_j \in r_j : t_i[\boldsymbol{X}] = t_j[\boldsymbol{Y}]$. We write $\boldsymbol{X} \subseteq \boldsymbol{Y}$ or $R_i.\boldsymbol{X} \subseteq R_j.\boldsymbol{Y}$ if it is clear from the context that an IND is meant; in these cases, the projection is implicit.

Note that the dependent ($\boldsymbol{X}$) and referenced ($\boldsymbol{Y}$) part denote attribute *lists* for INDs, i.e., their attribute order may differ from the attribute order in R and they may contain repeated attributes. By removing attributes with the same indices from the lists $\boldsymbol{X}$ and $\boldsymbol{Y}$, we can derive generalizations $\boldsymbol{X}' \subseteq \boldsymbol{Y}'$ with $\boldsymbol{X}' \subset \boldsymbol{X}$ and $\boldsymbol{Y}' \subset \boldsymbol{Y}$ from a valid IND $\boldsymbol{X} \subseteq \boldsymbol{Y}$ that are also valid. This is important for query optimization, because a query might not contain all attributes of a known IND but a subset that also forms a valid IND.

Origin of INDs Relational databases contain INDs, because data models representing real-world data often rely on relationships between tables, and INDs are a prerequisite for foreign-key relationships. Most DBMS implementations allow users to specify foreign keys, e.g., via SQL's `FOREIGN KEY X REFERENCES Y`, which can also specify the behavior if a tuple is `null` in one of the key's attributes via `MATCH (FULL|PARTIAL|SIMPLE)`.

Discovery and maintenance of INDs Inclusion dependencies are different from the previously mentioned types of dependencies as they can span across multiple relations, the attribute order in the dependency *does* matter, and the position of value combinations within the sets of left- and right-hand-side value combinations *does not* matter. For this reason, their discovery is in $\mathcal{O}(2^m \cdot m!)$ with $m!$ being a simplification [5,77]; it is even one of only few real-world $W[3]$-complete problems [14] and, hence, particularly hard. Nevertheless, many data profiling algorithms, such as BINDER [98] or SINDY [73], are able to discover INDs in most relational datasets, as experimentally surveyed in [37]. The maintenance of INDs for dynamic datasets is possible with incremental discovery algorithms, such as S- INDD [110].

3.5 Data dependency properties

Dynamic datasets change through inserts, updates, and deletes of records and values. For this reason, we have already referenced some first incremental and dynamic data profiling algorithms that are able to maintain the knowledge about valid data dependencies over time. *Up-to-dateness* is, however, only one of many properties of data dependencies. We briefly discuss other properties and their relevance for query optimization.

Minimality All state-of-the-art data profiling algorithms mine only *minimal* (or *maximal*) data dependencies, because the sets of *all* valid dependencies are usually extremely large. Query optimizers, however, might require non-minimal (or non-maximal) dependencies, which is why these dependencies need to be inferred from discovered dependency sets. Fortunately, data dependencies follow certain axiomatizations that enable the simple generation of further dependencies. Functional dependencies, for example, follow Armstrong's axioms (reflexivity, augmentation, and transitivity [7]) that generate additional, also valid FDs from existing FDs. Inferring non-minimal (or non-maximal) dependencies from *complete* sets of minimal/maximal dependencies is particularly simple, because augmentation is sufficient to generate every possible valid dependency: Given a *complete* set of minimal data dependencies Γ, a desired valid but non-minimal UCC X or FD $X \rightarrow Y$ can easily be inferred from Γ by searching for subsets of X, similar as proposed in [97]; for ODs $\boldsymbol{X} \mapsto \boldsymbol{Y}$, we can remove common prefixes in $\boldsymbol{X}$ and $\boldsymbol{Y}$ to test for minimal valid ODs in Γ; and for INDs $\boldsymbol{X} \subseteq \boldsymbol{Y}$, Γ needs to be a complete set of maximal dependencies and we can extend it with valid superlists of $\boldsymbol{X}$. In all four cases, no transitive checks are necessary, because if dependencies derived via transitive extensions are valid, they (or some minimal/maximal version of them) are guaranteed to be included in Γ as well. So if Γ is complete and the desired dependency is true, the described lookups will retrieve at least one minimal/maximal specialization. Organizing the dependencies in prefix-trees, as proposed in [41], makes these lookups efficient. For query optimization, this means that a desired dependency might not be directly available but needs to be inferred on the fly from minimal data dependencies.

Completeness If the set of available data dependencies Γ is not a complete set of minimal/maximal dependencies, it might be necessary to pre-calculate the closure over all dependency-specific axioms, minimize the result and construct an index. This needs to be done in a preprocessing step prior to any query optimization, because the first two steps are known to be expensive, NP-complete tasks. Afterward, we can efficiently retrieve the required dependencies at query time via efficient subset and superset lookups. Despite the existence of dependency maintenance algorithms, preprocessing techniques, and on-the-fly dependency inference methods, these activities are sometimes still not affordable with regard to performance, which is why all three activities remain ongoing research areas.

Regardless of these considerations, some missing dependencies might not be inferable if the discovery and/or maintenance process is not complete. For this reason, a query optimizer will, in practice, need to deal with *incomplete* data dependency sets. Although missing out on a valid but not retrievable dependency is a lost optimization opportunity, it has no negative consequences. Incompleteness is, therefore, no issue for query optimization—the optimizer simply uses what is available with the lookup effort it can afford.

Approximation Approximate, partial and relaxed dependencies are ones that are not valid for the entire dataset [19,54]. They are produced by approximate (and usually more efficient) discovery algorithms and arise from exact dependencies on dynamic datasets if the exact dependencies are not maintained. When used for query optimization, approximate dependencies can cause incomplete and incorrect results. Hence, they are in general not usable for query optimization unless they are implemented in approximate query processing systems [90], other data structures compensate their optimization mistakes [70], or they are used only for cardinality and cost estimation optimizations (see Table 1).

Conditions Data dependencies are sometimes tied to conditions that limit their scope. A conditional dependency [15] holds on only a particular subset of tuples for which a specific condition is true. Such dependencies can be used for query optimization in the same way as unconditional dependencies if the query's selection condition (`WHERE` clause) is at least as strict as the condition of the dependencies. In summary, all query optimization techniques surveyed in this work

require a set of exact[5] data dependencies (or constraints) as input, regardless of whether these constraints are given by the schema or have been discovered from the data.

Null-semantics Relational database systems often use `null` values to indicate missing information. The comparison of `null` values, i.e., `null` = `null` evaluates to `unknown` [28], which is sometimes effectively treated as `true` or `false` in SQL. For instance, `DISTINCT`, `GROUP BY`, and `ORDER BY` statements as well as set operations evaluate `null` = `null` to `true` while `WHERE` selections and `JOIN` statements evaluate to `unknown` which, in turn, does not satisfy the predicate. So whenever a dataset may contain `null` values and we use a data dependency for query optimization, then this dependency needs to be true under the same `null` semantics as the SQL operator that is being optimized. For example, a `DISTINCT` can be removed only with a UCC that uses `null` = `null` semantics and a `JOIN`-removing IND needs to use `null` ≠ `null`. Because we can configure the `null` semantics in most data profiling algorithms, both semantics are technically available and the optimizer can pick the required ones. Unfortunately, `null` semantics are hardly considered in the surveyed literature, which is why we add this information where it is relevant in this survey. To shorten the individual discussions, we define that, if not stated otherwise, all required data dependencies use the `null` = `null` semantics, which is not only the most commonly required interpretation but also the default configuration for most dependency discovery and maintenance algorithms.

Note that both semantics `null` = `null` and `null` != `null` are practical `null` interpretations. While this practical interpretation is very useful for our objective of query optimization, a more accurate interpretation of `null` values for data dependencies is actually *no information* [8], so that the validity of a dependency depends on whether we can find a substitution for all `null` values that makes the dependency true (possible world) or we find that any substitution of all `null` values makes the dependency true (certain world) [71,72]. Albeit interesting for schema design, semantic reasoning, and many other use cases, possible and certain world interpretations are not relevant for the surveyed query optimization techniques.

4 Classification of dependency-driven query optimization techniques

This section provides an overview of all optimizations that are explained in detail in Sects. 5, 6, 7, and 8. The *optimization technique matrix* in Table 1 shows which dependencies enable a particular optimization and which query optimization activity is affected by each optimization. More specifically, the matrix classifies all dependency-driven query optimization techniques with respect to three dimensions:

i. *Dependency type:* In many query optimization scenarios, a new type of dependency is made available to the query optimizer by, for instance, recent developments in the area of dependency discovery and maintenance. The question then is how these data dependencies can be utilized. For this reason, we chose the dependency type—UCC, FD, OD, or IND—as the main classification dimension.
ii. *Relational operator:* Every optimization targets a particular query operator or set of query operators. For this reason, we use the operators of the relational algebra as second classification criterion. In practice, most optimizations target only one operator. For those optimizations that affect multiple operators, we show the most relevant operator, which is usually the most expensive one.
iii. *Query optimization activity:* Every optimization can be assigned to the query optimization activity that is mainly affected by it. Hence, we use the three activities that we discussed in Sect. 2—cost-independent transformation, cost-based transformation, and cardinality and cost estimation—as our third classification dimension.

Note that for different systems, techniques might be classified differently w.r.t. their query optimization activity depending on the implementation and degree of sophistication of the query optimizer. For this classification, we place the techniques into the most likely categories. We also again emphasize that some optimizations affect not only the structure of the query plan (e.g., operator reordering) or the choice of operator implementations (e.g., sequential vs. index scan) but also the behavior of operators at runtime. Scan operations, for instance, might abort early in certain cases. Instead of attributing such optimizations to the execution phase, we decided to attribute them to query optimization, because, ultimately, the optimizer prepares for such behavior, decides on the query plan, and instructs the physical operators to act accordingly.

Counting the entries, Table 1 shows that more optimizations exist for UCCs (17), FDs (14), and ODs (18) than for INDs (10): INDs are dependencies between unordered sets of values (or value combinations); their known optimizations, therefore, support only join and set operations. Most optimizations influence *cost-independent transformations* (29), followed by *cost-based transformations* (18) and lastly *cardinality and cost estimation* (9).

We also identify a fourth dimension, in which the presented transformation-based optimization techniques could

[5] Optimizations applied during cardinality and cost estimation can also utilize approximate dependencies.

Table 1 Possible optimizations categorized by (i) the examined data dependencies, (ii) the area of application, i.e., operators of the relational algebra, and where possible (iii) the query optimization activity that is affected by the optimization, which is indicated by background color and symbol: cost-based plan transformations (†), cost-independent transformations/rewriting (∗), cardinality and cost estimation (‡). For optimizations that have not been scientific section where this optimization is explained

Application area	Unique Column Combinations (Sec. 5)	Functional Dependencies (Sec. 6)	Order Dependencies (Sec. 7)	Inclusion Dependencies (Sec. 8)
Join	• Spurious-free back-joins [129] ∗ • Semijoin tranformation [89] ∗ • Pipeline with grouping [30, 126] † • Invisible join [2] †	• Simplification / avoidance [65] ∗ • Complexity reduction (Sec. 6.2) ∗ • Self-join avoidance [6] ∗ • Plan generation [38] †	• Join avoidance [117, 119] ∗ • Pipeline with grouping [23, 49] † • Avoid sort for sort-merge-joins [42, 102, 112] † • Attribute substitution (Sec. 7.2) † • Pipeline index scan with join [49] †	• Join elimination [24, 64] ∗ • Substitute relations [33] † • Avoid semijoin reductions (Sec. 8.1) † • Accurate cardinalities [56] ‡
Selection	• Early abort (Sec. 5.6) ∗ • Accurate cardinalities (Sec. 5.6) ‡	• Early abort (Sec. 6.3) ∗ • Substitute attributes [24, 68] † • Estimations without independency assumption [25, 58, 108] ‡	• Use binary search [102] †	
Grouping & Aggregate functions	• Grouping is redundant [23] ∗ • Accurate cardinalities (Sec. 5.6) ‡	• Reduce attributes [17, 116] ∗	• Simplify `MIN`, `MAX`, `MEDIAN` [95, 102] ∗ • Sort-based grouping [102, 112, 117, 127, 128] †	
Projection & Distinctness	• Avoid `DISTINCT` [101, 102, 103] ∗	• Distinctness: see grouping [123] ∗ • Simplification [29] ∗ • Estimate projections [44] ‡	• Distinctness: See grouping †	
Sorting	• Reduce attributes (Sec. 5.6) ∗ • Unstable sorting (Sec. 5.6) ∗	• Reduce attributes [24, 112, 116] ∗	• Reduce attributes [115, 116, 117] ∗ • Avoid sort [49, 102] ∗ • `ORDER BY` with index [117] ∗ • Main-memory sorts [117] † • Substitute attributes [102] † • Accurate estimates (Sec. 7.4) ‡	
Set Operations	• `EXCEPT` to `EXCEPT ALL` [102] ∗ • `INTERSECT` to `INTERSECT ALL` [60, 101] ∗ • `INTERSECT` to join [101, 102] † • Accurate cardinalities (Sec. 5.6) ‡		• Order optimizations [102] †	• Simplify `UNION` (Sec. 8.2) ∗ • Simplify `INTERSECT` (Sec. 8.2) ∗ • Eliminate `EXCEPT` (Sec. 8.2) ∗ • Accurate cardinalities (Sec. 8.2) ‡
Other	• Subquery to join [101, 102] ∗ • Subquery sort avoidance [110] †	• Scalar subqueries [29] • Table decomposition rewrites [45]	• Correlated subqueries [102] ‡ • Sparse over dense indexes [34]	• Query folding [33, 51, 57] ∗ • Eliminate correlated subqueries in `EXISTS` [85] (Sec. 8.2) ∗

be categorized: the *optimization method* with which the improvement is achieved. This dimension is not visualized in the table, because it does not apply to all depicted techniques.

- *Simplification:* The task that the operator needs to fulfill is simplified, e.g., by removing attributes from a `GROUP BY` list or by omitting the sort phase of a sort-merge join. Avoiding the operator execution altogether is an extreme case of simplification. Examples can be found in [2,6,17,23,24,29,49,65,84,88,94,100–102,109,114–116,118,122,128].
- *Algorithm choice:* Oftentimes, a specific implementation of a query operator can be selected from different alternatives. The available data dependencies can guide this decision. For instance, a binary search is superior to a sequential scan if a dependency indicates that the data is sorted. Examples are found in [42,101,111,116,126,127].
- *Substitution:* Certain data dependencies indicate that an operator can, instead of processing an attribute A or a relation R, process an attribute B or a relation S with the same result. This might be beneficial if B offers superior properties, such as a more compact data-type or being indexed. Examples are presented in [24,33,33,51,57,68,101].
- *Pipelining:* Some dependencies provide additional guarantees that enable pipelining between operators in cases where it would usually be infeasible. Examples are found in [23,30,49,125].

In conclusion, Table 1 provides an overview of all optimization techniques presented in this survey and categorizes them to show which optimizations are enabled by each dependency type.

5 Unique column combinations

In the following subsections, we present various query optimization techniques that are enabled by the existence of UCCs. (Primary) keys are by definition UCCs, which, vice versa, serve as key candidates. For that reason, all presented optimizations can be applied analogously given either key constraints on the schema or UCCs discovered from a relational instance. Apart from query optimization and keys, uniqueness is often used for data integration, indexing, and anomaly detection.

5.1 UCCs and joins

If an SQL query joins two relational instances that both have a UCC, then the resulting relation contains a (not necessarily minimal) UCC, which is the concatenation of both UCCs. Furthermore, if the join attributes are UCCs in both relational instances, they are also unique for the join result. These two properties can be used to track unique column combinations across joins, so that they may be used for the optimization of downstream operators and query plans.

Most UCC-based join optimizations require the existence of uniqueness on at least one of the input relations' join attributes. For example, Dayal shows that aggregations and asymmetrically implemented joins can be pipelined if the grouping attributes contain the primary key, i.e., a UCC of the outer join relation [30]. Such join implementations group the result by tuples of the outer relation's join attributes. Consider the query:

```
SELECT R.A, SUM(S.B) FROM R, S
    WHERE R.A = S.A
    GROUP BY R.A
```

If $R.A$ is a UCC and chosen as outer relation, the join's results can be streamed directly to the aggregate function `SUM(`$S.B$`)`, because the absence of duplicate values guarantees that the records are already grouped by $R.A$. Yan states that this applies to candidate keys and UCCs if these are the outer relation's join attributes [125].

An inner join of two relations R and S on the join attributes X can be executed by potentially more efficient semijoin[6] strategies [88] if X is a UCC on S, and S does not need to supply columns to the result. Intuitively, the UCC ensures that any tuple $r_k \in R$ from the outer relation can match only one single tuple $s_i \in S$ of the inner relation [23], i.e., $r_k[X] = s_i[X]$ is unique with respect to $r_k[X]$. This is true, because by the definition of UCCs no second tuple $s_j \in S, i \neq j$ exists for which $s_i[X] = s_j[X]$. If X was not a UCC in S, the inner join could replicate rows in R's instance and, hence, produce a different result than the semijoin. Rewriting inner joins as semijoins is useful, for example, in distributed query optimization, to send less data over the network [11,87]. For nested loop join strategies, a UCC on the inner join loop's attribute enables aborting the inner loop early and continuing the outer loop as soon as the first match is found.

Yang and Larson illustrated another use case for UCCs when working with derived relations and so-called *back-joins* [128]: Let us assume a derived relation E_1 from R holds all the required tuples to answer a query Q, but misses some attribute A. With a back-join,[7] the attribute A can be obtained from another relation E_2 that was derived from R as well. Such a back-join can produce *spurious* tuples that contain values that originate from different tuples of the base relation R. Performing this back-join on a UCC with attributes X, however, prevents the generation of such spurious tuples, because, by the definition of UCCs, if tuples agree on X, they also agree on all other attributes.

Abadi et al. [2] introduce so-called *invisible joins* for star schemas in column-oriented database engines. This technique improves the performance of foreign-key/primary-key joins by, among others, transforming such joins into predicates on fact table columns. Given the required UCC, i.e., a primary key, the optimizer can choose this special execution technique to improve the performance of the join. Because the detailed description of this join technique is beyond the scope of this paper, we refer the interested reader to [2].

5.2 UCCs and grouping and aggregation

When grouping on a UCC, it is by the definition of UCCs obvious that the maximum group size is 1. For this reason, the entire grouping step is superfluous to calculate aggregations on these groups: The data is implicitly grouped already [23]. Hence, both sort- and hash-based aggregation implementations can omit the grouping phase, i.e., sorting or hashing, if they are aware of the UCC.

Because UCCs are essentially special forms of functional dependencies, we list further UCC-based optimizations for grouping and aggregation operations with the FD-based optimizations in Sect. 6.1.

5.3 UCCs and distinctness

SQL statements containing the `DISTINCT` keyword are common in practice [101]. Being able to optimize duplicate eliminations is, for this reason, very important.

Paulley and Larsson explain that query results in certain combinations with UCCs cannot contain duplicate tuples and, hence, the execution of a `DISTINCT` operation is unnecessary [100,101]. Given a UCC X on a relation R, they show: If either (i) all attributes of X are part of the query's projection list, or (ii) a subset $Y \subset X$ is contained in the projection list and the other attributes $X \setminus Y$ are selected via equality predicates, the query result is unique and the `DISTINCT` operation can be removed. Pirahesh et al. mention a similar technique, but not as detailed as Paulley and Larsson and without explicitly taking `null` values into account [102]. Since distinctness is usually ensured by costly sort- or hash-based approaches,

[6] The semijoin $R \ltimes S$ selects tuples of R that would match a tuple of S if joined, i.e., $R \ltimes S \Leftrightarrow R \bowtie_X \pi_X(S)$.

[7] A *back-join* is used when a derived relation holds all necessary tuples to answer a query but requires additional attributes from other relations.

the removal of redundant distinctness keywords is a substantial optimization.

5.4 UCCs and subqueries

Paulley and Larsson describe how correlated subqueries can be transformed into ordinary join queries [100,101]. Optimizers can then apply all the rules and optimization techniques that are relevant to joins, e.g., choosing a particular join algorithm with better performance for that particular case or adjusting the join order to find a more efficient query plan [102]. While these techniques have been proposed earlier, e.g., by Kim [67], Ganski et al. [43], and Pirahesh et al. [102], Paulley and Larsson explicitly consider duplicate entries and `null` values.

To illustrate the idea,

```
SELECT R.A, R.B FROM R WHERE EXISTS
    (SELECT * FROM S WHERE S.A = R.A)
```

can safely be rewritten to

```
SELECT R.A, R.B FROM R, S
WHERE S.A = R.A
```

iff $S.A$ is a UCC; otherwise, the transformed version of the query might result in more results than the original version. In case a rewrite was possible, also the semijoin strategies might apply to further optimize the query [93]. The UCC ensures that the subquery cannot provide more than one matching tuple, which enables the transformation. The rewrite is also possible for a multi-attribute UCC X if the non-join attributes are selected via equality predicates (see Sect. 5.3).

Sometimes, correlated subqueries cannot be unnested. In such cases, the query engine can cache results obtained from subquery evaluations to reuse these results for evaluations with repeated, i.e., the same referenced values. With the cached results, redundant subquery evaluations can be avoided. Selinger et al. develop this idea one step further and propose to first sort the outer relation by the referenced column and then execute the subqueries [109]; in this way, the query engine needs to cache only one, i.e., the last subquery result. However, if the referenced column is a UCC, both the caching and the sorting are redundant and should not be applied, because repeated values do not exist [109].

5.5 UCCs and set operations

While the relational algebra is based on set semantics, SQL generally uses bag semantics. SQL's set operations, such as `INTERSECT, EXCEPT, and UNION`, however, provide set semantics, unless disabled by the `ALL` keyword. By the set definition, sets cannot contain duplicate values, but relations in database systems typically allow them. Therefore, implementations of set operations need to ensure that duplicates are removed before providing the final result.

Because uniqueness plays a central role for set operations, it is apparent that UCCs can be used to optimize them. For the following examples, we assume two tables R and S with a UCC on $R.A$. Paulley and Larsson note that "the semantics of `INTERSECT` and `INTERSECT ALL` are equivalent if at least one of the involved tables cannot produce duplicate rows" [100,101]. If this pre-condition is guaranteed by a UCC, the costly duplicate elimination of `INTERSECT` can be avoided by rewriting it as an `INTERSECT ALL`. The rational is as follows: According to the SQL standard [60, p. 202], the result of an `INTERSECT ALL` statement on the tables R, S contains $min(m, n)$ instances of a duplicate tuple t, where m and n are the numbers of occurrences of t in R and S. Hence, the UCC guarantees either m or n to be 1 which, in turn, guarantees only a single occurrence of t in the result. Thus, no duplicate elimination is necessary.

Similarly, R `EXCEPT` S can be rewritten to R `EXCEPT ALL` S, simply because a difference operation cannot introduce duplicates if these are not already present in R [101]. Their absence is guaranteed by the UCC on $R.A$.

Further, some database systems, such as MySQL, do not support `INTERSECT` statements [89]. Instead, the documentations of these systems often suggested to express the semantics of `INTERSECT` (manually) with an inner join:

```
SELECT DISTINCT(R.A) FROM R, S
  WHERE R.A = S.A
  OR (R.A IS NULL AND S.A IS NULL)
```

If a UCC exists not only on $R.A$ but also on $S.A$, the `DISTINCT` can be removed and, hence, `INTERSECT` can be formulated as a join. The correct handling of `null` values for such rewrites can be achieved as demonstrated above [100,101]. The main advantage of these transformations is that they help to avoid costly duplicate removals [62] when these are superfluous.

5.6 Further optimization opportunities with UCCs

Apart from the main concepts described above, unique column combinations enable some further potential optimizations.

Selection If a `SELECT` clause defines an equality predicate for all attributes of a UCC X, the query can be aborted after the first matching tuple is identified, because the UCC guarantees that no further matches can be found. While such an operation on primary-key columns would be handled usually by an index lookup, not all UCCs are necessarily indexed. In real-world ERP database systems, for example, there are many equality predicates on such attributes [16].

Cardinality and cost estimation Uniqueness information can be of use for cardinality and cost estimation of various operators. For example, selections with equality predicates on a UCC can have at maximum one resulting tuple. Non-equal checks result in either $|R|$ or $|R| - 1$ results and, for equality joins, there is at most a single matching tuple, cf. Section 5.1. Furthermore, uniqueness information can be utilized to determine the number of groups for `GROUP BY` operations and to estimate lower bounds for set operations.

Sorting The presence of UCCs allows to remove all attributes following that UCC in the attribute list of an `ORDER BY` clause. For instance, under the UCC X, the attributes Y can be removed from `ORDER BY X, Y`, because sorting by Y would affect only the order of tuples that agree in X. The UCC guarantees that such tuples do not exist.

Stable sort algorithms keep equal elements in their original order. This stability guarantee is usually exchanged for higher runtime complexity or elevated memory consumption. However, UCCs ensure that no duplicate elements exist so that stable sort algorithms are not needed. So if the execution engine offers multiple physical sort operator implementations, UCCs can be used to find the most suitable implementation during optimization.

Embedded unique constraints Embedded unique constraints allow for expressing the existence of UCCs on fragments of incomplete data, i.e., data containing `null` values. Wei et al. demonstrate how such embedded unique constraints can be utilized for query optimization, e.g., to improve the efficiency of joins or scans in queries that handle incomplete data [123].

6 Functional dependencies

The use cases for functional dependencies are manifold. They were initially used to normalize database schemata [26], but meanwhile also aid data cleansing, data integration, and data translation tasks. In the remainder of this section, we focus on their utilization for query optimization.

6.1 FDs and grouping

Grouping operations can be simplified through query rewriting if functional dependencies are present. If the functional dependency $B, C \rightarrow A$ holds and a group operation on the combination B, C, A is to be executed, the grouping attribute A is unnecessary, because all elements that fall into the same group for B, C necessarily also have the same value for A. Thus, the grouping attribute A can simply be removed and it is sufficient to group only on B, C [115]. The same rule applies if the determinant B, C is (partly) filtered in the `WHERE` clause with an equals condition. For example, the `GROUP BY` clause in the query

```
SELECT C, SUM(D) FROM R WHERE B = 17
 GROUP BY C, A
```

can be rewritten to `GROUP BY C`. Since 1999, the SQL standard explicitly allows selecting columns that are not part of the `GROUP BY` clause if they are functionally dependent on grouping columns [59]. Date and Darwen [29] mentioned this problem earlier in their work. The presented technique can in fact also be applied to several TPC-H queries [17]. For example, TPC-H Query 3 contains a grouping operation on l_orderkey, o_orderdate, o_shippriority. Because l_orderkey is a key, o_orderkey → o_orderdate, o_shippriority holds. The grouping statement in Q3 can, therefore, be simplified to group on l_orderkey only.

As the enforcement of distinctness is a special case of grouping [23], the above-described method can also be applied to SQL `DISTINCT` clauses: `DISTINCT X, A` reduces to `DISTINCT X` if $X \rightarrow A$. In the context of duplicate elimination, Weddell also explained how duplicate eliminating projections (in the sense of the relational algebra) can be avoided with known functional dependencies [122].

Date and Darwen also partly describe an optimization if a `DISTINCT` is applied after a relation has been filtered with an equals predicate [29]: Consider $A \rightarrow B$ to hold and the query

```
SELECT DISTINCT B FROM R WHERE A = 4
```

In this case, the `DISTINCT` is necessary, because the selection can possibly yield multiple results. However, above's FD ensures that all resulting rows will have the same value in B, and a costly duplicate elimination can be avoided by just returning the first row.

6.2 FDs and joins

The optimization potential of FDs for joins might appear limited at first glance, because joins test for value correspondences *across* potentially different relations, while FDs test for value dependencies *within* one relation. There are, however, a few interesting applications related to joins in query rewriting and join ordering.

First, Eich et al. examine optimizations for eager aggregation that were initially presented by Yan and Larson [126], i.e., group-by operations that are pushed below joins. The authors show how functional dependencies can be used to prune join trees during query plan generation and, hence, speed up the plan generation process by orders of magnitude [38]. For example, consider the question whether a join subtree T_1 that is more expensive than a join subtree T_2 can be pruned during plan generation. In T_1, attribute A of a relation R is (eagerly) grouped before R is joined with S. In T_2, $R \bowtie S$ is executed before attributes A, B (B is an attribute of S) are grouped. Now, T_1 should only be pruned if

at least the same set of FDs holds after executing T_2 as after T_1. The intuition is as follows: The execution of a group-by influences which functional dependencies hold in the subsequent intermediate results; without grouping, the FDs would be equivalent for all join trees. If the FDs that are necessary to fulfill the specified query do not hold in the end, a final group-by would have to be added to ensure the correct result. This final group-by operation could introduce additional costs that, when taking the aforementioned pruning criteria into account, renders the plan suboptimal. Note that *early partial aggregation* could even reduce the overall costs [74].

Kambayashi and Yoshikawa [65] apply FDs to simplify queries that involve joins. Assume a natural join $R \bowtie S$ that joins the relations R and S on the attributes Z with $S[Z] \subseteq R[Z]$ and vice versa. If the FD $X \rightarrow A$ holds and $X \subseteq Z$, then the join attributes Z can be reduced to $Z \setminus A$, because matching values in X guarantee matching values in A. The same reduction also applies to arbitrary self-joins $R \bowtie_Z R$ on same attributes Z, if A, i.e., the removed attribute, does not contain `null` values; such `null` values would prevent certain result pairs that match on $Z \setminus A$ and would, therefore, arise after the reduction. The described rewriting technique improves the join execution, because fewer join attributes can reduce both computation time and memory consumption.

In combination with selections that are executed prior to the join, FDs can be used for optimizing these joins. Again, consider the FD $X \rightarrow A$ to hold on a relation R and a query that filters R with an equality predicate on X and then joins R with a relation S on A. The FD ensures that all tuples in R that remain after the filter operation have the same value in A. Thus, all tuples from S that match the first filtered tuple in R also match all other remaining tuples in R. This insight can substantially reduce the complexity of the join operation: After joining the first filtered tuple of R to its matching tuples in S, we can reuse the same set of S tuples as matching tuples for all other filtered R tuples by simply duplicating these tuples. In other words, $\sigma_X(R) \bowtie_A S$ with $X \rightarrow A$ can be written as $\sigma_X(R) \times (S \ltimes_A \pi_A(\sigma_X(R)))$, which is, we calculate the cross-product of the filtered R tuples with the S tuples filtered by the right-hand-side value A defined by the X value. For performance reasons, the result of $\sigma_X(R)$ can be cached to execute the filter only once. Also note that, under bag semantics, the *limit* operator must be applied to obtain only the first matching R tuple for the semijoin with S.

Finally, Abiteboul, Hull, and Vianu show that an equality self-join can be avoided in cases, where the join attribute functionally determines all other attributes in the distinct projection of the query [6]. Consider a relational instance r over R with attributes X, Y and the FD $X \rightarrow Y$ to hold. Any self-join of r on X is semantically superfluous and can, therefore, be avoided, because $\forall s, t \in r$: iff $s[X, Y]$ and $t[X, Y']$ the aforementioned FD ensures $Y = Y'$. If R contains the attributes X, Y, Z, but $X \nrightarrow Z$, the self-join on X is still unnecessary if only X and Y are in the projection of the query. In both cases, the self-join adds a redundant column. More specifically, $\pi_{XY}(R) \bowtie_X \pi_{XY}(R)$ with $X \rightarrow Y$ can be rewritten to $\pi_{XYXY}(R)$. Admittedly, the join also (inefficiently) removes `null`-valued records if X contains `null` values and it increases the cardinality of the relation if X is not distinct, i.e., it is not a UCC.

6.3 FDs and selection

The use of FDs with selections offers powerful simplifications. Some of the below mentioned optimizations might not be efficiently realizable with standard implementations of physical operators but require alternative implementations.

For an FD $X \rightarrow A$ on a relational instance r over R, consider a query that filters R with two *equality* predicates on X and A. If the filter on X is evaluated first, it then suffices to check only a single element of A, because the FD guarantees that all other tuples (that matched X) have the same value in A: The overall result is empty, if the A value differs from its filter value; otherwise, it is non-empty and no further A value needs to be checked. In an extreme case, a tuple-at-a-time execution model can abort the selection process after checking the first tuple of a potentially large table, if its X value matches but its A value differs.

Furthermore, selections can be shifted to another attribute that is less expensive to process, for instance because it is indexed or of a less complex data-type, such as *int* instead of *string*. This technique is also called *predicate introduction*[8] and was originally intended to be used with check constraints by Cheng et al. [24]. Kimura et al. [68] explicitly mention the application of FDs in such scenarios. Given the FD $A \rightarrow B$ for a relational instance r over schema R and a query with the selection $\sigma_{A=v_A}$, the system could determine the first tuple $s \in r$ with $s[A] = v_A$ (for example, with a partial scan) and, then, find the value v_B as $s[B] = v_B$ that corresponds to the selection value v_A. Now, the system replaces $\sigma_{A=v_A}$ with the cheaper selection $\sigma_{B=v_B}$. This new query serves only as a pre-filter, because not all tuples that match on $\sigma_{B=v_B}$ also match on $\sigma_{A=v_A}$. So in the end, the (potentially small) result-set of the adapted query must be re-evaluated on $\sigma_{A=v_A}$. This optimization technique requires highly selective B-predicates and relatively large amounts of data to result in performance advantages. Furthermore, v_B must not be `null`, because `null` comparisons in selections always resolve to effectively false. The evaluation of $\sigma_{A=v_A}$ can be omitted, if the reverse FD $B \rightarrow A$ is also true.

[8] Predicate introduction can be applied not only if FDs are present, but also if correlations [85] or algebraic constraints [18] exist which are not considered in this work.

Scalar subqueries are required to return either no or exactly one row. Many systems throw an exception otherwise [95, 106]. Date and Darwen [29] mention a possible optimization regarding such queries in combination with FDs. Consider the query

```
SELECT * FROM R WHERE R.B >
(SELECT B FROM S WHERE S.A = 4)
```

If the FD $A \rightarrow B$ holds, the equality predicate on A in combination with the FD ensures that B has the same value for all rows. Thus, the result could be computed even though the subquery returns more than a single row. However, to the best of our knowledge, this technique is not implemented in any commercial database system.

6.4 FDs and sorting

FDs can be used to simplify operations that introduce order in the involved relations [24,115]. We can, in particular, reduce the attribute lists in `ORDER BY` clauses with the use of known FDs, which was first shown by Simmen et al. [111]. For example, the clause `ORDER BY X, A` can be reduced to `ORDER BY X` if the FD $X \rightarrow A$ holds, because for a certain value of X, there is only one value in A.

6.5 Further optimization opportunities

Apart from the optimizations for the operations described above, functional dependencies also support various further query-related tasks. Analogously to embedded unique constraints, embedded functional dependencies [124] can be used for query optimization under incomplete data.

Cardinality estimation Query plan optimizers or other database components that estimate the cardinality of database operators often assume value independence for the different attributes and uniform value distributions[9] [75,76,109]. As the name implies, functional dependencies indicate the contrary. Known FDs can, therefore, improve cardinality estimations [25,58,107] and lead to better query plans. For instance, the cardinality of the conjunction of two predicates $\sigma_{A=v_A}$ and $\sigma_{B=v_B}$ is usually estimated as the product of their individual selectivities, which is $\frac{1}{|A|} \cdot \frac{1}{|B|}$. If, however, $A \rightarrow B$ is true, then only $\frac{1}{|A|}$ is the appropriate cardinality estimate, because v_A *always* co-occurs with v_B due to the FD.

Furthermore, given $A \rightarrow B$, we know that $|B| \leq |A|$ and, given both $A \rightarrow B$ and $B \rightarrow A$, we know that $|B| = |A|$. In this way, we can use FDs to let cardinality information (or estimations) propagate from one attribute to another.

[9] Not all systems assume uniformity for all values. For example, PostgreSQL assumes uniform value distributions only for all values *inside each histogram bucket* [120].

Finally, Gelenbe et al. utilize functional dependencies to estimate the size of projections (in the duplicate-removing semantics of relational algebra) [44]. They use the above-described guarantee of $|B| \leq |A|$ in the presence of $A \rightarrow B$. Following from that, $|\Pi_{A,B}| = |\Pi_A|$.

Rewrite after decomposition Gianella et al. [45] demonstrate how FDs, can be used for query rewriting in combination with horizontal table partitioning. The idea is to split a relational table horizontally into two partitions, one in which an FD is actually true and one that contains all the violations to the FD. All queries to the partitioned table, then, also need to be split, i.e., rewritten to read from both partitions. The subquery that reads from the partition, in which the FD holds true, can use all the optimizations that we introduced before; the other subquery is executed just regularly. DBMSs that implicitly partition the stored data, such as Hyper [66] or Hyrise [36], could use this technique and determine FDs on a per partition basis. Please note that, in practice, partitioning criteria should be based not only on FDs but on other aspects, e.g., availability or performance, too [91].

7 Order dependencies

Order information serves a variety of tasks, such as optimizing the physical storage of records (e.g., for run length encoding in columnar data stores [1]) and improving readability of query results by ordering them.

Information about order and so-called interesting orders (first introduced by Selinger et al. [109]) are a crucial part for query rewriting and query plan optimization; they can, in particular, further be utilized during the actual operator execution and for cost estimation. ODs present an important opportunity to make the most use of order-based optimization techniques, because they help to derive additional order information from knowledge about currently available orders. As a result, knowing that $\boldsymbol{X} \overset{\leq}{\mapsto} \boldsymbol{Y}$ holds and that $\boldsymbol{X}$ is ordered opens up opportunities to utilize the order information about both attributes $\boldsymbol{X}$ and $\boldsymbol{Y}$ during query optimization. Operations that generate ordered data can be explicitly pushed down (closer to the beginning of the query plan, cf. *sort-ahead* [111]) to enable order-based optimizations for the subsequent operators. The more order information is available the wider is the range of potential plan optimizations.

We compile information on how exactly order and order dependencies can be utilized for query optimization and to improve the execution of individual operators in the remainder of this section. Some optimization ideas presented in this section are similar to ideas presented for FDs, showing interesting relationships between FDs and ODs.

7.1 ODs and sorting

The purpose of sort operations, explicitly expressed by `ORDER BY` statements in SQL, is to produce order. Hence, it is not surprising that this operation offers several potential optimizations regarding ODs. First, the number of attributes in the order clause can be reduced in the presence of interesting orders [111] and, hence, order dependencies [114–116]. A reduced number of sorting attributes leads to fewer sort operations, which can potentially decrease the execution time. Additionally, reducing the number of attributes in the order clause increases the possibility that this operation can be solved with an index.

If $\boldsymbol{X} \overset{\leq}{\mapsto} \boldsymbol{Y}$ holds on a relation R, the clause `ORDER BY X, Y` can be reduced to `ORDER BY X`, because an ordered $\boldsymbol{X}$ ensures an ordered $\boldsymbol{Y}$.[10] With the aforementioned OD, $\boldsymbol{Y}$ can furthermore be removed from both clauses `ORDER BY W, X, Y` and `ORDER BY W, Y, X` [115]. The latter might not be intuitively clear, but if we replace $\boldsymbol{Y}$ with $\boldsymbol{X}$, which is possible because $\boldsymbol{X}$ imposes an order on $\boldsymbol{Y}$, the resulting `ORDER BY W, X, X` still guarantees that the result is ordered by $\boldsymbol{Y}$ following from the definition of ODs; obviously, one of the consecutive $\boldsymbol{X}$ could be removed. Sorting $\boldsymbol{Y}$ can be avoided altogether under above's OD and if other previously executed operations, such as sort-merge joins, sort-based aggregates, or index scans internally order $\boldsymbol{X}$ (or $\boldsymbol{Y}$) [49,101]. In the case of $\boldsymbol{X} \overset{\leq}{\mapsto} \boldsymbol{Y}$, `ORDER BY Y, X` can be reduced to `ORDER BY Y`, because $\boldsymbol{X}$ cannot break ties in $\boldsymbol{Y}$.

Furthermore, ODs can be utilized to substitute sorting attributes [101]. Imagine the attributes A (integer), B (string), and C (integer) and the OD $A \overset{\leq}{\mapsto} B$ to hold. Hence, a statement `ORDER BY B, C` could be replaced by `ORDER BY A, C`, because an ordered A implies B to be ordered. Substitutions are beneficial if they decrease the cost of the sorting operation: in our example the costly string-sort was replaced by the cheaper integer-sort. Alternatively, the attribute to be substituted could be replaced by an indexed attribute that allows efficient ordered retrieval.

Szlichta et al. [116] also mention that ODs and near-sortedness can be combined to execute small, on-the-fly main-memory sorts instead of external sorts.

7.2 ODs and joins

Order information can be used to simplify joins in the plan optimization phase, to improve the join operator execution phase, and to better estimate join result-set cardinalities.

Sort-merge join algorithms, as the name implies, are split into two phases: (i) an initial sort phase that provides ordered input for the (ii) merge phase, where the actual join takes place by merging two ordered lists. It is evident that sort-merge joins can benefit from already sorted inputs, by omitting the initial sort phase [42,101,111]. If $\boldsymbol{X} \overset{\leq}{\mapsto} \boldsymbol{Y}$ holds and $\boldsymbol{X}$ is ordered, sort-merge joins on either $\boldsymbol{X}$ or $\boldsymbol{Y}$ could both omit the sort phase on their corresponding relation, and could be a preferred choice.

Another opportunity is the pipelining of grouping and join operators [23,49]. In [23], the authors present a cost model-based approach to push grouping operators past joins to find more efficient query plans. Some grouping operator implementations sort the data to create groups. If $\boldsymbol{X} \overset{\leq}{\mapsto} \boldsymbol{Y}$ holds and X constitutes the grouping attributes, this technique is promising if the later join is either on $\boldsymbol{X}$ or on $\boldsymbol{Y}$.

Szlichta et al. further describe a data warehouse scenario in which ODs can be used for query rewriting to avoid expensive joins between fact and dimension tables [116,118]. Consider the SQL query

```
SELECT ... FROM sales s, date d WHERE
  s.sold_date_sk = d.date_sk AND
  d.date BETWEEN '20201104' AND
  '20201204'
```

The insight that there is usually an order dependency in the date dimension table between a well-constructed surrogate key and natural date values, i.e., $\mathsf{date}_{\mathsf{sk}} \overset{\leq}{\mapsto} \mathsf{date}$, enables the rewriting of joins on dates, because date attributes of the fact and dimension table can be replaced with probably cheaper local predicates. The intuition is as follows: Two simple probes find the minimum and maximum[11] surrogate keys for the corresponding dates in the dimension table. These surrogate keys are used as local predicates on the fact table:

```
SELECT ... FROM sales s,
(SELECT MIN(date_sk) min_d FROM date
    WHERE date >= '20201104') d1,
(SELECT MAX(date_sk) max_d FROM date
    WHERE date <= '20201204') d2
WHERE s.sold_date_sk BETWEEN d1.min_d
    AND d2.max_d
```

The combination of dimension table probing and a local fact table predicate effectively replaces the join. This is possible only because the OD $\mathsf{date}_{\mathsf{sk}} \overset{\leq}{\mapsto} \mathsf{date}$ ensures that the correct surrogate keys are picked for the local predicate via `MIN` and `MAX`.

Order dependencies can also be used to optimize certain theta joins via query rewriting if combined with UCCs. For

[10] Note that $\boldsymbol{X} \overset{\leq}{\mapsto} \boldsymbol{Y}$ is not sufficient for this optimization, because the comparator $\prec$ imposes only a partial order. Therefore, $\boldsymbol{Y}$ would still be needed to impose a clear order on tuples with the same $\boldsymbol{X}$ values.

[11] This optimization benefits from an engine supporting the simultaneous computation of `MIN` and `MAX` with a single scan.

this, assume that the OD $\boldsymbol{X} \overset{\prec}{\mapsto} \boldsymbol{Y}$ holds on a relation R and that X is a UCC. Then, consider the example SQL query

```
SELECT * FROM R Rl, R Rr WHERE
    Rl.X < Rr.X
```

which performs a theta self-join on $\boldsymbol{X}$. Due to the OD, we can replace the term `Rl.X < Rr.X` with the OD's right-hand-side attribute $\boldsymbol{Y}$: `Rl.Y < Rr.Y`. The UCC ensures that R does not contain tuples with the same value in X and potentially unordered values in Y. Scenarios with such a combination of UCC and OD occur naturally if attributes $\boldsymbol{X}$ correlate with, for example, an incremental surrogate key attribute $\boldsymbol{Y}$. This technique improves the execution of the query if $\boldsymbol{Y}$ has, e.g., an index, a smaller domain or a more join-friendly data-type. The same optimization also applies for the other inequality predicates $\leq$, $>$, and $\geq$ in the query as well as for the OD $\boldsymbol{X} \overset{\succ}{\mapsto} \boldsymbol{Y}$.

7.3 ODs and grouping

Similar to joins, there are two approaches for grouping data and aggregating data, respectively: hashing and sorting [86]. Both operators profit from pre-ordered inputs. With the knowledge of ODs, additional inputs are known to be ordered. Hence, optimizers can choose sort-based operators in more cases. Sort-based implementations can benefit from pre-ordered inputs, because the sorting step of the operator can—exactly as for joins—be (partially) omitted [101,111,116,126,127]. Hash-based implementations, on the other hand, can exploit ordered inputs to minimize the number of hash calculations, i.e., they simply avoid repeated re-hashing of the same value by recycling the previous hash until the next new value in the ordered input is read. The positive effect of ordered data on the performance of hash-based algorithms was investigated by Memarzia et al. [83].

7.4 Further optimization opportunities with ODs

Apart from the main concepts described above, order dependencies offer various further optimizations.

Selection Selections can benefit from ODs in the following way: If $A \overset{\prec}{\mapsto} B$ holds and their relation is ordered by A, then table scans on data that is held in-memory can be replaced with binary searches not only for selections on A but also for selections on B [101], which reduces the complexity from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$. Range predicates can analogously use a binary search to determine the starting element of the range.

Aggregate functions Some aggregation functions, such as `MIN`, `MAX`, and `MEDIAN`, allow obvious optimizations on ordered data [101]: If $A \overset{\prec}{\mapsto} B$ holds and their relation is ordered by A, it is not necessary to check all elements of the relation to determine the result of `MIN` on either A or B. Instead, it is sufficient to return the first element of the respective attribute [94, p. 566]. This shortcut works analogously for `MAX` and `MEDIAN`, but instead of the first, the last and middle element, respectively, are selected.

Leveraging indexes Indexes can be used to retrieve tuples in order, which is useful to pipeline index operations with other operators that rely on ordered data. In this way, we can, for instance, pipeline sort-merge joins with index scans [49]. Also, a clustered index on salary along with the ODs salary $\overset{\prec}{\mapsto}$ taxes and salary $\overset{\prec}{\mapsto}$ percent allow a query that contains an `ORDER BY taxes, percent` clause to be evaluated directly by retrieving the data from the index, without an additional sort operation [116]. Another index optimization based on ODs was proposed by Dong et al., who found that with the knowledge of ODs sparse indexes can be chosen over dense ones to save space [34].

Generating distinctness According to Chaudhuri et al. [23], distinctness is a special case of a grouping operation. Hence, optimizations presented in Sect. 7.3, such as the utilization of already sorted input [101,111], apply here as well.

Set operations Set operations, unless explicitly specified, eliminate duplicate rows. As stated above, the removal of duplicate entries often relies on sorting the input data, thereby opening up opportunities for order optimizations for set operations [101]: We consider a `UNION` statement whose select list consists only of $\boldsymbol{Y}$ and we assume $\boldsymbol{X} \overset{\prec}{\mapsto} \boldsymbol{Y}$ to hold while $\boldsymbol{X}$ has been ordered, for example, by an `ORDER BY` clause that has been pushed down. In such a case, the duplicate elimination within the set operation can avoid an additional sort operation, because the tuples are already sorted with respect to $\boldsymbol{Y}$. Similar optimizations are possible for `INTERSECT` and `EXCEPT`.

Cost estimation While executing subqueries, a query engine might decide to cache previously computed results of the inner query to reuse them in case the inner query is executed with the same correlation attribute values again. In the case of ordered correlation attributes (compare the optimization of Selinger et al. [109] mentioned in Sect. 5.4), e.g., by order dependencies, a cache size of 1 is sufficient, because the order guarantees that the subquery will never be executed with the same inputs again once new inputs are read from the ordered correlation attributes. Again, ODs extend the applicability of such order-based optimizations to further attributes.

Also, cost predictions, e.g., for operator costing, can take ODs into account to more accurately estimate execution costs of operators that rely on ordered data.

8 Inclusion dependencies

Inclusion dependencies often serve data linkage and data integration scenarios as they may span across multiple data

sources. We illustrate in this section, how they can be utilized also for query optimization.

8.1 INDs and joins

By their definition, INDs are most relevant for join operations. An often used join variant are semijoins $R \ltimes_X S$, which are used to filter R by the tuples that have a matching tuple in S. In SQL, semijoins are expressed as joins with a post-projection on the attributes of the R side, which is $\pi_R(R \bowtie_X S)$. Given the IND $R.X \subseteq S.X$ with its formal definition $\forall t_r \in R, \exists t_s \in S : t_r[X] = t_s[X]$, all tuples from R match tuples in S so that the semijoin is redundant; hence, it can be removed. Note that this optimization requires all $R.X$ to be `NOT NULL` or the IND to follow the `null` $\neq$ `null` semantics. This is because the semijoin filters out all tuples with `null` values in $R.X$; the IND $R.X \subseteq S.X$ with `null` $=$ `null` semantics, on the contrary, might find a matching tuple in S with the same `null` values as the R tuple.

A popular, yet dependency-independent optimization technique is *semijoin reduction* [11,112], which reduces the number of tuples considered by join operators before the actual join is conducted. For this reduction, the matching (non-dangling) tuples are determined a priori by utilizing semijoins. The semijoin reduction $(R \ltimes \pi_X(S)) \bowtie_X S$ is equivalent to the plain join operation $R \bowtie_X S$. The rationale is that identifying the matching tuples in R and joining only these with S is more efficient than joining R and S directly. This semijoin technique is particularly useful for distributed database setups that need to minimize expensive data transfers. However, if *all* tuples from R match tuples in S, which is the case under an IND, semijoin reductions are unnecessary but still costly as they effectively perform the join twice. So if the IND $R.X \subseteq S.X$ is known, semijoin reductions on $R.X$ can be avoided. Because the semijoins are used only as pre-filters in this optimization, it is possible to use INDs based on `null` $=$ `null` and `null` $\neq$ `null` semantics in this optimization: By using `null` $=$ `null` INDs, we might eliminate semijoins that would filter some records with `nulls` in the join attributes, but (i) the following join filters them anyway so that the result remains correct, (ii) removing such semijoins might not even impact the performance negatively depending on the number of actually filterable `null` records, because the filtering costs might outweigh the data transfer overhead, and (iii) adding a default `null` check to all join tuples before sending them would easily solve the issue.

Johnson and Klug [64] as well as Cheng et al. [24] discuss and evaluate techniques to eliminate joins in the presence of INDs and foreign-key constraints, respectively. If a foreign-key constraint holds, it allows to eliminate certain joins whose result is known without executing them: Given the IND $R.X \subseteq S.X$ and a UCC on $S.X$ on the relations R and S, then the join $R \bowtie_X S$ can be avoided if there are no further selections or projections on any attributes of S. This is, because the IND guarantees that every tuple of S joins with a tuple in T, and the UCC ensures the absence of multiple matching tuples. The described optimization, again, requires that either $R.X$ is free of `null` values or the IND is valid under `null` $\neq$ `null` semantics.

The idea of join elimination can be extended to intermediate joins: If there are two joins $R \bowtie_X S$ and $S \bowtie_X T$ with the INDs $R.X \subseteq S.X$ and $S.X \subseteq T.X$ and a UCC on $S.X$, the two joins can be reduced to a single join $R \bowtie_X T$ following from the transitivity of INDs [21]. The UCC on $S.X$ is necessary, because duplicate values in $S.X$ would cause record duplication that would be ignored in the optimized case. The removal of S also requires that the SQL query neither contains attributes from S in its final projection nor that it filters S. The authors state that transitive join queries are suboptimal and could be avoided in the first place, but they are nevertheless often seen in practice, because applications and object-relational mappers automatically generate them or database users can access only specific views that contain unnecessary joins. For this reason, query optimizers can benefit from such IND-based optimization techniques.

Deutsch et al. show that in certain cases a join $R \bowtie_X S$ can be replaced with a join $R \bowtie_X T$ [33]. This replacement can be beneficial for peer database systems, in which T might offer faster access or higher availability due to being located at a different site than S. This rewrite requires the INDs $T.X \subseteq S.X$ and $(R \bowtie_X S).X \subseteq T.X$ to hold. Because the second IND depends on a query result, a practical implementation would need to check the stricter, but discoverable IND $S.X \subseteq T.X$.

8.2 Further optimization opportunities with INDs

Although IND-based query optimizations seem to focus primarily on join operations, various further optimization opportunities exist.

Query folding Query folding is an optimization technique to answer queries by rewriting them in a way that lets them use certain resources, e.g., cached query results or materialized views instead of base tables. Jarek Gryz demonstrated in [51] how INDs enable further scenarios for the application of query folding with views.

Assume a query that accesses the attribute combination X of a relation S. If no materialized view contains $S.X$, this query does not have a query folding and cannot be answered using views. However, assuming that there is an IND $R.X \subseteq S.X$ and a materialized view on $R.X$, the query could be rewritten to access $R.X$ instead of $S.X$ so that the answer to the query can be retrieved via query folding from $R.X$'s materialized view. Clearly, the rewrite may return only a subset of the actual result of the initial query, but this might

be acceptable for certain use cases where users need their answers quickly [51].

Deutsch et al. [33] as well as Ileana et al. [57] explain how INDs (the original work uses more general tuple-generating dependencies) enable rewritings that utilize materialized views or result caches if set semantics[12] are assumed. For example, a join $S \bowtie_X T$ can be replaced by accessing a view V_{ST} if the INDs $S \bowtie_X T \subseteq V_{ST}.X$ and $V_{ST}.X \subseteq S \bowtie_X T$ hold.

Exists Correlated subqueries as part of `EXISTS` statements can also be simplified with INDs. Given the IND $R.A \subseteq S.A$, the query

```
SELECT * FROM R WHERE EXISTS
    (SELECT * FROM S WHERE S.A = R.A)
```

can be computed without accessing S, because the IND ensures that the subquery returns `TRUE` for every value of $R.A$. For known foreign-key constraints, such `EXISTS` optimizations have been adopted by productive query optimizers, e.g., by Microsoft SQL Server [84]; whether or not they also use INDs is not known. Note that because joins and select statements do not match `null` values, we require `null` $\neq$ `null` semantics *or* an additional `null` check here.

Set operations Inclusion dependencies can further be used to simplify the computation of set operations. A query of the form

```
SELECT A FROM R
  UNION
SELECT A FROM S
```

could be rewritten to `SELECT DISTINCT A FROM S` if the IND $R.A \subseteq S.A$ holds. By the set-based definition of the `UNION` operation, the result contains all distinct values from $R.A$ and $S.A$ while the IND guarantees that all values from $R.A$ are already included in $S.A$. Here again, the IND must be true under `null` $\neq$ `null` semantics (or $R.A$ must not contain `null` values), because the rewrite would otherwise miss the `null` values from $R.A$ that the `UNION` would have added. Analogously to the `UNION` case, this optimization can be applied to `INTERSECT` operations, but instead of returning the values of $S.A$, the distinct values of $R.A$ need to be returned. For this, the optimization requires an IND with `null` $=$ `null` semantics (or $R.A$ without `null` values), because the `INTERSECT` removes `null` values from $R.A$ if these are also present in $S.A$. Note that in cases of `INTERSECT ALL`, the query optimizer can omit even the `DISTINCT` operation. Besides `UNIQUE` and `INTERSECT` also difference operations, such as

```
SELECT A FROM R
  EXCEPT
SELECT A FROM S
```

can be simplified. In the `EXCEPT` case, the IND guarantees that only an empty result-set can be returned, allowing the query optimizer to skip the execution of the query altogether and return an empty set. Note that `EXCEPT` removals require INDs with `null` $=$ `null` semantics, because the `EXCEPT` removes records with `null` values in $R.A$ only if they are also present in $S.A$.

Cardinality estimation Similar to some of the dependencies discussed before, INDs also allow for more accurate cardinality estimations. Consider, for example, an IND $R.X \subseteq S.X$. An equality join on the attribute combination X of these two relations returns a minimum of $|R.X|$ results, because the IND guarantees for every tuple in R at least one matching tuple in S. Again, we need the `null` $\neq$ `null` semantics for this optimization, because it effectively ensures that the IND is valid only if there are no `null` values in $R.X$. In case $S.X$ is a UCC on $S.X$, the IND basically appears as a foreign-key constraint and returns *exactly* $|R.X|$ results. For *theta* joins with predicate $\neq$, i.e., $R \bowtie_{R.X \neq S.X} S$, the number of results is exactly $|R.X| \times (|S.X| - 1)$ records given the IND $R.X \subseteq S.X$ and the UCC $S.X$. Considering implementations of such cardinality estimation strategies in real products, we found only one example (although further examples might exist): IBM's DB2 [56] database management system uses foreign-key constraints to make cardinality estimations more efficient by reducing the number of considered statistical views.

9 Further optimizations

In this section, we first discuss opportunities for further optimizations with semantic integrity constraints and other types of dependencies. Afterward, we summarize open issues for future work that remain to be solved to close the gap between research and practice in dependency-driven query optimization.

9.1 Semantic query optimization

The field of *semantic query optimization* [22,52,69,129] offers more techniques that utilize further constraints and dependencies for query optimization. For instance, semantic integrity constraints [113] are typically user-defined and encode knowledge about attributes of a relation. For example, a German citizen relation might follow the semantic integrity constraint $citizen.city = Berlin \rightarrow citizen.zipcode \in [10115, 14199]$. While such constraints can also be used for efficient query processing, they differ from data dependencies that do not necessarily carry any semantic meaning.

[12] Set semantics occur in SQL-based DBMS in the presence of keys, set operations, or the `DISTINCT` keyword.

9.2 Further dependency types

A substantial body of work discusses how the combination of the *Chase* and *Backchase* procedures can be used to find minimal, equivalent plans for a particular query [31,82]. These procedures can reveal opportunities to use certain auxiliary structures, such as materialized views or indexes, for answering a query [104]. The dependencies that are often used in the aforementioned work are so-called equality-generating dependencies (egds) and tuple-generating dependencies (tgds) [10]. Both egds and tgds can be seen as generalizations of other dependencies [33], including the dependencies covered in this work. As such, we already showed their usefulness in Sects. 8.1 and 8.2. There are, however, no general automatic discovery algorithms for tgds and egds (yet). What is more, they are usually present in integration scenarios where they span multiple schemata as derived from user-defined schema mappings [40]. Further, prior work often assumes set semantics [32,57], so it is not always suitable for typical relational database scenarios in which bag semantics prevail.

Although the data dependencies examined in this survey, namely UCCs, FDs, ODs, and INDs, are arguably the most important dependencies for query optimization today, many further types of discoverable data dependencies and data rules exist, such as multi-valued dependencies (MVDs) [39], neighborhood dependencies (NDs) [9], sequential dependencies (SDs) [48], denial constraints (DCs) [12], and others. If an SQL query tests for such a dependency, this test can be avoided if the dependency is already known to be true. Apart from this general rule, we find hardly any research on the use of these dependencies for query optimization and, thus, do not cover them in this survey.

10 Summary and outlook

In this paper, we surveyed various core techniques with which database management systems can use genuine and observed data dependencies, namely uniqueness, functional, order, and inclusion dependencies, to improve their query optimization capabilities. The surveyed optimizations are of increasing relevance, because recent advances in the field of automatic data profiling algorithms now enable the efficient discovery of such metadata for large and, hence, practically relevant datasets—database systems today have access not only to user-defined dependencies and constraints but also to much larger corpora of automatically discovered dependencies.

The compilation of techniques presented in this survey can serve as a starting point to equip query optimizers with interesting and potentially powerful optimization capabilities that rely on data dependencies. Furthermore, it supports database engineers of established systems in identifying additional techniques to further increase the efficiency of their already advanced execution engines or query optimizers. The survey finally also provides researchers with an exhaustive collection of related work and facilitates the uncovering of open research questions and opportunities for future work. Next, we summarize three major research directions.

Efficient implementation and integration While some of the mentioned optimization techniques require relatively simple implementations or are already implemented in some of today's database systems, other advanced techniques will be much more complex to incorporate in practice and might require broad modifications of existing systems. Even though many of the surveyed techniques have been evaluated in the respective research papers, transferring them to practice in commercial database systems is not an easy undertaking and, therefore, a challenge for future work.

Incremental discovery and maintenance The knowledge of available data dependencies is essential for all dependency-driven optimization techniques. Although we did reference incremental discovery and maintenance approaches for all of the four considered dependency types, more efficient approaches that can handle large dynamic datasets under real-world workloads are necessary to enable the surveyed query optimization techniques in practice.

Empirical impact evaluation While this survey collects many optimization techniques, their effectiveness, relevance and overhead is still to be evaluated. To measure effectiveness, a systematic study is needed that measures the impact of the various optimizations on query performance. Such impacts are neither obvious nor simple to determine, as they depend on various factors, such as the database system's implementation, the specific query at hand, and the underlying dataset. To judge the relevance of each optimization, an empirical study is needed that evaluates how often and, hence, how likely the presented techniques apply to queries in real-world workloads. This study is a challenging task as it requires a representative collection of query workloads. Finally, another study to quantify the optimization overhead introduced by the proposed optimization techniques would require specific, well-tuned implementations to measure potentially elevated optimizer runtimes and assess code complexity.

Acknowledgements We thank Glenn Norman Paulley for his valuable input to this survey.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Abadi, D.J., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 671–682 (2006)
2. Abadi, D.J., Madden, S., Hachem, N.: Column-stores vs. row-stores: how different are they really? In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 967–980 (2008)
3. Abedjan, Z., Golab, L., Naumann, F.: Profiling relational data: a survey. VLDB J. **24**(4), 557–581 (2015)
4. Abedjan, Z., QuianéRuiz, J.-A., Naumann, F.: Detecting unique column combinations on dynamic data. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 1036–1047 (2014)
5. Abedjan, Z., et al.: Data Profiling. Vol. 10. Synthesis Lectures on Data Management 4. Morgan & Claypool Publishers (2018)
6. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Boston (1995)
7. Armstrong, W.W.: Dependency structures of data base relationships. In: IFIP Congress, pp. 580–583 (1974)
8. Atzeni, P., Morfuni, N.M.: Functional dependencies and constraints on null values in database relations. Inf. Control **70**(1), 1–31 (1986)
9. Bassée, R., Wijsen, J.: Neighborhood dependencies for prediction. In: Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), pp. 562–567. Springer, Berlin (2001)
10. Beeri, C., Vardi, M.: A proof procedure for data dependencies. J. ACM **31**(4), 718–741 (1984)
11. Bernstein, P.A., et al.: Query processing in a system for distributed databases (SDD-1). ACM Trans. Database Syst. **6**(4), 602–625 (1981)
12. Bertossi, L.E.: Database Repairing and Consistent Query Answering. Morgan and Claypool Publishers (2011)
13. Birnick, J., et al.: Hitting set enumeration with partial information for unique column combination discovery. PVLDB **13**(11), 2270–2283 (2020)
14. Bläsius, T., Friedrich, T., Schirneck, M.: The parameterized complexity of dependency detection in relational databases. In: Proceedings of the International Symposium on Parameterized and Exact Computation (IPEC), vol. 6, no. (1–6), p. 13 (2017)
15. Bohannon, P., et al.: Conditional functional dependencies for data cleaning. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 746–755 (2007)
16. Boissier, M., Schlosser, R., Uflacker, M.: Hybrid data layouts for tiered HTAP databases with pareto-optimal data placements. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 209– 220 (2018)
17. Boncz, P.A., Neumann, T., Erling, O.: TPC-H analyzed: hidden messages and lessons learned from an influential benchmark. In: Performance Characterization and Benchmarking—5th TPC Technology Conference (TPCTC), pp. 61–76 (2013)
18. Brown, P., Haas, P.J.: BHUNT: automatic discovery of fuzzy algebraic constraints in relational data. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 668–679 (2003)
19. Caruccio, L., Deufemia, V., Polese, G.: Relaxed functional dependencies: a survey of approaches. IEEE Trans. Knowl. Data Eng. **28**(1), 147–165 (2016)
20. Caruccio, L.: et al.: Incremental discovery of functional dependencies with a bit-vector algorithm. In: Proceedings of the Italian Symposium on Advanced Database Systems (2019)
21. Casanova, M.A., Fagin, R., Papadimitriou, C.H.: Inclusion dependencies and their interaction with functional dependencies. In: Proceedings of the Symposium on Principles of Database Systems (PODS), pp. 171–176 (1982)
22. Chakravarthy, U.S., Grant, J., Minker, J.: Logic-based approach to semantic query optimization. ACM Trans. Database Syst. **15**(2), 162–207 (1990)
23. Chaudhuri, S., Shim, K.: Including group-by in query optimization. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 354–366 (1994)
24. Cheng, Q., et al.: Implementation of two semantic query optimization techniques in DB2 universal database. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 687–698 (1999)
25. Ciaccia, P., Golfarelli, M., Rizzi, S.: On estimating the cardinality of aggregate views. In: Proceedings of the International Workshop on Design and Management of Data Warehouses, pp. 12.1–12.10 (2001)
26. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
27. Codd, E.F.: Further normalization of the data base relational model. In: IBM Research Report, San Jose, California RJ 909 (1971)
28. Codd, E.F.: Understanding relations (Installment #7). FDT Bull. ACM SIGFIDET SIGMOD **7**(3), 23–28 (1975)
29. Date, C.J., Darwen, H.: Relational database writings 1989–1991. In: The Role of functional Dependence in Query Decomposition, pp. 133–150. Addison-Wesley. Chap (1992)
30. Dayal, U.: Of nests and trees: a unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 197–208 (1987)
31. Deutsch, A., Nash, A., Remmel, J.B.: The chase revisited. In: Proceedings of the Symposium on Principles of Database Systems (PODS), pp. 149–158 (2008)
32. Deutsch, A., Popa, L., Tannen, V.: Physical data independence, constraints, and optimization with universal plans. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 459–470 (1999)
33. Deutsch, A., Popa, L., Tannen, V.: Query reformulation with constraints. SIGMOD Rec. **35**(1), 65–73 (2006)
34. Dong, J., Hull, R.: Applying approximate order dependency to reduce indexing space. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 119–127 (1982)
35. Downey, R.G., Fellows, M.R., Regan, K.W.: Parameterized circuit complexity and the W hierarchy. Theor. Comput. Sci. **191**(1–2), 97–115 (1998)
36. Dreseler, M., et al.: 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), pp. 313–324 (2019)
37. Dürsch, F, et al.: Inclusion dependency discovery: an experimental evaluation of thirteen algorithms. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM), pp. 219–228 (2019)
38. Eich, M., Fender, P., Moerkotte, G.: Faster plan generation through consideration of functional dependencies and keys. PVLDB **9**(10), 756–767 (2016)

39. Fagin, R.: Multivalued dependencies and a new normal form for relational databases. ACM Trans. Datab. Syst. **2**(3), 262–278 (1977)
40. Fagin, R., Kolaitis, P.G., Popa, L.: Data exchange: getting to the core. ACM Trans. Database Syst. **30**(1), 174–210 (2005)
41. Flach, P.A., Savnik, I.: Database dependency discovery: a machine learning approach. AI Commun. **12**(3), 139–160 (1999)
42. Ganguly, S., Hasan, W., Krishnamurthy, R.: Query optimization for parallel execution. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 9–18 (1992)
43. Ganski, R.A., Wong, H.K.T.: Optimization of nested SQL queries revisited. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 23–33 (1987)
44. Gelenbe, E., Gardy, D.: The size of projections of relations satisfying a functional dependency. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 325–333 (1982)
45. Giannella, C., et al.: Improving query evaluation with approximate functional dependency based decompositions. In: Proceedings of British National Conference on Databases BNCOD, pp. 26–41 (2002)
46. Ginsburg, S., Hull, R.: Order dependency in the relational model. Theor. Comput. Sci. **26**, 149–195 (1983)
47. Ginsburg, S., Hull, R.: Sort sets in the relational model. J. ACM **33**(3), 465–488 (1986)
48. Golab, L., et al.: Sequential dependencies. PVLDB **2**(1), 574–585 (2009)
49. Graefe, G.: Query evaluation techniques for large databases. ACM Comput. Surv. **25**(2), 73–170 (1993)
50. Graefe, G.: The cascades framework for query optimization. IEEE Data Eng. Bull. **18**(3), 19–29 (1995)
51. Gryz, J.: Query folding with inclusion dependencies. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 126–133 (1998)
52. Hammer, M., Zdonik, S.B.: Knowledge-based query processing. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 137–147 (1980)
53. Heise, A., et al.: Scalable discovery of unique column combinations. PVLDB **7**(4), 301–312 (2013)
54. Huhtala, Y., et al.: Efficient discovery of functional and approximate dependencies using partitions. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 392–401 (1998)
55. Huhtala, Y., et al.: TANE: an efficient algorithm for discovering functional and approximate dependencies. Comput. J. **42**(2), 100–111 (1999)
56. IBM. Referential integrity constraints help reduce the number of statistical views. (2019). https://www.ibm.com/support/knowledgecenter/SSEPGG_11.5.0/com.ibm.db2.luw.admin.perf.doc/doc/c0059081.html (visited on 04/28/2020)
57. Ileana, I., et al.: Complete yet practical search for minimal query reformulations under constraints'. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 1015–1026 (2014)
58. Ilyas, I.F. et al.: CORDS: automatic discovery of correlations and soft functional dependencies. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 647–658 (2004)
59. International Organization for Standardization: ISO/IEC 9075-2:1999 (SQL Standard 1999). Standard (1999)
60. International Organization for Standardization: ISO/IEC 9075:1992 (SQL Standard 1992). Standard (1992)
61. Ioannidis, Y.E.: Query optimization. ACM Comput. Surv. **28**(1), 121–123 (1996)
62. Klebanoff, J: Apache derby: intersect and except design (2005). https://db.apache.org/derby/papers/Intersect-design.html (visited on 11/23/2020)
63. Jarke, M., Koch, J.: Query optimization in database systems. ACM Comput. Surv. **16**(2), 111–152 (1984)
64. Johnson, D.S., Klug, A.C.: Testing containment of conjunctive queries under functional and inclusion dependencies. J. Comput. Syst. Sci. **28**(1), 167–189 (1984)
65. Kambayashi, Y., Yoshikawa, M.: Query processing utilizing dependencies and horizontal decomposition. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 55–67 (1983)
66. Kemper, A., Neumann, T.: HyPer: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 195–206 (2011)
67. Kim, W.: On optimizing an SQL-like nested query. ACM Trans. Datab. Syst. **7**(3), 443–469 (1982)
68. Kimura, H., et al.: Correlation maps: a compressed access method for exploiting soft functional dependencies. PVLDB **2**(1), 1222–1233 (2009)
69. King, J.J.: Modelling concepts for reasoning about access to knowledge. In: Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling, pp. 138–140 (1980)
70. Klaebe, S., Baumann, S., Sattler, K.-U.: PatchIndex - exploiting approximate constraints in self-managing databases. In: Proceedings of the International Conference on Data Engineering (ICDE) Workshops, pp. 139–146 (2020)
71. Köhler, H., Link, S.: SQL schema design: foundations, normal forms, and normalization. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 267–279 (2016)
72. Köhler, H., Link, S., Zhou, X.: Possible and certain SQL keys. PVLDB **8**(11), 1118–1129 (2015)
73. Kruse, S., Papenbrock, T., Naumann, F.: Scaling out the discovery of inclusion dependencies. In: Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW), pp. 445–454 (2015)
74. Larson, P.-A., Galindo-Legaria, C.A.: Partial pre-aggregation in relational database queries. US Patent 7,593,926 (2009)
75. Leis, V., et al.: How good are query optimizers, really? PVLDB **9**(3), 204–215 (2015)
76. Leis, V., et al.: Query optimization through the looking glass, and what we found running the Join Order Benchmark. VLDB J. **27**(5), 643–668 (2018)
77. Liu, J., et al.: Discover dependencies from data: a review. IEEE Trans. Knowl. Data Eng. **24**(2), 251–264 (2012). As per manuscript, References [77] and [78] are same, so we have deleted the duplicate reference and renumbered accordingly. Please check and confirm
78. Liu, L., Özsu, M.T. (eds.): Encyclopedia of Database Systems, 2nd edn. Springer, Berlin (2018). After renumbered the reference [78] is given in list but not cited in text. Please cite in text or delete from list
79. Lucchesi, C.L., Osborn, S.L.: Candidate keys for relations. J. Comput. Syst. Sci. **17**(2), 270–279 (1978)
80. Manegold, S.: Cost estimation. In: Liu, L., Özsu, M.T. (eds.) Encyclopedia of Database Systems, 2nd edn. Springer, Berlin (2018)
81. Marchi, D.F., Lopes, S., Petit, J.-M.: Unary and n-ary inclusion dependency discovery in relational databases. J. Intell. Inf. Syst. **32**, 53–73 (2009)
82. Meier, M.: The backchase revisited. VLDB J. **23**(3), 495–516 (2014)
83. Memarzia, P., Ray, S., Bhavsar, V.C.: A six-dimensional analysis of in-memory aggregation. In: Proceedings of the International

Conference on Extending Database Technology (EDBT), pp. 289–300 (2019)
84. Microsoft: Foreign key constraints (without NOCHECK) boost performance and data integrity (2004). https://web.archive.org/web/20101219111457/http://www.microsoft.com/technet/abouttn/flash/tips/tips_122104.mspx (visited on 04/28/2020)
85. Microsoft: Optimizing queries that access correlated date-time columns (2008). https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2005/ms177416(v=sql.90)?redirectedfrom=MSDN (visited on 04/30/2020)
86. Müller, I. et al.: Cache-efficient aggregation: hashing is sorting. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 1123–1136 (2015)
87. Mullin, J.K.: Optimal semijoins for distributed database systems. ACM Trans. Softw. Eng. **16**(5), 558–560 (1990)
88. MySQL: MySQL 8.0 reference manual—optimizing IN and EXISTS subquery predicates with semijoin transformations (2020). https://dev.mysql.com/doc/refman/8.0/en/semijoins.html (visited on 05/07/2020)
89. MySQL: MySQL 8.0 reference manual—SELECT statement (2020). https://dev.mysql.com/doc/refman/8.0/en/select.html (visited on 11/23/2020)
90. Nambiar, U., Kambhampati, S.: Mining approximate functional dependencies and concept similarities to answer imprecise queries. In: Proceedings of the ACM Workshop on the Web and Databases (WebDB), pp. 73–78 (2004)
91. Navathe, S.B., et al.: Vertical partitioning algorithms for database design. ACM Trans. Datab. Syst. **9**(4), 680–710 (1984)
92. Neumann, T.: Engineering high-performance database engines. PVLDB **7**(13), 1734–1741 (2014)
93. Neumann, T., Kemper, A.: Unnesting arbitrary queries. In: Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW), pp. 383–402 (2015)
94. O'Neil, P.E.: Database Principles, Programming, Performance. Morgan Kaufmann, Burlington (1994)
95. Oracle: Scalar subquery expressions (2019). https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Scalar-Subquery-Expressions.html (visited on 09/06/2019)
96. Papenbrock, T., Naumann, F.: A hybrid approach to functional dependency discovery. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 821–833 (2016)
97. Papenbrock, T., Naumann, F.: Data-driven schema normalization. In: Proceedings of the International Conference on Extending Database Technology (EDBT), pp. 342–353 (2017)
98. Papenbrock, T., et al.: Divide and conquer-based inclusion dependency discovery. PVLDB **8**(7), 774–785 (2015)
99. Papenbrock, T., et al.: Functional dependency discovery: an experimental evaluation of seven algorithms. PVLDB **8**(10), 1082–1093 (2015)
100. Paulley, G.N., Larson, P.: Exploiting uniqueness in query optimization. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 68–79 (1994)
101. Paulley, G.N.: Exploiting Functional Dependence in Query Optimization. AAINQ51220. Ph.D Thesis. Waterloo, ON, Canada (2000)
102. Pirahesh, H., Hellerstein, J.M., Hasan, W.: Extensible/rule based query rewrite optimization in starburst. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 39–48 (1992)
103. Pitoura, E.: Query rewriting. In: Liu, L., Özsu, M.T. (eds.) Encyclopedia of Database Systems, 2nd edn. Springer, Berlin (2018)
104. Popa, L. et al.: A chase too far? In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 273–284 (2000)
105. Saiedian, H., Spencer, T.: An efficient algorithm to compute the candidate keys of a relational database schema. Comput. J. **39**(2), 124–132 (1996)
106. SAP: Expressions—Subqueries in Expressions (2019). https://help.sap.com/viewer/4fe29514fd584807ac9f2a04f6754767/2.0.04/en-US/20a4389775191014b5a6bf2ccc0df2ed.html (visited on 04/28/2020)
107. Schiefer, B., Strain, L.G., Yan, W.P.: Method for estimating cardinalities for query processing in a relational database management system. US Patent 5,761,653 (1998)
108. Schirmer, P., et al.: DynFD: functional dependency discovery in dynamic datasets. In: Proceedings of the International Conference on Extending Database Technology (EDBT), pp. 253–264 (2019)
109. Selinger, P.G., et al.: Access path selection in a relational database management system. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 23–34 (1979)
110. Shaabani, N., Meinel, C.: Incremental discovery of inclusion dependencies. In: Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM), pp. 2:1–2:12. ACM (2017)
111. Simmen, D.E., Shekita, E.J., Malkemus, T.: Fundamental techniques for order optimization. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 57–67 (1996)
112. Stocker, K., et al.: Integrating semi-join-reducers into state of the art query processors. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 575–584 (2001)
113. Stonebraker, M.: Implementation of integrity constraints and views by query modification. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 65–78 (1975)
114. Szlichta, J., Godfrey, P., Gryz, J.: Chasing polarized order dependencies. In: Proceedings of the Alberto Mendelzon International Workshop on Foundations of Data Management, pp. 168–179 (2012)
115. Szlichta, J., Godfrey, P., Gryz, J.: Fundamentals of order dependencies. PVLDB **5**(11), 1220–1231 (2012)
116. Szlichta, J., et al.: Business-intelligence queries with order dependencies in DB2. In: Proceedings of the International Conference on Extending Database Technology (EDBT), pp. 750–761 (2014)
117. Szlichta, J., et al.: Effective and complete discovery of order dependencies via set-based axiomatization. PVLDB **10**(7), 721–732 (2017)
118. Szlichta, J., et al.: Queries on dates: fast yet not blind. In: Proceedings of the International Conference on Extending Database Technology (EDBT), pp. 497–502 (2011)
119. Tan, Z., et al.: Fast incremental discovery of pointwise order dependencies. PVLDB **13**(10), 1669–1681 (2020)
120. The PostgreSQL Global Development Group: PostgreSQL: Documentation: 13: 70.1. Row estimation examples (2020). https://www.postgresql.org/docs/13/row-estimation-examples.html (visited on 10/30/2020)
121. Wang, S.-L., et al.: Maintenance of discovered functional dependencies: incremental deletion. In: Intelligent Systems Design and Applications, pp. 579–588. Springer, Heidelberg (2003)
122. Weddell, G.E.: Reasoning about functional dependencies generalized for semantic data models. ACM Trans. Database Syst. **17**(1), 32–64 (1992)
123. Wei, Z., Leck, U., Link, S.: Entity integrity, referential integrity, and query optimization with embedded uniqueness constraints. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 1694–1697 (2019)

124. Wei, Z., Link, S.: Embedded functional dependencies and data-completeness tailored database design. PVLDB **12**(11), 1458–1470 (2019)
125. Yan, W.P.: Query Optimization Techniques for Aggregation Queries'. Ph.D Thesis, Department of Computer Science, University of Waterloo (1995)
126. Yan, W.P., Larson, P.: Eager aggregation and lazy aggregation. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 345–357 (1995)
127. Yan, W.P., Larson, P.: Performing group-by before join. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 89–100 (1994)
128. Yang, H.Z., Larson, P.: Query transformation for PSJ-queries. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 245–254 (1987)
129. Yu, C.T., Sun, W.: Automatic knowledge acquisition and maintenance for semantic query optimization. IEEE Trans. Knowl. Data Eng. **1**(3), 362–375 (1989)