DPQL: The Data Profiling Query Language

Marcian Seeger[1] Sebastian Schmidl[2] Alexander Vielhauer[3] Thorsten Papenbrock[4]

Abstract: Data profiling describes the activity of extracting implicit metadata, such as schema descriptions, data types, and various kinds of data dependencies, from a given data set. The considerable amount of research papers about novel metadata types and ever-faster data profiling algorithms emphasize the importance of data profiling in practice. Unfortunately, though, the current state of data profiling research fails to address practical application needs: Typical data profiling algorithms (i. e., challenging to operate structures) discover all (i. e., too many) minimal (i. e., the wrong) data dependencies within minutes to hours (i. e., too long). Consequently, if we look at the practical success of our research, we find that data profiling targets data cleaning, but most cleaning systems still use only hand-picked dependencies; data profiling targets query optimization, but hardly any query optimizer uses modern discovery algorithms for dependency extraction; data profiling targets data integration, but the application of automatically discovered dependencies for matching purposes is yet to be shown - and the list goes on. We aim to solve the profiling-and-application-disconnect with a novel data profiling engine that integrates modern profiling techniques for various types of data dependencies and provides the applications with a versatile, intuitive, and declarative Data Profiling Query Language (DPQL). The DPQL enables applications to specify precisely what dependencies are needed, which not only refines the results and makes the data profiling process more accessible but also enables much faster and (in terms of dependency types and selections) holistic profiling runs. We expect that integrating modern data profiling techniques and the post-processing of their results under a single application endpoint will result in a series of significant algorithmic advances, new pruning concepts, and a profiling engine with innovative components for workload autoconfiguration, query optimization, and parallelization. With this paper, we present the first version of the DPQL syntax and its semantics, which introduces a fundamentally new line of research in data profiling.

Keywords: data profiling; query language; functional dependencies; unique column combinations; inclusion dependencies

1 About Data Profiling and Application Requirements

Structural metadata is a set of rules that shape datasets, their formats, evolution, correctness, and accessibility. For this reason, metadata is an essential input to many data management processes ranging from data exploration [Fe18; Ro09] over data integration [DR02; Zh10]

[1] Philipps-University of Marburg, Big Data Analytics, Hans-Meerwein-Str. 6, 35032 Marburg, Germany seegerma@students.uni-marburg.de
[2] Hasso Plattner Institute, University of Potsdam, Information Systems, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany sebastian.schmidl@hpi.de
[3] Philipps-University of Marburg, Big Data Analytics, Hans-Meerwein-Str. 6, 35032 Marburg, Germany avielhauer@informatik.uni-marburg.de
[4] Philipps-University of Marburg, Big Data Analytics, Hans-Meerwein-Str. 6, 35032 Marburg, Germany papenbrock@informatik.uni-marburg.de

doi:10.18420/BTW2023-19

and data cleaning [IC15; VA18] to query optimization [KPN22; Pa00] and machine learning [Ch19; KPN20] to name only a few. Due to the importance of metadata, most database management systems store not only the data but also structural information, such as data types, basic statistics, and constraints. This is only a fraction of the structural metadata that characterizes a dataset, and actually having access to it is a lucky case because many formats and systems for storing datasets do not even provide any metadata. For this reason, data engineers (and scientists) conduct *data profiling* [Ab18] to extract metadata from raw data. This first manual and meanwhile largely automated process has been improved significantly over the past 30 years. To give a few examples, we can now automatically mine *unique column combinations* [Bi20], *functional dependencies* [PN16], *inclusion dependencies* [Dü19], *order dependencies* [SP22], *matching dependencies* [Sc20] and many more in exact and various relaxed versions [CDP16]. In the quest to meet application needs and user skills, the corresponding data profiling algorithms have been built into practical data profiling tools, such as Metanome [Pa15a], Desbordante [De22], or Viadotto [Vi22]. Despite these technological advances, data profiling still requires complicated and often manual post-processing efforts to make use of the discovered metadata in different applications.

To illustrate the current limitations in data profiling, consider the following example: In a data integration scenario, a data engineer is looking for possible foreign-key candidates between two to-be-integrated datasets R and S. A suitable foreign-key candidate is an inclusion dependency (IND) $X \subseteq Y$ where the attributes X and Y are from different datasets and the target Y is a key candidate, i. e., a unique column combination (UCC). The standard approach would be to, first, discover all INDs and UCCs and, then, filter the required statements for the actual results. This process introduces the following major issues:

Discovery of too many results: Many data profiling algorithms are exponential in their output complexity because the amount of syntactically valid and, hence, discoverable metadata is huge. They usually restrict the outputs to only minimal (or maximal) metadata statements, but the metadata result sets still often outgrow storage capacities and the data itself [Dü19; Pa15b]. Any semantic metadata selection is usually conducted as a post-processing step and deferred to the metadata application. If these applications could formulate their metadata requirements as pruning rules for the profiling of the data, giant result sets could be avoided. In our example, only very few INDs overlap with a UCC, such that a clever profiling run would never need to enumerate all INDs and UCCs.

Discovery of the wrong results: To limit the size of metadata result sets, the profiling algorithms restrict the enumeration to only minimal (or maximal) statements. It is possible to derive any valid metadata statement from these collections, but the inference requires complex post-processing procedures based on different axiomatizations [Ab18]. If the INDs in our example are all maximal and the UCCs are all minimal, then the needed foreign-key candidates might be formed by an IND-UCC-combination in which neither the IND is maximal nor the UCC is minimal; and apart from linking the two dependency statements, additional inference work based on IND- and UCC-axioms is needed. A sophisticated

profiling algorithm would do this already during metadata discovery, which requires a standardized profiling language to configure the algorithm executions accordingly.

Discovery in the wrong way: All state-of-the-art data profiling tools operate on a one-type-at-a-time basis, which means that they offer distinct discovery functionalities for every type of metadata. To fulfill a certain metadata need, a user must first decompose the application demands into these different metadata types. For each type, the best algorithm then needs to be selected and parameterized. The latter involves specifying whether relaxation, approximation, parallelization, disk-swapping, etc. is needed and if yes, to what degree. The user must also configure algorithm-specific parameters, such as window sizes, search depths, or filter sizes. This complexity prevents many users from applying modern data profiling tools. For the foreign-key discovery example, the data engineer needs to parameterize an IND and UCC algorithm and combine the results, which is something that a holistic, application-driven data profiling tool with a simple, declarative query language should be able to do automatically.

Discovery that takes too long: Data profiling algorithms are highly optimized, extremely effective metadata discovery tools; they are still output bound, and the outputs grow exponentially with the input sizes. For this reason, even the most effective algorithms may take hours to days to enumerate complete metadata sets for certain inputs [Kr16]. The only way to achieve further significant performance improvements is to pull the application-specific selection of the metadata statements from the preprocessing into the profiling algorithms. This requires a generic language for pruning rules and holistic profiling algorithms that discover multiple types of metadata simultaneously. In our example, we aim to discover INDs and UCCs simultaneously and, for this, need to specify the relationship between them. These relationships can be specified with a data profiling query language and translate directly into pruning rules for the discovery.

A holistic data profiling engine with a standardized *Data Profiling Query Language (DPQL)* would resolve all four mentioned issues: The declarative query language serves to formulate exactly what metadata statements are needed, such that only truly required results (not too many) and carefully linked results (not the wrong) are discovered. Based on the explicit metadata queries, the data profiling engine can automate the parameterization (not in the wrong way) and optimize the discovery strategy (not too long). In this paper, we introduce the first version of such a data profiling query language and provide concrete examples of its usage. DPQL is a generic, SQL-like language that serves to specify metadata requirements across different metadata types. From a user perspective, DPQL is an intuitive interface to filter and join metadata statements that are transparently discovered on demand.

Holistic data profiling via a standardized, declarative metadata query language is a fundamentally new approach to data profiling and should have a major impact on how data profiling algorithms and tools are developed in the future. The descriptions of the DPQL language in this paper focus on the three most popular data dependencies, which are UCCs, FDs, and INDs, but they generalize to all other types of dependencies and metadata statements.

In the subsequent sections, we first introduce a small running example with a practical DPQL query (Sect. 2). Then, we discuss related work on data profiling and profiling-related query languages (Sect. 3) and recap the definitions of UCCs, FDs, and INDs (Sect. 4). Afterward, we introduce DPQL's **SELECT-FROM-WHERE** syntax and result format (Sect. 5). We then present the novel functions that can be used within a DPQL query and explain their purposes (Sect. 6). As an evaluation of DPQL, we consider different application areas of data profiling and formulate their demands in DPQL (Sect. 7). In the end, we motivate novel research challenges inspired by DPQL (Sect. 8) and summarize our proposal (Sect. 9).

2 A Running Example

ID	Name	Evolution	Location	Sex	Weight	Size	Type	Weak	Strong
25	Pikachu	Raichu	Viridian Forest	m/f	6.0	0.4	electric	ground	water
29	Nidoran	Nidorino	Safari Zone	m	9.0	0.5	poison	ground	gras
32	Nidoran	Nidorina	Safari Zone	f	7.0	0.4	poison	ground	gras
63	Abra	Kadabra	Cerulean Cave	m/f	19.5	0.9	psychic	ghost	fighting
64	Kadabra	Simsala	Cerulean Cave	m/f	56.5	1.3	psychic	ghost	fighting

(a) `Pokemon`

Title	Biome	Region
Viridian Forest	gras	Kanto
Safari Zone	gras	Kanto
Cerulean Cave	rock	Kanto
Fuchsia City	fighting	Kanto
Anemonia City	water	Jotho

(b) `Locations`

Firstname	Rank	Pokecount
Marcian	8	38
Sebastian	5	42
Alexander	2	19
Thorsten	1	2
Elisa	9	73

(c) `Trainers`

Trainer	Pokemon
Marcian	64
Elisa	29
Elisa	32
Sebastian	25
Sebastian	64

(d) `Teams`

Tab. 1: A running example with a small excerpt of Pokémon data.

As an introduction to DPQL, let us examine a small example with the Pokémon data shown in Tab. 1. In this example, we aim to discover all foreign-key relationships between the tables in the Pokémon dataset. A foreign-key is an integrity constraint between two lists of attributes that requires an *inclusion dependency* between the attribute lists and a *unique column combination* on the referenced attribute list. In some domain-specific settings, we might also want these relationships to cover at most *two attributes*, to link attributes from *different tables*, and to have *at*

```
1  SELECT
2    X AS ForeignKey, Y AS Key
3  FROM
4    CC(Pokemon,Locations,Trainers,Teams) X,
5    CC(Pokemon,Locations,Trainers,Teams) Y
6  WHERE
7    IND(X,Y)
8    AND UCC(Y)
9    AND SPLIT(X,Y)
10   AND SIZE(Y) <= 2
11   AND CARDINALITY(X) >= 2
```

List. 1: Find all foreign-key candidates between the tables `Pokemon`, `Locations`, `Trainers`, and `Teams`.

least two different values in the foreign-key columns. All these conditions can be formulated in a single DPQL query, such as the one shown in List. 1. Following the SQL syntax, the **SELECT** clause defines the output to be pairs of attribute lists `X` and `Y`, which represent the needed `ForeignKey` and `Key` attributes, respectively. The **FROM** clause specifies the search space for `X` and `Y` with the help of the **CC**() function that describes all possible *column combinations* of its arguments; for this, DPQL needs to be able to fetch attribute information from relations. The **WHERE** clause specifies the constraints on the metadata that we are looking for: the IND `X`⊆`Y` and the UCC `Y` should be valid (**IND**(X,Y) and **UCC**(Y)), `X` and `Y` should be from different relations (**SPLIT**(X,Y))), the size of the foreign-key should not be greater than two (**SIZE**(Y) <= 2), and `X` should contain at least two different values (**CARDINALITY**(X) >= 2). The answer to this query contains the tuples ([*Trainer*], [*Firstname*]), ([*Pokemon*], [*ID*]), and ([*Location*], [*Title*]), which are precisely the foreign-keys of the Pokémon dataset.

To obtain the result of a DPQL query, a novel data profiling engine is needed that can parse the filter criteria from the query and apply them effectively. Note that the query implies many implicit profiling constraints, e. g., that `X` and `Y` need to be of the same size, both column combinations need to be lists while column combinations that do not appear in INDs can be interpreted as sets, and the results should be minimal/maximal according to dependency axioms. These constraints do not need to be specified and can automatically be derived by the profiling engine and algorithms. To the best of our knowledge, not a single existing data profiling system can consider all such profiling constraints.

For demonstration purposes, we implemented a very early query processing prototype for the DPQL language that can answer the queries shown in this paper. With the prototype, we executed the foreign-key query of List. 1 on the TPC-H (425 MB, 7 Tables, and 56 Attributes) and the MusicBrainz (104 GB, 232 Tables, and 1 562 Attributes) datasets: The foreign-key query on the TPC-H dataset yields 19 foreign-key candidates containing all seven true foreign-key constraints; in contrast, a full profiling run yields 52 maximal INDs and 408 minimal UCCs that still need to be combined. The foreign-key query on the MusicBrainz dataset yields 7 625 foreign-key candidates; in contrast, a full profiling run yields 209 572 unary INDs and 496 minimal UCCs that still need to be combined. These experiments demonstrate that DPQL queries can produce smaller and more specific results; it enables holistic profiling and new pruning rules for faster executions.

We need to emphasize that most real-world datasets are much wider and longer than our tiny Pokémon example dataset; additionally, they often lack descriptive labels, offer only cryptic values, and are hard to parse. For this reason, automatic data profiling starting at the source data and delivering suitable results directly to the applications – just as we did with the foreign-key query – is highly needed. DPQL is a first and essential building block for this.

3 Related Work

Most of the research on data profiling focuses primarily on improving existing data profiling algorithms including aspects, such as their scalability [Sc20; SGI19; SP22], relaxation [Ca21b; CDP16; Li20; WHL21], or dynamics [Ca21a; Xi22]. Consequently, many very effective algorithms exist for the discovery of basic metadata [HN17; HPN21], unique column combinations [Bi20; WLL19], functional dependencies [PN16; WLL19], inclusion dependencies [Dü19; Pa15c], order dependencies [SP22; Sz17], matching dependencies [Sc20; Wa17], denial constraints [BKN17; PAN21] and many further. All these algorithms target only one type of dependency and try to enumerate complete result sets; they are written in different languages and serve quite heterogeneous interfaces and result formats, which makes them relatively difficult to apply in real-world settings.

So far, very little research has been done on holistic profiling techniques. Some works exist that consider UCCs and FDs simultaneously [Eh16; Hu99] or reason about FDs and INDs jointly [HL18]. These approaches demonstrate the potential of holistic data profiling w.r.t. runtime improvements, but a query language is needed to cover more than two types of dependencies and semantically combine and filter the results.

Data profiling tools aggregate profiling algorithms and present them as services to the user. They make the algorithms easier to operate and store the results in some tool-specific but at least type-unified format. The open-source research framework *Metanome* [Pa15a] was the first data profiling tool to support the discovery of various types of metadata. Another very recent profiling tool inspired by Metanome is *Desbordante* [De22]. Meanwhile, commercial products, such as *Viadotto* [Vi22], developed the idea further and professionalized the concepts. All these tools effectively ease the profiling for non-expert users, but since they do not offer any metadata management features, they effectively shifted the problem from complicated-to-operate data to complicated-to-operate metadata.

A promising approach to the metadata management concern is to store the discovered metadata in the form of data profiles in a database. In this way, users can issue SQL queries to find, join, and filter the metadata according to their specific application needs. A practical implementation of this idea, which works nicely with Metanome, is the metadata management system *Metacrate* [Kr17a]. Metacrate proposes effective, relational storage formats for various types of metadata, and SQL as a flexible and generic query language. Another metadata store that focuses on statistical metadata rather than structural metadata is *Splash* [FL10]. Similar to Metacrate, Splash is based on SQL and tries to persist all metadata. The general approach of persisting the metadata, though, comes with various issues: Synchronization of data and metadata is expensive, metadata contains a lot of redundancy, schemata with many types (UCCs, INDs, FDs, ODs, . . .) and relaxations (partial, approximate, conditional, . . .) become incomprehensible, and, most importantly, metadata sets are huge if they are stored in their entirety. Standard SQL also appears to be an unfavorable match for working with metadata, which is why we propose a novel query language and a profiling engine that collects the metadata at query time.

As part of our related work, we also consider the enormous space of business intelligence systems with data profiling capabilities, including IBM InfoSphere, Talent Data Quality, Informatica Data Explorer, Trillium Software Data Profiling, OpenRefine, SAP Business Objects, and many, many more. Apart from the fact that many data profiling features of these tools are still behind state-of-the-art in research, they suffer from the same metadata management and accessibility issues as the whole field of data profiling.

Because metadata statements, and data dependencies in particular, are defined on schema level, a data profiling query language needs to be able to access schema elements. *SchemaSQL* [LSS96] is an SQL extension that already offers these capabilities: The queries can access database-, relation-, and attribute-names, join values with labels, compare schema elements, and alter them. The only schema operation needed for data profiling, though, is referring to attribute lists. For this reason and because SchemaSQL also lacks data profiling features, we create a new SQL-like dialect.

Defining SQL extensions or entirely new query languages to query derived information is not a novelty. In the data mining area, which is closely related to data profiling, various efforts have been made to extend SQL with data mining capabilities. For example, `MINE RULE` is a keyword extension to discover rules [MPC+96; MPC98] and the `profile` function is Splash's extension to extract estimated joint probability density functions [FL10]. Data mining algorithms have also been defined via user-defined functions [OP11] or virtual views [Bl12]. Similar extensions would be possible also for data profiling algorithms, but a query language specifically designed for data profiling is easier to understand, clearer in semantics and result formats, and better to be parsed into data profiling pruning rules.

The SQL-like data mining language *RQL* [Ch17] is a query language for discovering exact, extended and relaxed functional dependencies. It is the closest challenger of our proposal, but it can discover only simple *if-then*-statements and no arbitrary complex metadata constructs with different types of metadata. The extension of RQL to a more comprehensive data profiling language would change not only the language, but also its execution engine significantly. Therefore, we propose a novel, more intuitive query language.

4 Data Dependencies

Throughout the paper, we follow established notations for data profiling [Ab18]: Because these notations consider schemata and data to be ordered (e. g. by their physical order on disk), we use the terms *attribute* and *column*, as well as *record*, *tuple*, and *row* interchangeably. We denote a relational schema as R and instances of R as r. Letters from the start of the alphabet denote attributes ($A, B, C, D, \ldots \in R$) and letters from the end of the alphabet denote attribute lists ($\ldots, W, X, Y, Z \subseteq R$). Attributes in these lists can be accessed via index, e. g., as X_i. For some metadata statements, the order of the attributes in attribute lists is important (e. g. INDs) and for others it is not (e. g. UCCs and FDs). We, therefore, name these lists *column combinations* and let the profiling algorithm infer, based on the type of

dependency, whether a combination needs to be interpreted as a list or a set. The notations $R[X]$ and $t[X]$ denote projections of schema R and tuple t on the attributes X, respectively. With these notations, we define UCCs, FDs, and INDs as follows:

Definition 1 (Unique column combination (UCC)) *Given a schema R with instance r, a* UCC *X with $X \subseteq R$ is* valid *in r, iff $\forall t_i, t_j \in r, i \neq j : t_i[X] \neq t_j[X]$.*

Definition 2 (Functional dependency (FD)) *Given a schema R with instance r, the* FD *$X \rightarrow A$ with $X \subseteq R$ and $A \in R$ is* valid *in r iff $\forall t_i, t_j \in r : t_i[X] = t_j[X] \Rightarrow t_i[A] = t_j[A]$.*

Definition 3 (Inclusion dependency (IND)) *Given the schemata R and S with instances r and s, respectively, the* IND *$R[X] \subseteq S[Y]$ (abbreviated $X \subseteq Y$) with attribute lists $X \subseteq R$ and $Y \subseteq S$, and cardinalities $|X| = |Y|$ is* valid *iff $\forall t_i \in r, \exists t_j \in s : t_i[X] = t_j[Y]$.*

Considering our running example in Tab. 1, we find that, for example, {*Name*, *Sex*} is a UCC, {*Type*}→{*Weak*} is an FD, and {*Location*}⊆{*Title*} is an IND. Because UCCs indicate keys, FDs indicate value associations, and INDs indicate referential integrity, these three dependencies are among the most important metadata statements. For more details on axiomatization, inference rules, and minimality/maximality properties, we refer to [Ab18]. In the context of this paper, it should be sufficient to understand that all profiling-related aspects are pushed down to the profiling engine and/or algorithm(s).

5 Data Profiling Query Language

The *Data Profiling Query Language* (DPQL) is a variant of SQL that follows the popular **`SELECT-FROM-WHERE`** syntax. A central element of this syntax is the *column combination* function `CC()`. This function allows DPQL to access schema elements as values. In the following, we first introduce the `CC()` function and, then, discuss the DPQL query syntax.

5.1 DPQL Column Combination Function

Data profiling is about discovering metadata statements on column combinations. With the *column combination* function `CC()`, the user can refer to these groups of attributes and, then, specify restrictions and connections for them. The parameter list of the `CC()` function is a list of relational attributes from which the column combinations should be drawn. For example, `CC(Pokemon.ID, Pokemon.Size)` describes the following list-based column combinations: {∅, [*Pokemon.ID*], [*Pokemon.Size*], [*Pokemon.ID, Pokemon.Size*], [*Pokemon.Size, Pokemon.ID*]}. We can enumerate these from the list of attributes when considering the power set lattice of these attributes [Ab18]. While the `CC()` function defines

the *origin* of the columns, we later introduce further restrictions on column combinations that filter concrete patterns of specific dependency types. Note that the `CC()` function in DPQL is used as a *declarative* construct to define sets of column combinations, which can be named in the queries. `CC()` provides the context for the data profiling and describes the search space for the profiling algorithms, but it is not supposed to be fully materialized.

The attributes for a `CC()` call can be provided explicitly, collectively via their relations, or as negations; we can also specify concrete column combination sets as literals. Tab. 2 provides an overview of the specification options for column combinations:

Attributes: The most basic call of the `CC()` function lists all relational *attributes* that should be considered for the generation of column combinations. If the parameter contains attributes from different tables, these attributes will also form column combinations. For many types of metadata, such as FDs, UCCs, and INDs, all attributes of a column combination must stem from the same relation and the profiling algorithms will prune the search space accordingly; for some types, such as MDs and DCs, mixed column combinations are needed, though.

Relations: By specifying *relations* in the `CC()` parameter lists, we denote all attributes of the respective relations. This shortcut is well established in the data profiling community, as most data profiling algorithms operate on this abstraction level.

Negations: With *negations*, the user can exclude certain attributes from relations in a `CC()` call. In Tab. 2, we consider all attributes in the `Pokemon` relation and exclude only the `Pokemon.ID` attribute from it. The negation is particularly useful to profile the majority of attributes while excluding certain irrelevant attributes, such as empty, generated, or binary attributes. Note that negative statements supersede positive statements, and a negative statement without a positive relation is redundant.

Literal: Instead of modeling the search space with the `CC()` function, sets of column combinations can also be specified explicitly with a *literal* statement. A literal groups one or multiple column combinations, which are represented as attributes in square brackets, into a set in curly brackets. A literal takes the place of any `CC()` call and can contain arbitrary many column combinations, which the profiling uses exactly as specified. If the literal cannot be parsed into a valid column combination, an error is thrown. The option to provide fixed column combinations is important to ask specific profiling questions, such as *"Where does this foreign-key point to?"* or *"Which attributes functionally depend on this key candidate?"*.

Parameter	Example	Description: All CCs formable with . . .
Attributes	`CC(Pokemon.ID,Pokemon.Size)`	the provided attributes.
Relations	`CC(Pokemon,Teams)`	the attributes of the provided relations.
Negations	`CC(Pokemon,!Pokemon.ID)`	all attributes but the provided exceptions.
Literals	`{[Pokemon.ID,Pokemon.Size], [Trainers.Rank]}`	exactly the two provided column combinations.

Tab. 2: Specification options for column combination sets.

Because the order of column combinations matters for INDs (and some other dependencies), column combinations in DPQL queries are always considered as lists, i. e., the order of attributes in column combinations is meaningful. However, for set-based data dependencies, such as UCCs and FDs, the profiling engine automatically prunes redundant results.

5.2 DPQL Query Syntax

We now introduce the **SELECT**-**FROM**-**WHERE** syntax of DPQL and provide further examples of DPQL queries. The queries use the **CC**() function to reference column combinations (short CCs) and the functions **UCC**(), **FD**(), and **IND**() to specify dependencies (and their interactions); we provide more details on the metadata discovery functions later in Sect. 6.

SELECT The **SELECT** clause defines the column combinations that shall appear in the query's result. Each listed column combination translates into a column in the relational output, and every row in the relational output is a set of column combinations that answers the DPQL query. Column combinations refer to the search spaces defined by the **CC**() calls in the **FROM** clause, and can be renamed with the **AS** keyword. List. 2 shows a DPQL query with a simple **SELECT** clause that selects the left- and right-hand-sides of functional dependencies within the relations Pokemon and Trainers. Result tuples of this query would be ([*Pokemon.Type*], [*Pokemon.Weak*]) or ([*Trainer.Rank*], [*Trainer.Pokecount*]), which represent the FDs *Type*→*Weak* and *Rank*→*Pokecount*. Note that **SELECT** describes a projection on the column combinations defined in the **FROM** clause and, therefore, does not need to list all CCs – if we require only left-hand-sides, we would project on X alone in List. 2.

```
1 SELECT
2   X AS Left, Y AS Right
3 FROM
4   CC(Pokemon,Trainers) X,
5   CC(Pokemon,Trainers) Y
6 WHERE
7   FD(X,Y)
```

List. 2: Find all functional dependencies in the relations Pokemon and Trainers.

FROM The **FROM** clause uses the **CC**() function (or literals) to specify the search space of the profiling. Every **CC**()-defined set of column combinations needs to be named, such that it can be referenced in the **SELECT** and/or **WHERE** clause. The DPQL query in List. 3 demonstrates the discovery of inclusion dependencies in the Pokémon example with two different **CC**() sets. The results of this query contain all X and Y column combination pairs, for which the X values link Pokemon to Y values in either Locations or Teams.

```
1 SELECT
2   X AS Dependent, Y AS Referenced
3 FROM
4   CC(Pokemon) X,
5   CC(Locations,Teams) Y
6 WHERE
7   IND(X,Y)
```

List. 3: Find all inclusion dependencies from the relation Pokemon to either Locations or Teams.

WHERE The **WHERE** clause is a logical filter expression. It defines the metadata patterns that serve a specific application need and follows SQL operator precedence. While the **FROM** clause restricts the data profiling process to certain tables (and attributes), the **WHERE** clause can be used to formulate conditions and metadata patterns that further prune the metadata search space. Handed over to the actual data profiling, these restrictions can greatly reduce runtime and memory consumption. Expressions in the **WHERE** clause are based on data profiling functions that cover different types of metadata statements, such as **UCC()**, **FD()**, and **IND()**, and additional restrictions on column combinations, such as **SIZE()**, **MIN()**, and **MAX()** (more details in Sect. 6). Filter criteria can be linked via **AND** and **OR**, and any valid answer to a DPQL query needs to fulfill the entire **WHERE** clause. An example with a slightly larger **WHERE** clause than before is shown in List. 4: The query asks for all inclusion and functional dependencies that point to unique column combinations. Both ([*Pokemon.Name,Pokemon.Sex*], [*Pokemon.ID*]) and ([*Teams.Pokemon*], [*Pokemon.ID*]) are valid answers to the query, the former being the FD $\{Name,Sex\} \rightarrow \{ID\}$ and the latter the IND $\{Pokemon\} \subseteq \{ID\}$; the result does not differentiate FDs and INDs, but the way this query is issued (via **OR**) indicates that this information is irrelevant for the application.

```
1 SELECT
2   X Determinant, Y AS Unique
3 FROM
4   CC(Teams,Trainers) X,
5   CC(Pokemon) Y
6 WHERE
7   UCC(Y)
8   AND (IND(X,Y) OR FD(X,Y))
```

List. 4: Find all unique column combinations that are a target of a functional or inclusion dependency.

5.3 DPQL Result Format

In contrast to SQL, which queries database records, DPQL extracts statements about the schemata, i. e., combinations of attributes and their interactions. These schema statements are compositions of column combinations, which introduce special challenges for the output format. To understand these challenges and our design decisions for overcoming them, we first describe the straightforward case and address the complicated situations afterwards.

Basic DPQL Results

A DPQL query returns a result in relational format: The **SELECT** clause determines the schema of the result table by turning every provided column combination variable, which is a **CC()** call, into a relational attribute. The name of each result attribute is equal to the column combination's variable name or, if provided, its AS-alias. Each row in the result relation is a valid response to the DPQL query; structurally, a response row is a set of column combinations, which is a set of attribute lists. For example, Tab. 3 lists the results of

our introductory foreign-key example (see List. 1). Each of the three result tuples describes a valid foreign-key candidate according to the specified filter criteria.

ForeignKey	Key
[Teams.Pokemon]	[Pokemon.ID]
[Teams.Trainer]	[Trainers.Name]
[Pokemon.Locations]	[Location.Name]

Tab. 3: The result table for the DPQL query from List. 1 with column combinations `X` and `Y`.

Finding the most effective strategy for obtaining DPQL query results will be subject to extensive future research, but a possible way of processing the foreign-key query with state-of-the-art algorithms is as follows: We first discover all INDs, which are $\{Pokemon\} \subseteq \{ID\}$, $\{Trainer\} \subseteq \{Firstname\}$, $\{Location\} \subseteq \{Title\}$, and $\{Strong\} \subseteq \{Biome\}$; then we discover all UCCs, which are $\{ID\}$, $\{Weight\}$, $\{Name, Sex\}$, $\{Name, Size\}$, $\{Title\}$, $\{Firstname\}$, $\{Rank\}$, $\{Pokecount\}$; after obtaining both type-specific profiling results, we intersect the IND right-hand-sides and the UCCs with a subset-aware comparison (i. e., if any subset of a right-hand-side is a UCC, the IND-UCC-pair is valid); this leaves us with the INDs shown in Tab. 3; finally, we apply the size and origin filters, which do not change the results. This process demonstrates that DPQL queries can be answered automatically with state-of-the-art profiling technology, although this way of processing is terribly expensive.

Normalization

While the foreign-key example is an ideal case of a result table, the relational result structure for data profiling statements has a major size issue when it comes to more complex result sets: Because every row in the table represents a unique valid result, DPQL queries with more than one dependency in the output column combinations generate a lot of redundancy. For illustration purposes, consider the DPQL query in List. 5 that aims to profile all unique column combinations in the relations `Pokemon` and `Trainers`. Because these UCCs are associated with two independent `CC()` calls, every combination of a `Pokemon` UCC and a `Trainers` UCC is a valid answer to the query. We show the list of results in Tab. 4.

```
1  SELECT
2    X AS PokemonUCCs,
3    Y AS TrainersUCCs
4  FROM
5    CC(Pokemon) X,
6    CC(Trainers) Y
7  WHERE
8    UCC(X)
9    AND UCC(Y)
```

List. 5: Find all UCCs in `Pokemon` and `Trainers`.

PokemonUCCs	TrainersUCCs
$\{ID\}$	$\{Firstname\}$
$\{ID\}$	$\{Rank\}$
$\{ID\}$	$\{Pokecount\}$
$\{Name, Sex\}$	$\{Firstname\}$
$\{Name, Sex\}$	$\{Rank\}$
$\{Name, Sex\}$	$\{Pokecount\}$
...	...

Tab. 4: UCCs of List. 5 in one result.

The redundancy that we find in this result table can be described as a *join* or *multivalued dependency* [Ab18]. The redundancy introduced with such dependencies grows quadratically with increasing data volume, which is problematic considering that data profiling result sets grow exponentially with the input schema sizes – even the increased pruning capabilities of DPQL cannot resolve this general issue.

```
1 SELECT
2   X, Y, Z
3 FROM
4   CC(Pokemon,Trainers,Teams) X,
5   CC(Pokemon,Trainers,Teams) Y,
6   CC(Pokemon,Trainers,Teams) Z
7 WHERE
8   IND(X,Y)
9   AND FD(Y,Z)
```

List. 6: Find all INDs that point to FDs.

In a first solution attempt, we might reject DPQL queries with non-correlated column combinations, but the redundancy issue also exists for properly correlated column combinations: The DPQL query in List. 6 asks for all inclusion dependencies that point to functional dependencies; the result should list both the INDs and FDs. Now, if an IND points to multiple FDs or an FD is the target of multiple INDs, we generate duplicate, i. e., redundant IND and FD outputs, respectively. So, we again observe redundancy from *join* or *multivalued dependencies* in the results. To resolve these dependency-caused redundancies, relational database theory suggests schema normalization. For this reason, we propose normalized outputs for DPQL queries and, hence, potentially multiple result tables. The algorithm for creating these tables is shown in Algorithm 1. It creates a table for every pair of column combinations that appears together in at least one binary dependency, such as an IND or FD (Lines 3-6); then, it creates separate tables for individual column combinations that are not linked to other column combinations (Lines 7-10). In this way, DPQL results can be represented without their inherent redundancy.

Algorithm 1 Creation of the normalized DPQL result schema

```
1:  procedure createResultSchema(dpqlQuery)
2:      resultSchema ← ∅
3:      for every binary dependency D(X,Y) in the WHERE clause of the dpqlQuery do
4:          if there is no table T(X,Y) or T(Y,X) with D's two CCs X and Y in resultSchema then
5:              if both X and Y are selected in the SELECT clause then
6:                  create the table T(X,Y) and store it in resultSchema
7:      for every CC Z in the SELECT clause of the dpqlQuery do
8:          if there is no table T(Z), T(X,Z) or T(Z,X) with this CC in resultSchema then
9:              if Z is selected in the SELECT clause then
10:                 create the table T(Z) and store it in resultSchema
        return resultSchema
```

With normalization, the output of the DPQL query in List. 1 remains one table with schema {[ForeignKey, Key]}. The output of the DPQL query in List. 5, though, becomes {[PokemonUCCs], [TrainerUCCs]} and the output of the DPQL query in List. 6 becomes {[X, Y], [Y, Z]}. To reconstruct the single, not-normalized result table or to read a full result

row, we simply join the individual result tables on common column combinations (e. g. `Y` for the query in List. 6); the reconstruction of unrelated result tables requires a cross join.

In summary, we recommend normalizing DPQL query results for result compaction. The non-normalized, single relation results can always be obtained by joining the result tables, which is useful, for instance, if a DPQL query is embedded into an SQL query.

Extension Columns

Many data profiling results, such as functional and inclusion dependencies, simply mark a special relation between column combinations. These relations can be expressed with the (normalized) relational result format on column combinations. However, the data profiling toolbox offers a plethora of metadata statements, relaxations, and conditions that provide additional information about the properties of a column combination or column combination relationship. Therefore, we propose to extend DPQL result schemata dynamically with additional columns that store well-defined, metadata-dependent information. The rationale here is simple: If a DPQL function, such as **`UCC()`**, **`FD()`**, or **`IND()`**, extracts more than a relationship of column combinations, the DPQL engine adds a DPQL function-specific, additional column to the output schema. In theory, we can assume that the union of all possible extension columns is implicitly present in all DPQL query results and the fields are `NULL` by default, but in practice, these columns should be hidden if they are empty. We now briefly introduce some basic extension columns for popular data profiling metrics (see Tab. 5). It is worth noting that the table is incomplete and needs to be extended in the development process of the data profiling engine:

Name	Property	Type	Values
`Approximate`	Relaxation	*boolean*	`true` if validity is not certain
`Partial`	Relaxation	*float*	*Fraction* of records that fulfill the statement
`Conditional`	Relaxation	*string*	*Condition* for defining the statement's scope
`Minimum`	Statistic	<target type>	*Minimum value* of the target CC
`Maximum`	Statistic	<target type>	*Maximum value* of the target CC
`Histogram`	Statistic	*string*	*Value distribution* in the target CC
`Denial`	Special	*string*	*Denial constraint expression* on the target CC
`Matching`	Special	*string*	*Matching dependency expression* on the target CC
`Order`	Special	*string*	*Order dependency expression* on the target CC

Tab. 5: Extension columns that relax metadata statements or belong to special metadata statements.

Relaxation: Any relational metadata statement can be relaxed in different ways [CDP16]: We can, i. a., make the statement *approximate* signaling that the statement's validity is not guaranteed, *partial* to restrict the statement's validity to a certain percentage of records, or *conditional* to tie the statement's validity to specific constraints. Such relaxations have been implemented for many data profiling algorithms and are required by many data profiling applications. With the extension columns, we can also return them in DPQL results.

Statistic: Data profiling often targets basic statistics, such as a column combination's min-, max-, avg-, or median-values, `NULL`-counts, data types, histograms, lengths- and size-measurements, or frequent item sets. The results of such profiling tasks can easily be stored in extension columns.

Special: Some special data dependencies, which are frequently extensions of functional dependencies, can describe more complex relationships between column combinations. This includes, for example, *matching dependencies* (MDs) [Fa08], *order dependencies* (ODs) [GH83], and *denial constraints* (DCs) [Be11]. The extra information hidden in these relationships can be similarity functions and thresholds (see MDs), order directions (see ODs), or entire first-order logic statements (see DCs). While the column combinations of these dependencies are stored in the normal CC-columns of the relational DPQL result sets, the engine adds extension columns for the dependency-specific details.

As shown in Tab. 5, extension columns can be typed to improve their accessibility for applications. The standard for basically all existing data profiling tools and algorithms is to provide all results as strings; therefore, typed extension columns can add some additional information. The concept of extension columns adapts well to the dynamic nature of data profiling, as it allows a flexible combination of properties. For example, Tab. 6 shows the result of a DPQL query that discovered all *partial, conditional matching dependencies*. Although no existing data profiling algorithm can actually discover such dependencies, there is certainly a practical use for them in, for instance, data integration. The result table lists the two column combinations of this dependency (`Pokemon` and `PoMos`), the matching dependency condition (`Matching`), and the two relaxations (`Partial` and `Conditional`) in one relational table. Each entry in this result relation – in this case, only one entry – is an answer to the discovery query. The shown example describes the matching dependency $\{Pokemon[Name] \approx_{Jac,0.92} PoMos[ID]\} \rightarrow \{Pokemon[Sex] \approx_{Lev,1.0} PoMos[Gender]\}$, which is true for 97% of the tuples under the condition $\{Weight > 0 \wedge Name \neq \text{'Mewtwo'}\}$.

Pokemon	PoMos	Partial	Conditional	Matching
[`Pokemon.Name`, `Pokemon.Sex`]	[`PoMos.ID`, `PoMos.Gender`]	0.97	$\{Weight > 0 \wedge$ $Name \neq$ 'Mewtwo'$\}$	[(*Jaccard*, 0.92), (*Levenstein*, 1.0)]

Tab. 6: Result of a DPQL query that discovered all *partial, conditional matching dependencies*.

6 DPQL Functions

The purpose of DPQL is to restrict data profiling activities and their results in such a way that only truly needed metadata is delivered to the application. To filter and combine the column combinations purposively, DPQL offers a variety of functions that are applied in the **`WHERE`** clause. This section introduces the most important DPQL functions.

Metadata Discovery Functions

Metadata discovery functions are the core of DPQL. These functions represent the data profiling services that were traditionally implemented as separate algorithms. In this paper, we already used three metadata discovery functions in the various examples, namely **UCC**(<CC>), **FD**(<CC>,<CC>), and **IND**(<CC>,<CC>) for unique column combinations, functional dependencies, and inclusion dependencies, respectively. The reading order of column combinations in dependency functions is from left to right: First, the dependent/cause/included part, then the referenced/effect/containing part. Throughout this paper, we showed example queries with UCCs, FDs, and INDs, but thanks to extension columns (see Sect. 5.3), the functional concept extends seamlessly to all other types of metadata, such as order dependencies (**OD**(<CC>,<CC>)), matching dependencies (**MD**(<CC>,<CC>)), or denial constraints (**DC**(<CC>,<CC>)). Short DPQL queries with a single metadata function call can be used as an interface for existing data profiling algorithms, but the strength of DPQL lies in the combination of metadata functions. With a good understanding of the discovery functions, a query engine can combine multiple functions into a single, holistic profiling task and, then, optimize execution orders, share intermediate results for additional search space pruning, and re-use temporary data structures.

To support possibly all variations of data profiling, we need a standard to pass optional configuration parameters to metadata functions. For example, suppose we want to relax a dependency as discussed in Sect. 5.3 or force the declarative query into a certain execution strategy, which is to bypass the automatic query optimizer. In such cases, we can specify these objectives as parameters in the metadata functions. Passing parameters to DPQL functions is done via named parameters with the <parameter>=<value> syntax. This syntax ensures that parameter specifications are order-invariant and differ from column combination specifications. To enforce, for instance, a partial functional dependency that has at least a coverage of 95% of the tuples, we could write **FD**(X, Y, partial=0.95) or to force the engine to discover approximate INDs with the FAIDA method [Kr17b], we write **IND**(X, Y, approximate=true, method='*FAIDA*'). We acknowledge that this is not the most idiomatic approach for a declarative query, but it addresses the variety of data profiling demands and the fact that data profiling is still a quickly evolving discipline.

Result Restriction Functions

CARDINALITY: The **CARDINALITY**(<CC>) function counts the number of distinct values in a column combination. It can be used together with numeric comparators (i. e., <,<=,=,>=,>) in filter statements to restrict valid column combinations to those that have a certain (minimum or maximum) cardinality. We have seen this function already in the query of Tab. 1, where we demanded foreign-keys to hold at least two different values.

SIZE: The `SIZE(<CC>)` function can be used to restrict the number of attributes in a column combination to a fixed, minimum or maximum size. Recall that the `CC()` function creates a (virtual) power-set-shaped lattice of column combinations of various sizes. With the `SIZE()` function and a numeric comparator (i. e., `<,<=,=,>=,>`), we can bind the sizes of certain column combinations to numeric values or the sizes of other column combinations (see List. 7). In this way, `SIZE()` effectively prunes the search space with a simple criterion that many profiling algorithms can already process.

```
1  SELECT
2    X, Y
3  FROM
4    CC(Pokemon) X,
5    CC(Pokemon) Y,
6    CC(Teams) Z
7  WHERE
8    FD(X,Y) AND SIZE(X) < 3
9    AND IND(Y,Z) AND SIZE(Y) = SIZE(X)
```

List. 7: Find FDs with less than three attributes in `Pokemon` that functionally determine an IND of same size into the `Teams` relation.

MIN and **MAX:** To keep metadata results concise, data profiling algorithms discover only result sets of minimal (UCCs, FDs, MDs, ...) or maximal (INDs) dependencies; via dependency axioms, all non-enumerated dependencies can be derived from these sets. Now that we combine dependencies into patterns via DPQL, minimality/maximality properties are less clear. Consider, for example, the query in List. 8. The answer to this query might be an `(X,Y)`-tuple, where neither `Y` is a minimal UCC nor `X`⊆`Y` is a maximal IND – this is what makes the traditional application of data profiling results such a hard task. To lead the results in a useful and clear direction, we can define specific CCs to be `MIN(<CC>)` or `MAX(<CC>)`. Minimizing means that we cannot remove a single attribute from the CC without violating the entire query result; maximizing means that we cannot add any further attribute. By default, all profiling functions but `IND()` produce minimal results; `IND()` and combinations with this function produce maximal results.

```
1  SELECT
2    X AS ForeignKey, Y AS Key
3  FROM
4    CC(Pokemon,Teams,Trainers) X,
5    CC(Pokemon,Teams,Trainers) Y
6  WHERE
7    UCC(Y) AND IND(X,Y) AND MIN(X)
```

List. 8: Find foreign-key candidates with attribute sets of minimal size – effectively unary INDs.

CONTAINS: The DPQL function `CONTAINS(<CC>,<CC>)` specifies that in every valid result, the first column combination contains all attributes of the second column combination. To understand the usefulness of this function, again consider the example query in List. 8. Assume we want, as an output of this query, not the actual IND-UCC-pair that answers the query but instead the maximal IND and minimal UCC that frame these solutions. Hence, we specify three outputs `X`, `Y`, and `Z` constrained to `UCC(X)`, `IND(Y,Z)`, and `CONTAINS(Y,X)`, then `X`⊆`Y` (referring to attributes here; not INDs!) with minimal `X` and maximal `Y` and `Z` column combinations.

SPLIT and **PAIR:** Column combinations in DPQL query results are, by default, unrelated unless some metadata function connects them. Occasionally, however, we want to filter results such that certain column combinations in a result (= a row in the result table) are either *paired* (= potentially different attribute lists but from the same relation) or *split* (= different attribute lists from different relations). The `PAIR(<CC>,<CC>)` and `SPLIT(<CC>,<CC>)` functions allow the user to specify these requirements in a DPQL query. We recall that in our introductory example on foreign-key discovery (see List. 1) the source and target columns should stem from different relations; this was ensured with the `SPLIT()` function. If we would like to discover, for example, redundant keys (= more than one UCC in the same table) in multiple relations (see List. 9), we need the `PAIR()` function to co-locate `X` and `Y`. Note that `PAIR()` and `SPLIT()` are commutative operations, so that e. g. `PAIR(X,Y) = PAIR(Y,X)`.

```
1  SELECT
2    X, Y
3  FROM
4    CC(Pokemon,Teams,Trainers) X,
5    CC(Pokemon,Teams,Trainers) Y
6  WHERE
7    UCC(X) AND UCC(Y) AND PAIR(X,Y)
```

List. 9: Find redundant keys in multiple relations.

7 DPQL in Practical Applications

Data profiling has many applications in data management and data analytics. To evaluate our data profiling query language, we selected a few representative scenarios from different applications to showcase the implementation of their profiling activities in our novel dialect.

Data linkage: Our foreign-key discovery example from List. 1 has been drawn from a data engineering task that aims to connect previously unconnected datasets or datasets for which the foreign-key relationships have been lost. The discovered combinations of INDs and UCCs present an application with structurally valid constraint candidates.

Data cleaning: Metadata is an important asset for error detection and correction. A cleaning system could, for instance, issue the DPQL query in List. 10 to discover partial INDs, UCCs, and FDs in the `Pokemon` relation. The system would then check the results for meaningful but not 100% correct results. We should, for example, find the IND *Location* $\subseteq$ *Title*, the UCC {*Name, Sex*}, and the FD *Type* $\rightarrow$ *Weak*. If one of these is indeed partial and not exact, we can use the dependency to identify and possibly correct the erroneous records [MA20].

```
1  SELECT
2    W, X, Y, Z
3  FROM
4    CC(Pokemon) V, CC(Pokemon) W,
5    CC(Pokemon) X, CC(Pokemon) Y,
6    CC(Locations) Z
7  WHERE
8    IND(V,Z,partial=0.8)
9    AND UCC(W,partial=0.95)
10   AND FD(X,Y,partial=0.90)
```

List. 10: Find partial dependencies in the `Pokemon` relation for error detection.

Query optimization: Research on optimizing SQL queries with profileable metadata has generated many approaches, ranging from various query rewriting strategies over physical execution optimization techniques to cost-based query plan rewriting rules [Ko22]. For illustration purposes, consider an example of a distinct semi-join filter, which is an SQL query of the form `SELECT DISTINCT Name, Sex FROM Pokemon WHERE Location IN {SELECT Title FROM Locations};` The DPQL query in List. 11 checks if $\{Name, Sex\}$ is unique to remove the `DISTINCT` operator and if $Location \subseteq Title$ is an IND to remove the entire `WHERE` clause.

```
1  SELECT
2    UCC_C, IND_L, IND_R
3  FROM
4    CC(Pokemon.Name,Pokemon.Sex) UCC_C,
5    {[Pokemon.Location]} AS IND_L,
6    {[Locations.Title]} AS IND_R
7  WHERE
8    UCC(UCC_C) AND IND(IND_L,IND_R)
```

List. 11: Find a specific UCC and IND.

Data integration: Schema matching is an integral part of data integration. One flavor of schema matching are structure-based approaches [RB01]. The partial conditional matching dependency that we discussed in Sect. 5.3 is such a structure that describes matching attributes (the MD) with some failure tolerance and context information (the conditional properties). To discover the partial conditional MDs between the `Pokemon` and `PoMos` relations, we can use the DPQL query in List. 12.

```
1  SELECT
2    X, Y
3  FROM
4    CC(Pokemon,PoMos) X,
5    CC(Pokemon,PoMos) Y
6  WHERE
7    MD(X,Y,partial=0.95,
8       conditional=true)
9    AND SPLIT(X,Y)
```

List. 12: Find partial conditional MDs.

Data exploration: A look at the metadata of a relational dataset often helps to understand its structure and implicit logic better. Because data often comes without metadata, data profiling is conducted to gather possibly many insights from a given instance. The DPQL query in List. 13 does exactly this: It collects all UCCs, FDs, and INDs that are true in our Pokémon example. The normalized output presents the results in three tables – one for each dependency.

```
1  SELECT
2    UCC_C, FD_L, FD_R, IND_L, IND_R
3  FROM
4    CC(P,L,T,T) UCC_C,
5    CC(P,L,T,T) FD_L, CC(P,L,T,T) FD_R,
6    CC(P,L,T,T) IND_L, CC(P,L,T,T) IND_R
7  WHERE
8    UCC(UCC_C) AND FD(FD_L,FD_R)
9    AND IND(IND_L,IND_R)
```

List. 13: Find UCCs, FDs and INDs in all tables; table names in CC() calls were shortened for brevity.

8 Future Work

The data profiling query language (DPQL) that we introduced in this paper is an essential building block for a new generation of data profiling systems. For its practical implementation, we envision a database-like system that covers all standard query processing components. The setup of this system, however, is more like a *virtually integrated database* [DHI12] or, in modern terms, a *DataLakehouse* [Ar21], because the system answers metadata questions in a virtual fashion, across potentially multiple datasets, and without manipulating the data itself. Due to the size and complexity of this system, we expect that DPQL will spark innovative research on at least the following components:

Query parser: The DPQL queries require a parsing component that translates them into logical (and physical) execution plans. We assume that DPQL can be combined with SQL, but this imposes interesting parsing challenges. Another challenge for the parser (and all other components) is that further iterations of DPQL must be able to introduce new features, such as additional metadata types, metadata properties, or filter functions, because data profiling – in contrast to relational query processing – is a still evolving area.

Query optimizer: Despite their similarities with SQL queries, DPQL queries translate into quite different execution plans, for which other optimization rules apply. For query optimization, novel approaches for indexing, caching, query rewriting, operator ordering etc. need to be found. Effective approaches for selecting the most efficient execution strategy (e. g. UCCs first, INDs first, or both at the same time?) and automatically inferring empty results (e. g. from `CC(Pokemon,!Pokemon)` or `SIZE(CC(Team))>3`) are crucial for the system.

Query execution engine: The actual data profiling might change significantly given the new application-specific pruning rules and the potential of holistically combining profiling runs. Given the many existing profiling algorithms (and new techniques of the future), research will need to investigate which algorithms to combine, how to combine algorithms, and how to integrate them into one system. Considering the comprehensive amount of metadata types and discovery flavors, we expect a lot of future research on the actual query processing.

9 Summary

In this paper, we proposed DPQL, a declarative query language for the discovery of data dependencies and other metadata statements. DPQL is the first uniform data profiling language and an essential building block for a new generation of data profiling systems. The SQL-like language relieves data scientists from deploying complex profiling algorithms, and it renders most of the expensive and difficult post-processing efforts obsolete. Due to the increased filter- and pruning-capabilities, we expect significant efficiency gains for DPQL-based data profiling activities. With DPQL, we started to close the gap between data profiling results and actual applications needs; now, much research is needed for the technical design of the language and its profiling capabilities.

References

[Ab18] Abedjan, Z.; Golab, L.; Naumann, F.; Papenbrock, T.: Data Profiling: Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

[Ar21] Armbrust, M.; Ghodsi, A.; Xin, R.; Zaharia, M.: Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In: Proceedings of the Conference on Innovative Data Systems Research (CIDR). 2021.

[Be11] Bertossi, L. E.: Database Repairing and Consistent Query Answering. Morgan & Claypool Publishers, 2011.

[Bi20] Birnick, J.; Bläsius, T.; Friedrich, T.; Naumann, F.; Papenbrock, T.; Schirneck, M.: Hitting Set Enumeration with Partial Information for Unique Column Combination Discovery. Proceedings of the VLDB Endowment 13/12, pp. 2270–2283, 2020.

[BKN17] Bleifuß, T.; Kruse, S.; Naumann, F.: Efficient Denial Constraint Discovery with Hydra. Proceedings of the VLDB Endowment 11/3, pp. 311–323, 2017.

[Bl12] Blockeel, H.; Calders, T.; Fromont, É.; Goethals, B.; Prado, A.; Robardet, C.: An inductive database system based on virtual mining views. Data Mining and Knowledge Discovery 24/1, pp. 247–287, 2012.

[Ca21a] Caruccio, L.; Cirillo, S.; Deufemia, V.; Polese, G.: Efficient Discovery of Functional Dependencies from Incremental Databases. In: International Conference on Information Integration and Web Intelligence. Pp. 400–409, 2021.

[Ca21b] Caruccio, L.; Deufemia, V.; Naumann, F.; Polese, G.: Discovering Relaxed Functional Dependencies Based on Multi-Attribute Dominance. IEEE Transactions on Knowledge and Data Engineering (TKDE) 33/9, pp. 3212–3228, 2021.

[CDP16] Caruccio, L.; Deufemia, V.; Polese, G.: Relaxed Functional Dependencies - A Survey of Approaches. IEEE Transactions on Knowledge and Data Engineering (TKDE) 28/1, pp. 147–165, 2016.

[Ch17] Chardin, B.; Coquery, E.; Pailloux, M.; Petit, J.-M.: RQL: a query language for rule discovery in databases. Theoretical Computer Science 658/, pp. 357–374, 2017.

[Ch19] Chen, H.; Jajodia, S.; Liu, J.; Park, N.; Sokolov, V.; Subrahmanian, V. S.: FakeTables: Using GANs to Generate Functional Dependency Preserving Tables with Bounded Real Data. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). International Joint Conferences on Artificial Intelligence Organization, pp. 2074–2080, 2019.

[De22] Desbordante: Open-source data profiling tool, 2022, URL: `https://desbordante.unidata-platform.ru/`, visited on: 09/19/2022.

[DHI12] Doan, A.; Halevy, A.; Ives, Z.: Principles of data integration. Elsevier, 2012.

[DR02] Do, H.-H.; Rahm, E.: COMA – A System for flexible combination of schema matching approaches. In: Proceedings of the International Conference on Very Large Databases (VLDB). Pp. 610–621, 2002.

[Dü19] Dürsch, F.; Stebner, A.; Windheuser, F.; Fischer, M.; Friedrich, T.; Strelow, N.; Bleifuß, T.; Harmouch, H.; Jiang, L.; Papenbrock, T.; Naumann, F.: 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.

[Eh16] Ehrlich, J.; Roick, M.; Schulze, L.; Zwiener, J.; Papenbrock, T.; Naumann, F.: Holistic Data Profiling: Simultaneous Discovery of Various Metadata. In: Proceedings of the International Conference on Extending Database Technology (EDBT). Pp. 305–316, 2016.

[Fa08] Fan, W.: Dependencies Revisited for Improving Data Quality. In: Proceedings of the Symposium on Principles of Database Systems (PODS). Pp. 159–170, 2008.

[Fe18] Fernandez, R. C.; Abedjan, Z.; Koko, F.; andSamuel Madden, G. Y.; Stonebraker, M.: AURUM: A data discovery system. In: Proceedings of the International Conference on Data Engineering (ICDE). 2018.

[FL10] Fang, L.; LeFevre, K.: Splash: ad-hoc querying of data and statistical models. In: Proceedings of the International Conference on Extending Database Technology (EDBT). Pp. 275–286, 2010.

[GH83] Ginsburg, S.; Hull, R.: Order dependency in the relational model. Theoretical Computer Science 26/1–2, pp. 149–195, 1983.

[HL18] Hannula, M.; Link, S.: On the interaction of functional and inclusion dependencies with independence atoms. In: International Conference on Database Systems for Advanced Applications. Pp. 353–369, 2018.

[HN17] Harmouch, H.; Naumann, F.: Cardinality Estimation: An Experimental Survey. Proceedings of the VLDB Endowment 11/4, pp. 499–512, 2017.

[HPN21] Harmouch, H.; Papenbrock, T.; Naumann, F.: Relational Header Discovery using Similarity Search in a Table Corpus. In: Proceedings of the International Conference on Data Engineering (ICDE). Pp. 444–455, 2021.

[Hu99] Huhtala, Y.; Kärkkäinen, J.; Porkka, P.; Toivonen, H.: TANE: An efficient algorithm for discovering functional and approximate dependencies. The Computer Journal 42/2, pp. 100–111, 1999.

[IC15] Ilyas, I. F.; Chu, X.: Trends in Cleaning Relational Data: Consistency and Deduplication. Foundations and Trends in Databases 5/4, pp. 281–393, 2015.

[Ko22] Kossmann, J.; Lindner, D.; Naumann, F.; Papenbrock, T.: Workload-driven, lazy discovery of data dependencies for query optimization. In: Proceedings of the Conference on Innovative Data Systems Research (CIDR). 2022.

[KPN20] Koumarelas, l.; Papenbrock, T.; Naumann, F.: MDedup: Duplicate Detection with Matching Dependencies. 13/5, pp. 712–725, 2020.

[KPN22] Kossmann, J.; Papenbrock, T.; Naumann, F.: Data dependencies for query optimization: a survey. The VLDB Journal 31/1, pp. 1–22, 2022.

[Kr16] Kruse, S.; Papenbrock, T.; Harmouch, H.; Naumann, F.: Data Anamnesis: Admitting Raw Data into an Organization. IEEE Data Engineering Bulletin 39/2, pp. 8–20, 2016.

[Kr17a] Kruse, S.; Hahn, D.; Walter, M.; Naumann, F.: Metacrate: Organize and analyze millions of data profiles. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM). Pp. 2483–2486, 2017.

[Kr17b] Kruse, S.; Papenbrock, T.; Dullweber, C.; Finke, M.; Hegner, M.; Zabel, M.; Zoellner, C.; Naumann, F.: Fast Approximate Discovery of Inclusion Dependencies. In: Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW). Pp. 207–226, 2017.

[Li20] Livshits, E.; Heidari, A.; Ilyas, I. F.; Kimelfeld, B.: Approximate Denial Constraints. Proceedings of the VLDB Endowment 13/10, pp. 1682–1695, 2020.

[LSS96] Lakshmanan, L. V. S.; Sadri, F.; Subramanian, I. N.: SchemaSQL - A Language for Interoperability in Relational Multi-Database Systems. In: Proceedings of the VLDB Endowment. Pp. 239–250, 1996.

[MA20] Mahdavi, M.; Abedjan, Z.: Baran: Effective Error Correction via a Unified Context Representation and Transfer Learning. Proceedings of the VLDB Endowment 13/12, pp. 1948–1961, 2020.

[MPC+96] Meo, R.; Psaila, G.; Ceri, S., et al.: A new SQL-like operator for mining association rules. In: Proceedings of the International Conference on Very Large Databases (VLDB). Vol. 96, pp. 122–133, 1996.

[MPC98] Meo, R.; Psaila, G.; Ceri, S.: An extension to SQL for mining association rules. Data Mining and Knowledge Discovery 2/2, pp. 195–224, 1998.

[OP11] Ordonez, C.; Pitchaimalai, S. K.: One-pass data mining algorithms in a DBMS with UDFs. In: Proceedings of the International Conference on Management of Data (SIGMOD). Pp. 1217–1220, 2011.

[Pa00] Paulley, G. N.: Exploiting Functional Dependence in Query Optimization, tech. rep., University of Waterloo, 2000.

[Pa15a] Papenbrock, T.; Bergmann, T.; Finke, M.; Zwiener, J.; Naumann, F.: Data Profiling with Metanome. Proceedings of the VLDB Endowment 8/12, pp. 1860–1863, 2015.

[Pa15b] Papenbrock, T.; Ehrlich, J.; Marten, J.; Neubert, T.; Rudolph, J.-P.; Schönberg, M.; Zwiener, J.; Naumann, F.: Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. Proceedings of the VLDB Endowment 8/10, pp. 1082–1093, 2015.

[Pa15c] Papenbrock, T.; Kruse, S.; Quiané-Ruiz, J.-A.; Naumann, F.: Divide & Conquer-based Inclusion Dependency Discovery. Proceedings of the VLDB Endowment 8/7, pp. 774–785, 2015.

[PAN21] Pena, E. H. M.; de Almeida, E. C.; Naumann, F.: Fast Detection of Denial Constraint Violations. Proceedings of the VLDB Endowment 15/4, pp. 859–871, 2021.

[PN16] 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.

[RB01] Rahm, E.; Bernstein, P. A.: A survey of approaches to automatic schema matching. Proceedings of the VLDB Endowment 10/4, pp. 334–350, 2001.

[Ro09] Rostin, A.; Albrecht, O.; Bauckmann, J.; Naumann, F.; Leser, U.: A Machine Learning Approach to Foreign Key Discovery. In: Proceedings of the ACM Workshop on the Web and Databases (WebDB). 2009.

[Sc20] Schirmer, P.; Papenbrock, T.; Koumarelas, I.; Naumann, F.: Efficient Discovery of Matching Dependencies. ACM Transactions on Database Systems (TODS) 45/3, pp. 1–33, 2020.

[SGI19] Saxena, H.; Golab, L.; Ilyas, I. F.: Distributed Implementations of Dependency Discovery Algorithms. Proceedings of the VLDB Endowment 12/11, pp. 1624–1636, 2019.

[SP22] Schmidl, S.; Papenbrock, T.: Efficient Distributed Discovery of Bidirectional Order Dependencies. VLDB Journal 31/1, pp. 49–74, 2022.

[Sz17] Szlichta, J.; Godfrey, P.; Golab, L.; Kargar, M.; Srivastava, D.: Effective and Complete Discovery of Order Dependencies via Set-based Axiomatization. Proceedings of the VLDB Endowment/, 2017.

[VA18] Visengeriyeva, L.; Abedjan, Z.: Metadata-Driven Error Detection. In: Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM). 2018.

[Vi22] Viadotto: Make your data profitable with our next-gen data profiling tools, 2022, URL: https://www.viadotto.tech/, visited on: 09/19/2022.

[Wa17] Wang, Y.; Song, S.; Chen, L.; Yu, J. X.; Cheng, H.: Discovering conditional matching rules. ACM Transactions on Knowledge Discovery from Data 11/4, pp. 1–38, 2017.

[WHL21] Wei, Z.; Hartmann, S.; Link, S.: Algorithms for the discovery of embedded functional dependencies. VLDB Journal 30/6, pp. 1069–1093, 2021.

[WLL19] Wei, Z.; Leck, U.; Link, S.: Discovery and Ranking of Embedded Uniqueness Constraints. Proceedings of the VLDB Endowment 12/13, pp. 2339–2352, 2019.

[Xi22] Xiao, R.; Yuan, Y.; Tan, Z.; Ma, S.; Wang, W.: Dynamic Functional Dependency Discovery with Dynamic Hitting Set Enumeration. In: Proceedings of the International Conference on Data Engineering (ICDE). Vol. 1. 1, pp. 286–298, 2022.

[Zh10] Zhang, M.; Hadjieleftheriou, M.; Ooi, B. C.; Procopiuc, C. M.; Srivastava, D.: On Multi-column Foreign Key Discovery. Proceedings of the VLDB Endowment 3/1-2, pp. 805–814, 2010.