Discovery and Application of Functional Dependencies in Conjunctive Query Mining

Bart Goethals[1], Dominique Laurent[2], Wim Le Page[1]

[1] University of Antwerp, Dept of Mathematics and Computer Science B-2020 Antwerp
[2] ETIS-CNRS-ENSEA-Université de Cergy-Pontoise F-95000 Cergy-Pontoise

Abstract. We present an algorithm for mining frequent queries in arbitrary relational databases, over which functional dependencies are assumed. Building upon previous results, we restrict to the simple, but appealing subclass of simple conjunctive queries. The proposed algorithm makes use of the functional dependencies of the database to optimise the generation of queries and prune redundant queries. Furthermore, our algorithm is capable of detecting previously unknown functional dependencies that hold on the database relations as well as on joins of relations. These detected dependencies are subsequently used to prune redundant queries. We propose an efficient database-oriented implementation of our algorithm using SQL, and provide several promising experimental results.

1 Introduction

The discovery of recurring patterns in databases is one of the main topics in data mining and many efficient solutions have been developed for different classes of patterns and data collections. Almost all techniques, however, work on so called transaction databases [1]. Not only for itemsets, but also in the case of trees [20] and graphs [12, 15, 19], the database consists of a collection of transactions, and a frequent pattern is discovered if it occurs in enough such transactions. Even in the multi-relational case, as considered in the WARMR system [4], the database can be seen as a collection of transactions in which each transaction consists of a small relational database. A query is then called frequent if it gives a non-empty answer in enough of such databases.

Obviously, many relational databases are not suited to be converted into such a transactional format and even if this would be possible, a lot of information implicitly encoded in the relational model would be lost after conversion. Recently, we have considered association rule mining on arbitrary relational databases by combining pairs of queries which could reveal interesting properties in the database [8, 13]. Intuitively, we pose two queries on the database such that one query is more specific than the other (w.r.t. query containment). Then, if the number of tuples in the output of both queries is almost the same, a potentially interesting discovery is revealed.

To illustrate, consider the well known Internet Movie Database [11] containing almost all possible information about movies, actors and everything related to that, and consider the following queries: first, we ask for all actors that have starred in a movie of the genre ‘drama’; then, we ask for all actors that have starred in a movie of the genre ‘drama’, but that also starred in a (possibly different) movie of the genre ‘comedy’.

Now suppose the answer to the first query consists of 1000 actors, and the answer to the second query consists of 900 actors. Obviously, these answers do not necessarily reveal any significant insights on themselves, but when combined, it reveals the potentially interesting pattern that actors starring in 'drama' movies typically (with a probability of 90%) also star in a 'comedy' movie. Of course, this pattern could also have been found by first preprocessing the database, and creating a transaction for each actor containing the set of all genres of movies he or she appeared in. Similarly, a pattern like: 77% of the movies starring Ben Affleck, also star Matt Damon, could be found by posing the query asking for all movies starring Ben Affleck, and the query asking for all movies starring both Ben Affleck and Matt Damon. Again, this could also be found using frequent set mining methods, but this time, the database should have been differently preprocessed in order to find this pattern. Furthermore, it is even impossible to preprocess the database only once in such a way that the above two patterns would be found by frequent set mining, as they are counting different types of transactions: actors in the first example and movies in the second example.

Also truly relational patterns can be found which can not be found using typical set mining techniques, such as 80% of all movie directors that have ever been an actor in some movie, also star in at least one of the movies they directed. This is expressed by two queries of which one asks for all movie directors that have ever acted, and the second one asks for all movie directors that have ever acted in one of their own movies.

The Conqueror algorithm recently developed by Goethals et al. [8] has shown to discover interesting association rules over a simple, but appealing subclass of conjunctive queries, called *simple conjunctive queries*. Furthermore, the algorithm had an efficient database-oriented implementation in SQL. One challenge that remained to be solved in this approach, was the huge number of generated patterns. Part of the volume is inherently due to the relational setting, but a substantial part, however, is due to redundancies induced by dependencies embedded in the data.

Jen et al. [13], studied the problem of mining all frequent queries from a single relational table. They considered projection-selection queries, and assumed that the table to be mined satisfies a set of functional dependencies. A pre-ordering over queries was defined, and shown to be anti-monotonic towards the support measure. Moreover, this pre-ordering induces an equivalence relation and two equivalent queries are shown to have the same support. Therefore, one computation per equivalence class allows to know the support of all queries in that class. In [14], this work has been generalised to several tables in the case where the database operates over a star schema. The challenge however remains to generalise the theory to arbitrary relational databases.

Clearly, the combination of the approaches in [13] and [8] would resolve the issues posed, *i.e.,* mining non redundant simple conjunctive queries (thus including arbitrary joins), given a collection of functional dependencies over the relations of an arbitrary relational database. This is one major contribution of this paper.

Moreover, combining these techniques also results in new opportunities. That is, next to the given functional dependencies, we introduce a novel technique to discover previously unknown functional dependencies, and immediately exploit them for reducing the number of frequent queries in the output. Furthermore, we do so not only for the relations of the database, but also for *any join* of relations. This is the second con-

tribution of this paper, and several experiments clearly show the benefits of this approach, thus making the discovery of simple conjunctive queries a feasible and attractive method towards the exploration of arbitrary relational databases.

The paper is organised as follows: In Section 2, we recall the basic concepts and definitions used in this work and we briefly review from [13] how functional dependencies are used to compare queries. We present our algorithm Conqueror$^+$ in Section 3, combining the two approaches [8, 13], and in Section 4, we report experiments, showing that Conqueror$^+$ clearly outperforms Conqueror. We conclude in Section 6.

2 Formal Model

2.1 Background

We consider a fixed attribute set U and a relational database schema $\mathcal{D} = \{R_1, \ldots, R_n\}$ over U in which, for $i = 1, \ldots, n$, R_i is a relation name associated with a subset of U, called the *schema* of R_i and denoted by $sch(R_i)$. Without loss of generality, we assume that, for all distinct i and j in $\{1, \ldots, n\}$, $sch(R_i) \cap sch(R_j) = \emptyset$. In order to make this assumption explicit, for all i in $\{1, \ldots, n\}$, every A in $sch(R_i)$ is referred to as $R_i.A$.

We also assume that we are given functional dependencies over $\mathcal{D}$. More precisely, each R_i is associated with a set of functional dependencies over $sch(R_i)$, denoted by $\mathcal{FD}_i$, and the set of all functional dependencies defined in $\mathcal{D}$ is denoted by $\mathcal{FD}$.

As in [8], the queries of interest in our approach, are conjunctive projection-selection-join queries whose joins are expressed using a conjunction of selection conditions of the form $R_i.A = R_j.A'$. We note that by doing so, all possible equi-joins can be considered, which would not the case using the universal relation associated to the given database. Moreover, we recall from [8] that such a conjunctive condition F induces a partition $blocks(F)$ of U, where every block β of $blocks(F)$ is a maximal set of attributes such that for all $R_i.A$ and $R_j.A'$ in β, $R_i.A = R_j.A'$ is a consequence of F. In such a case, we say that R_i and R_j are *connected through* F.

Definition 1 *Denoting by R the cartesian product $R_1 \times \ldots \times R_n$, let $Q = \pi_X \sigma_F R$ where $F = \bowtie(Q) \wedge \sigma(Q)$, such that $\bowtie(Q)$ and $\sigma(Q)$ are respectively conjunctions of selection conditions of the form $R_i.A = R_j.A'$ and $R_k.A = a$, where i, j and k are in $\{1, \ldots, n\}$ and a is in $dom(A)$. $Q = \pi_X \sigma_F R$ is said to be a* simple conjunctive query *if all relation names occurring in X or in $\sigma(Q)$ are connected through $\bowtie(Q)$.*

Given a simple conjunctive query $Q = \pi_X \sigma_F R$, the set X is denoted by $\pi(Q)$ and the tuple defined by the conjunctive selection condition $\sigma(Q)$ is denoted by Q^σ.

We call Q a join query *if $\sigma(Q)$ is the empty condition and if $\pi(Q)$ is the set of all attributes of all relation names occurring in $\bowtie(Q)$. Given a simple conjunctive query Q, we denote by $J(Q)$ the join query such that $\bowtie(J(Q)) = \bowtie(Q)$.*

To simplify notation, given a simple conjunctive query Q, the corresponding partition of U, $blocks(\bowtie(Q))$ is simply denoted by $blocks(Q)$. We emphasise that, according to Definition 1, considering simple conjunctive queries avoids computing cartesian products. We illustrate this definition below.

Example 1. Let us consider a database schema $\mathcal{D}$ consisting of two relation names R_1 and R_2 with the following schemas: $sch(R_1) = \{A, B\}$ and $sch(R_2) = \{C, D, E\}$.

According to Definition 1, R denotes the cartesian product $R_1 \times R_2$. Since $sch(R_1) \cap sch(R_2)$ is clearly empty, in this example and in the forthcoming examples dealing with $\mathcal{D}$, we do not prefix attributes with relation names. For example, $R_1.A$ is denoted by A.

The query $Q = \pi_{AD}\sigma_{(A=B)\wedge(E=e)}R$ is *not* a simple conjunctive query because R_1 and R_2 are not connected through the condition $A = B$. Computing the answer to this query requires to consider explicitly the cartesian product $R_1 \times R_2$.

On the other hand, $Q_1 = \pi_{AD}\sigma_{(A=C)\wedge(E=e)}R$ is a simple conjunctive query such that $\bowtie(Q_1) = (A = C)$, $\pi(Q_1) = AD$, $\sigma(Q_1) = (E = e)$ and $Q^\sigma = e$. Moreover, $J(Q_1) = \pi_{ABCDE}\sigma_{(A=C)}R$, and $blocks(Q_1)$ contains four blocks, namely: $\{A, C\}$, $\{B\}$, $\{D\}$ and $\{E\}$. In this case, computing the answer to Q_1 does not require to consider the cartesian product $R_1 \times R_2$, since R_1 and R_2 are joined through $A = C$. □

We now define as in [8] the *support* of a query, and when a query is said to be *frequent*.

Definition 2 *Given an instance $\mathcal{I}$ of $\mathcal{D}$ and a simple conjunctive query Q, the answer to Q in $\mathcal{I}$ is denoted by $Q(\mathcal{I})$ and is seen as a set in which no duplicates are allowed.*

The support *of Q in $\mathcal{I}$, denoted $support_{\mathcal{I}}(Q)$ or simply $support(Q)$, is the cardinality of the answer to Q in $\mathcal{I}$. Given a minimum support threshold minsup, Q is said to be* frequent *if $support(Q) > minsup$.*

To end the preliminaries, we mention the strong relationship between support and functional dependency, as stated by the following proposition whose easy proof is omitted.

Proposition 1 *Let T be a relational table over the attribute set $sch(T)$ and let X and X' be subsets of $sch(T)$. T satisfies $X \rightarrow X'$ if and only if $support(\pi_{XX'}T) = support(\pi_X T)$.*

In the context of Example 1, for $Q = \pi_{AD}\sigma_{(A=C)}R$ and $Q' = \pi_A\sigma_{(A=C)}R$, considering an instance $\mathcal{I}$ of $\mathcal{D}$ for which $support(Q) = support(Q')$ indicates that $\sigma_{(A=C)}R(\mathcal{I})$ satisfies the functional dependency $A \rightarrow D$. Consequently, for every conjunctive selection condition S, the queries $Q_S = \pi_{AD}\sigma_{(A=C)\wedge S}R$ and $Q'_S = \pi_A\sigma_{(A=C)\wedge S}R$ also have the same support. Thus, computing the support of Q'_S is redundant, assuming that the support of Q_S is known.

We recall that one of the main contributions of this paper is to discover functional dependencies in order to avoid computing unnecessary supports.

2.2 Query Comparison

Inspired by [13], we compare queries based on functional dependencies.

Definition 3 *Let $Q_1 = \pi_{X_1}\sigma_{F_1}R$ and $Q_2 = \pi_{X_2}\sigma_{F_2}R$ be two simple conjunctive queries. Denoting by Y_i the schema of Q_i^σ, for $i = 1, 2$, $Q_1 \preceq Q_2$ holds if*

1. *$\bowtie(Q_1) \subseteq \bowtie(Q_2)$,*
2. *$J(Q_2)(\mathcal{I})$ satisfies $X_1Y_2 \rightarrow X_2$ and $Y_2 \rightarrow Y_1$, and*
3. *the tuple $Q_1^\sigma Q_2^\sigma$ is in $\pi_{Y_1Y_2}J(Q_2)(\mathcal{I})$.*

Example 2. In the context of Example 1, assume that $\mathcal{FD}_1 = \emptyset$ and $\mathcal{FD}_2 = \{C \rightarrow D, E \rightarrow D\}$, and let $Q_1 = \pi_{AD}\sigma_{(A=C)\wedge(E=e)}R$ and $Q_2 = \pi_C\sigma_{(A=C)\wedge(D=d)}R$.

We have $\bowtie(Q_1) = \bowtie(Q_2)$ and $J(Q_1) = J(Q_2) = \pi_{ABCDE}\sigma_{(A=C)}R$. Then, if $\mathcal{I}$ is an instance of $\mathcal{D}$, $J(Q_2)(\mathcal{I})$ satisfies $\mathcal{FD}$. Moreover, due to the equality defining $\bowtie(Q_2)$, $J(Q_2)(\mathcal{I})$ also satisfies $A \rightarrow C$ and $C \rightarrow A$. Therefore, $J(Q_2)(\mathcal{I})$ satisfies $CE \rightarrow AD$ and $E \rightarrow D$, and so, if $de \in \pi_{DE}J(Q_2)(\mathcal{I})$, by Definition 3, $Q_2 \preceq Q_1$. □

It can be seen from [13] that $\preceq$ is a pre-ordering and that the support of queries is anti-monotonic with respect to $\preceq$. In other words, for all Q_1 and Q_2 such that $Q_1 \preceq Q_2$, we have $support(Q_2) \leq support(Q_1)$. Anti-monotonicity is used in our algorithms to prune infrequent queries, in much the same way as in Apriori [1].

Moreover, the pre-ordering $\preceq$ induces an equivalence relation, denoted by $\sim$, defined as follows: given two simple conjunctive queries Q_1 and Q_2, $Q_1 \sim Q_2$ holds if $Q_1 \preceq Q_2$ and $Q_2 \preceq Q_1$. As a consequence of anti-monotonicity, if $Q_1 \sim Q_2$ holds then $support(Q_1) = support(Q_2)$. Thus, only *one* computation per equivalence class modulo $\sim$ allows to know the support of *all* queries in that class.

In order to characterize equivalence classes modulo $\sim$, we denote by X^+ the closure of a relation schema X with respect to a given set of functional dependencies FD. Then, based on [13], it can be seen that for $Q_1 = \pi_{X_1}\sigma_{F_1}R$ and $Q_2 = \pi_{X_2}\sigma_{F_2}R$, $Q_1 \sim Q_2$ holds if and only if $\bowtie(Q_1) = \bowtie(Q_2)$, $(X_1Y_1)^+ = (X_2Y_2)^+$, $Y_1^+ = Y_2^+$ and $Q_1^\sigma Q_2^\sigma \in \pi_{Y_1Y_2}J(Q_1)(\mathcal{I})$.

Now, given a query Q, the representative of the equivalence class of Q considered in this paper is the query Q^+, such that $\pi(Q^+) = \pi(Q)^+$, $\bowtie(Q^+) = \bowtie(Q)$ and $\sigma(Q^+)$ is the selection condition corresponding to the super tuple of Q^σ, denoted by $(Q^\sigma)^+$, defined over $sch(Q^\sigma)^+$, and that belongs to $\pi_{sch(Q^\sigma)^+}J(Q)(\mathcal{I})$.

Moreover, if $\pi(Q) \subseteq sch(Q^\sigma)$ then the support of Q is 1, which is meant to be less than the minimum support threshold. Therefore, the queries Q of interest are such that

$$\pi(Q) = \pi(Q)^+, sch(Q^\sigma) = sch(Q^\sigma)^+, \text{and } sch(Q^\sigma) \subset \pi(Q).$$

In what follows, such queries are said to be *closed queries* and the closed query equivalent to a given query Q is denoted by Q^+.

It is important to notice that, considering only such queries in our algorithms, reduces the size of the output set of frequent queries.

Example 3. Referring back to the queries Q_1 and Q_2 of Example 2, it is easy to see that they do *not* satisfy the restrictions above. For instance, as $sch(Q_1^\sigma) = E$ and $\pi(Q_1) = AD$, the inclusion $sch(Q_1^\sigma) \subset \pi(Q_1)$ is not satisfied. It can be seen that none of these queries are closed, and thus, none of them is considered in our algorithms. But as $J(Q_1)(\mathcal{I})$ satisfies $C \rightarrow D$, $E \rightarrow D$, $A \rightarrow C$ and $C \rightarrow A$, the closed queries Q_1^+ and Q_2^+ defined below are processed instead.

$$Q_1^+ = \pi_{ACDE}\sigma_{(A=C)\wedge(E=e)}R \text{ and } Q_2^+ = \pi_{ACDE}\sigma_{(A=C)\wedge(E=e)\wedge(D=d)}R.$$

We also note that Q_1 and Q_2 would not be considered either in [8], as in there, $\pi(Q_i)$ $(i = 1, 2)$ is required to contain all attributes from the same block of $blocks(Q_i)$ but no attributes from $\sigma(Q_i)$. Thus, in [8], $Q_1' = \pi_{ACD}\sigma_{(A=C)\wedge(E=e)}R$ and $Q_2' = \pi_{AC}\sigma_{(A=C)\wedge(E=e)\wedge(D=d)}R$ are processed instead. As $Q_i \sim Q_i' \sim Q_i^+$ for $i = 1, 2$, these queries have the same support. □

3 Mining Queries under Functional Dependencies

3.1 Algorithm Conqueror$^+$

In this section, we present our algorithm called Conqueror$^+$ (given as Algorithm 1) for mining frequent queries. We mention in this respect that frequent simple conjunctive queries $\pi_X \sigma_F R$ are mined in much the same way as the Conqueror algorithm [8], that is, according to the following steps:

- **Join loop:** Generate all instantiations of F, without constants, in a breadth-first manner, using restricted growth to represent partitions [8]. Every partition gives rise to a join query JQ and functional dependencies of its ancestors are inherited.
- **Projection loop:** For each generated partition, all projections of the corresponding join query JQ are generated in a breadth-first manner, and their frequency is tested against the given instance $\mathcal{I}$. During this loop, functional dependencies are discovered and used to prune the search space.
- **Selection loop:** For each frequent projection-join query, constant assignments are added to F in a breadth-first manner, as in Conqueror. Moreover, here again, functional dependencies are used to prune the search space.

As in the Conqueror algorithm, attributes are ordered, so as candidate queries are generated at most once in the different loops: This ordering is implicit lines 1 and 12 in Algorithm 1 (the k-th element in the string refers to the k-th attribute according to the ordering), and is explicitly used line 17 in Algorithm 2 and line 10 in Algorithm 3.

As an important difference with the Conqueror algorithm, a (possibly empty) set of functional dependencies $\mathcal{FD}$ can be specified as input. This set is first used for the relations of the database instance (line 3 of Algorithm 1) and then augmented during the projection loop (line 15 of Algorithm 2).

3.2 Join Loop

The generation of joins is done in much the same way as in Conqueror ([8]), by generation of restricted growth strings [18]. Such a restricted growth string represents a partition of the attributes, and such a partition maps to a join.

For example, referring back to Example 1, the set U of all attributes occurring in $\mathcal{D}$ is $\{A, B, C, D, E\}$. Then, the restricted growth string 12231 represents the condition $(A = E) \wedge (B = C)$, which corresponds to the partition $\{\{A, E\}, \{B, C\}, \{D\}\}$.

As in the Conqueror algorithm, we include a check against the user defined most specific join, which allows a user to specify the sensible joins in the database (see line 11, Algorithm 1). By default, however, every possible join of every attribute pair is considered. A new addition to the join loop is the inheritance of functional dependencies shown on lines 13-14, and discussed in detail in Section 3.5.

3.3 Projection Loop

Compared to the Conqueror algorithm, one major change in the projection loop is the fact that the generation of selections is now performed after all projections are generated (line 22, Algorithm 2) so as to be able to immediately use the discovered functional

Algorithm 1 Conqueror$^+$

Input: Database $\mathcal{D}$, Set of functional dependencies $\mathcal{FD}$, Minimum support threshold *minsup*
Output: Frequent Queries FQ

```
 1: ⋈(Q) := "1" // initial restricted growth string
 2: for all R_i in D do
 3:    FD_Q := FD_i
 4:    push(Queue, R_i)
    // Join Loop
 5: while not Queue is empty do
 6:    JQ := pop(Queue)
 7:    if ⋈(JQ) does not represent a cartesian product then
 8:       FQ := FQ ∪ ProjectionLoop(JQ)
 9:    children := RestrictedGrowth(⋈(JQ), m)
10:    for all rgs in children do
11:       if join defined by rgs is not more specific than the user most specific join then
12:          ⋈(JQC) := rgs
13:          for all PJQ such that ⋈(JQC) = ⋈(PJQ) ∧ (R_i.A = R_j.A') do
14:             FD_JQC := FD_JQC ∪ FD_PJQ
15:             if ⋈(PJQ) = "1" then
16:                FD_JQC := FD_JQC ∪ {R_i.A → R_j.A', R_j.A' → R_i.A}
17:          blocks(JQC) := blocks(JQ) where the blocks containing R_i.A and R_j.A' are merged
18:          push(Queue, JQC)
19: return FQ
```

dependencies to prune redundant queries. The functional dependency discovery is performed lines 13-16 of Algorithm 2 and is discussed in Section 3.5.

We point out that, according to lines 17-20 of Algorithm 2, candidate projection queries are generated by removing blocks in $blocks(JQ)$, because attributes in a given block are mutually dependent. However, it might be the case that removing such a block does not result in a closed projection schema. This is why, line 9 of Algorithm 2, we check whether π(PQ) is closed; if not, the projection query is simply queued without any further processing. This however induces complications in the monotonicity check line 10 of Algorithm 2, because projections over non closed schemas are not processed. To cope with this difficulty, if PQ is such that π(PQ) is closed, for every predecessor PPQ of PQ, the closure of π(PPQ) under $\mathcal{FD}_{\mathrm{Q}}$ is computed. The check is passed if all corresponding projection queries are in FPQ.

Also notice that the function blocks(Q) returns the set of connected blocks of a restricted growth string, *i.e.,* the connected part of the partition $blocks(Q)$. We require such blocks to form a single connected component, so as to avoid considering cartesian products, as stated in Definition 1. Clearly, line 7 in Algorithm 1 prunes these queries.

3.4 Selection Loop

In the selection loop of our new algorithm, marked queries are not considered, since they are redundant (line 23, Algorithm 2). When adding blocks to the selection condition, the closure is taken, ensuring no redundant queries are generated (line 13, Al-

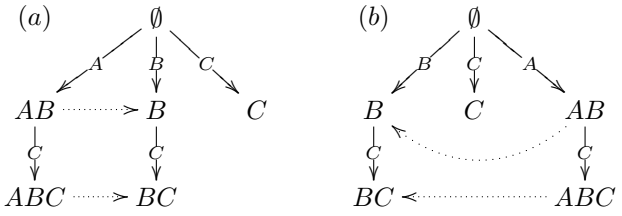
Algorithm 2 ProjectionLoop

Input: Conjunctive Query Q
1: **if** ⋈(Q) = "1" **then**
2: π(Q) := $sch(R_i)$ // Q is the query R_i
3: **else**
4: $\pi(Q)$:= **union of** blocks(Q)
5: push(*Queue*, Q)
6: FPQ := $\emptyset$
7: **while not** *Queue* is empty **do**
8: PQ := pop(*Queue*)
9: **if** π(PQ) is closed **then**
10: **if** monotonicty(PQ) **then**
11: **if** support(PQ) $>$ *minsup* **then**
12: FPQ := FPQ $\cup$ {PQ}
13: **for all** PPQ in FPQ such that ($\nexists$ PPQ$'$ $\in$ FPQ : π(PQ) $\subset$ π(PPQ$'$) $\subset$ π(PPQ)) **do**
14: **if** support(PQ) = support(PPQ) **then**
15: $\mathcal{FD}_{\mathrm{Q}} := \mathcal{FD}_{\mathrm{Q}} \cup \{\pi(\mathrm{PQ}) \rightarrow \pi(\mathrm{PPQ}) \setminus \pi(\mathrm{PQ})\}$
16: mark PQ
17: **for all** $\beta > lastremoved$(PQ) **do**
18: π(PQC) := π(PQ) with block β removed
19: $lastremoved$(PQC) = β
20: push(*Queue*, PQC)
21: FQ := FQ $\cup$ FPQ
22: **for all** PQ $\in$ FPQ **do**
23: **if** PQ is not marked **then**
24: FQ := FQ $\cup$ SelectionLoop(PQ)
25: **return** FQ

gorithm 3). However, closing of these sets of blocks requires to reorder the queue of candidates in order to use the Apriori-trick. The following example illustrates this point.

Example 4. Considering the attributes A, B and C, along with the functional dependency $A \rightarrow B$, the generation of sets for the selection results in the generation-tree (a) shown below. Indeed, the addition of A entails that B must also be added so as to consider closed schemas only.

However, because of the monotonicity property, we need to consider B before AB (since the selection according to B is less restrictive than that according to AB). We accomplish this by reordering the candidate queue, to ensure B is considered before AB and BC is considered before ABC, as shown in the generation-tree (b) below. □



Moreover, as stated previously, line 14 of Algorithm 3 ensures that $\sigma(Q)$ is a strict subset of $\pi(Q)$. However, not all strict subsets of $\pi(PQ)$ are considered, since we only have to consider assignments over closed schemas under $\mathcal{FD}_{JQ}$ (see line 13, Algorithm 3). Furthermore, in line 14 of Algorithm 3, we make sure that the corresponding closure has not been processed previously, which can happen since a closed set can be generated from several non-closed sets.

Then, in lines 7-8 of Algorithm 3, the obtained queries are processed against $\mathcal{I}$ using the same strategy as in [8]. The instantiation of constant values in Algorithm 3 is performed analogously to Conqueror by performing SQL queries in the database. For further details, we therefore refer the reader to [8].

Algorithm 3 SelectionLoop

Input: Conjunctive Query Q
1: push(*OrderedQueue*,Q)
2: **while** not *OrderedQueue* is empty **do**
3: CQ := pop(*OrderedQueue*)
4: **if** σ(CQ) $= \emptyset$ **then**
5: *toadd* := all blocks of π(Q)
6: **else if** monotonicty(CQ) **then**
7: **if** exist frequent constant values for σ(CQ) in $\mathcal{I}$ **then**
8: FQ := FQ $\cup$ instances of CQ
9: *uneq* := all blocks of π(Q) $\notin \sigma$(CQ)
10: *toadd* := all blocks B in *uneq* $>$ last of σ(CQ)
11: **for all** $B_i \in$ *toadd* **do**
12: σ(CQC) := σ(CQ) with B_i added
13: σ(CQC) := closure of σ(CQC) under $\mathcal{FD}_{\mathrm{Q}}$
14: **if** σ(CQC) has not been generated before **and** σ(CQC) is different than π(Q) **then**
15: push(*OrderedQueue*, CQC)
16: **return** FQ

3.5 Handling and Discovering Functional Dependencies

In this section, we show that, according to our algorithms:

1. A given join query is associated with the set of all functional dependencies satisfied by its predecessor join queries.
2. Only join and projection queries over *closed* relation schemas are processed.
3. Considering given functional dependencies along with *discovered* functional dependencies preserves the above property.

Handling Functional Dependencies. A given join query JQ is associated with a set of functional dependencies, denoted by $\mathcal{FD}_{JQ}$, and built up in Algorithm 1 as follows.

First, when $\bowtie$(Q) is the restricted growth string 1, every instantiated relation $R_i(\mathcal{I})$ in the database is pushed in *Queue* (lines 2 and 5, Algorithm 2), associated with the

set $\mathcal{FD}_i$ (see line 3, Algorithm 1). Then, the restricted growth strings represent a join condition of the form $(R_i.A = R_j.A')$. Denoting by JQ the corresponding join query, if $R_i = R_j$ then $JQ(\mathcal{I})$ satisfies $\mathcal{FD}_i$ (since JQ is a selection of R_i) along with $R_i.A \rightarrow R_i.A'$ and $R_i.A' \rightarrow R_i.A$. Thus, $\mathcal{FD}_{JQ}$ is set to $\mathcal{FD}_i \cup \{R_i.A \rightarrow R_i.A', R_i.A' \rightarrow R_i.A\}$. Similarly, if $R_i \neq R_j$, then JQ is a join of R_i and R_j, and so, $JQ(\mathcal{I})$ satisfies $\mathcal{FD}_i \cup \mathcal{FD}_j$, as well as $R_i.A \rightarrow R_j.A'$ and $R_j.A' \rightarrow R_i.A$. Thus, we set $\mathcal{FD}_{JQ} = \mathcal{FD}_i \cup \mathcal{FD}_j \cup \{R_i.A \rightarrow R_j.A', R_j.A' \rightarrow R_i.A\}$ (see lines 13-16 of Algorithm 1). At this stage, $\pi(JQ)$ is either $sch(R_i)$ (if $R_i = R_j$) or $sch(R_i) \cup sch(R_j)$ (if $R_i \neq R_j$), and so, $\pi(JQ)$ is closed under $\mathcal{FD}_{JQ}$.

In the general case, at a given level, the join query JQ is generated from join queries PJQ in the previous level by setting $\bowtie(JQ)$ to $\bowtie(PJQ) \wedge (R_i.A = R_j.A')$, and by augmenting $\pi(PJQ)$ accordingly. Therefore, $JQ(\mathcal{I})$ satisfies the dependencies of $\mathcal{FD}_{PJQ}$, and thus, *$\mathcal{FD}_{JQ}$ is set to be the union of all $\mathcal{FD}_{PJQ}$ where PJQ allows to generate JQ* (see lines 13-14 of Algorithm 1). Consequently, assuming that $\pi(PJQ)$ is closed under $\mathcal{FD}_{PJQ}$ clearly entails that $\pi(JQ)$ is closed under $\mathcal{FD}_{JQ}$.

Thus, for every join query JQ, $\pi(JQ)$ is closed under those functional dependencies of $\mathcal{FD}_{JQ}$ that belong to $\mathcal{FD}$ or that are obtained through the connected blocks of $blocks(JQ)$. Moreover, the discovered functional dependencies in the projection loop of JQ preserve this property, because these new dependencies are defined with attributes in $\pi(JQ)$ only. Thus, *for every join query JQ, $\pi(JQ)$ is closed under $\mathcal{FD}_{JQ}$.*

Then, the check performed line 9 of Algorithm 2 ensures that only those projection-join queries PQ such that $\pi(PQ)$ is closed under $\mathcal{FD}_{JQ}$ are considered. We note that for performing this check, it is enough to make sure that there is no dependency $X \rightarrow Y$ in $\mathcal{FD}_{JQ}$ such that $X \subseteq \pi(PQ)$ and $Y \not\subseteq \pi(PQ)$.

Discovering Functional Dependencies. Functional dependencies, other than those in $\mathcal{FD}$, are *discovered* in the projection loop (see lines 13-16 of Algorithm 2) as follows. At a given level, a projection-join query PQ is generated from the projection-join queries PPQ of the previous level by removing blocks from $\pi(PPQ)$. Thus, by Proposition 1, if $support(PQ) = support(PPQ)$ (see line 14 of Algorithm 2), $JQ(\mathcal{I})$ satisfies $\pi(PQ) \rightarrow \pi(PPQ) \setminus \pi(PQ)$. The dependency is thus added to $\mathcal{FD}_{JQ}$ and PQ is marked, since $\pi(JQ)$ is no longer closed (see lines 15 and 16 of Algorithm 2).

Notice that, as projection-join queries are generated in a breadth-first manner, the 'best' functional dependencies (*i.e.,* those with minimal left-hand side) are discovered last, during the projection loop. However, by doing so, we mark all queries that do not have to be processed in the selection loop. The following example illustrates this point.

Example 5. In the context of Example 1, let us consider the projection loop associated to the join query $JQ = \pi_{ABCDE}\sigma_{(A=C)}R$. In this case, $blocks(JQ) = \{\{A, C\}, \{B\}, \{D\}, \{E\}\}$. Assuming that all projections are frequent and that $JQ(\mathcal{I})$ satisfies $A \rightarrow D$, the following dependencies are found: $ACBE \rightarrow D$, $ACE \rightarrow D$, $ACB \rightarrow D$ and $AC \rightarrow D$. Consequently, the queries $\pi_{ACBE}(JQ)$, $\pi_{ACE}(JQ)$, $\pi_{ACB}(JQ)$ and $\pi_{AC}(JQ)$ are marked, and so, are not processed by the selection loop.

We note that, $A \rightarrow D$ is actually not found, because $\mathcal{FD}_{JQ}$ contains $A \rightarrow C$ and $C \rightarrow A$, which enforces A and C to appear together in the projections. Of course, $A \rightarrow D$ is a consequence of $AC \rightarrow D$ and $A \rightarrow C$ that now belong to $\mathcal{FD}_{JQ}$. □

The output of the projection loop is processed in the selection loop of Algorithm 3 as follows: for every *non marked* frequent projection-join query PQ, selections over closed schemas are generated breadth-first by assigning constant values to some of the attributes in $\pi(PQ)$.

4 Experimental Results

The Conqueror$^+$ algorithm was written in Java using JDBC to communicate with a sqlite relational database. Experiments were run on a standard computer with 2GB RAM and a 2.16 GHz processor. We also note that this implementation not only computes frequent queries, but also association rules. The issue of association rules is not addressed in this paper, due to lack of space. We performed experiments using Conqueror$^+$ and compared it to Conqueror [8]. We used the backend database of an online quiz website [2] and a snapshot of the Internet Movie Database (IMDB) [11]. The characteristics of these databases are shown in Table 1.

SCORES.*	868755
SCORES.SCORE	14
SCORES.NAME	31934
SCORES.QID	5144
SCORES.DATE	862769
SCORES.RESULTS	248331
SCORES.MONTH	12
SCORES.YEAR	6
QUIZZES.*	4884
QUIZZES.QID	4884
QUIZZES.TITLE	4674
QUIZZES.AUTHOR	328
QUIZZES.CATEGORY	18
QUIZZES.LANGUAGE	2
QUIZZES.NUMBER	539
QUIZZES.AVERAGE	4796

(a) Quiz database

ACTORS.*	45342
ACTORS.AID	45342
ACTORS.NAME	45342
GENRES.*	21
GENRES.GID	21
GENRES.NAME	21
MOVIES.*	71912
MOVIES.MID	71912
MOVIES.NAME	71906
ACTORMOVIES.*	158441
ACTORMOVIES.AID	45342
ACTORMOVIES.MID	54587
GENREMOVIES.*	127115
GENREMOVIES.GID	21
GENREMOVIES.MID	71912

(b) IMDB

Table 1: Number of tuples per attribute in the QuizDB and IMDB databases

4.1 Impact of Dependency Discovery

We performed four types of experiments with functional dependencies. As a first type, we executed the regular Conqueror. The second type, denoted 'disc' in Figure 1, is Conqueror$^+$ where discovery of dependencies is enabled, but the set of initial provided dependencies is empty. The third type, denoted 'given', is Conqueror$^+$ provided with a set of initial dependencies, but without any discovery of functional dependencies. For QuizDB we provided the key dependencies of the QUIZZES and SCORES relations, and for IMDB we provided the key dependencies for ACTORS, GENRES and MOVIES. The fourth type, denoted as 'given+disc', is Conqueror$^+$ provided with these dependencies as well as discovery of new functional dependencies.

As can be seen in Figure 1a, Conqueror$^+$ with discovery greatly outperforms Conqueror in runtime. This is due to the large reduction in number of queries generated which is clear from the figure. Adding an initial set of (key) functional dependencies results in a small gain in runtime, due to a small reduction in number of queries generated. Similarly, providing a set of dependencies whilst also discovering new ones, results in a small relative gain. We also observe that the exponential behavior of query generation is still present, but only for low support values. Furthermore, for Conqueror$^+$ with discovery, runtime remains almost linear for a large portion of the support values, while for Conqueror, it is increasing rapidly.

The experiments on the IMDB shown in Figure 1b show similar results, but in this case, the impact of discovery is smaller. The small amount of attributes in the database reduces the impact of the use of functional dependencies. It is however clear that also in this case, the discovery of functional dependencies reduces the exponentiality of query generation and has an almost linear runtime. Likewise the small impact of providing key dependencies as input to the algorithm, is comparable to QuizDB.

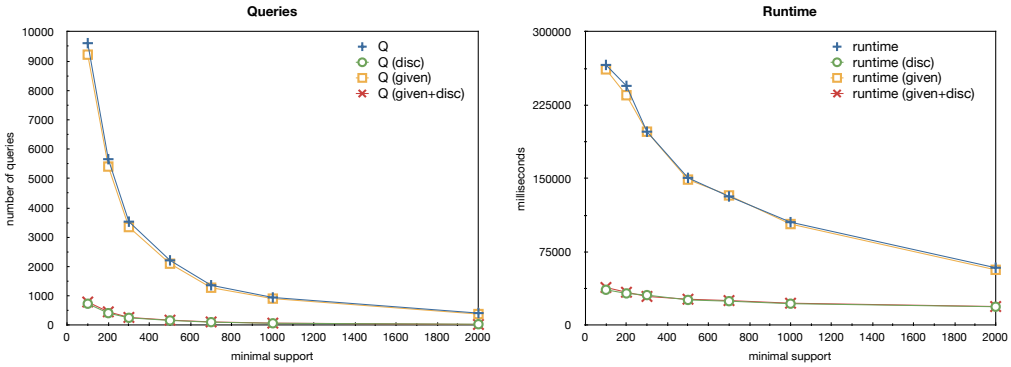
We also performed some time analysis to determine the cost of functional dependency discovery. The results for an experiment using QuizDB are shown in Figure 2. It is clear that the time needed for the discovery of functional dependencies (shown as 'fdisc' in Figure 2a) is negligible in comparison to the time gained in the selection loop (shown as 'sel'). Adding discovery also requires extra time in the join loop (shown as 'join'), but again, the gain in the selection loop outweighs this. Looking at the partitioning of time in Figure 2b, we clearly see that most time is spent in output and input. Since functional dependency discovery in Conqueror$^+$ greatly reduces output and input, we get a large reduction in runtime as was observed in Figure 1.

5 Related Work

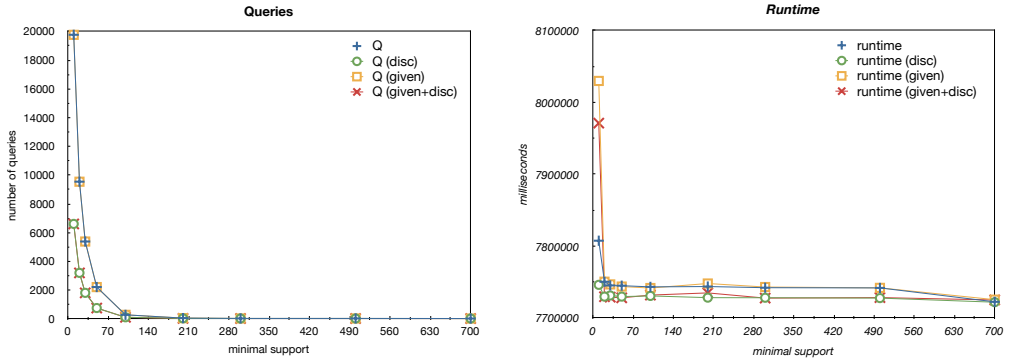
Mining frequently occurring patterns in arbitrary relational databases has been the topic of several research efforts. Dehaspe and Toivonen developed the WARMR algorithm [4], that discovers association rules in over a limited type of Datalog queries in an Inductive Logic Programming setting. The input to their algorithm consists of a collection of databases, and then, queries are generated in a level-wise manner, and each candidate query is evaluated against all of these databases. The frequency of a query is the number of databases for which it gives a nonempty answer. Therefore, the interpretation of frequent queries is incomparable to the conjunctive queries considered in this paper.

In [6], we studied a strict generalization of WARMR. The notion of *diagonal containment* provided an excellent tool to compare queries with different sets of projected attributes. Unfortunately, the search space is infinite and there exist no most general and no most specific patterns. However, the subclass of tree-shaped conjunctive queries defined over a single binary relation representing a graph was studied, showing that these tree queries are powerful patterns, useful for mining graph-structured data [7, 10]. In [8], we considered conjunctive queries over several relations, allowing a more efficient algorithm, called Conqueror.

Considering projection-selection queries over a single relation, Jen et al. introduced a new notion of query equivalence [13], taking functional dependencies into account,

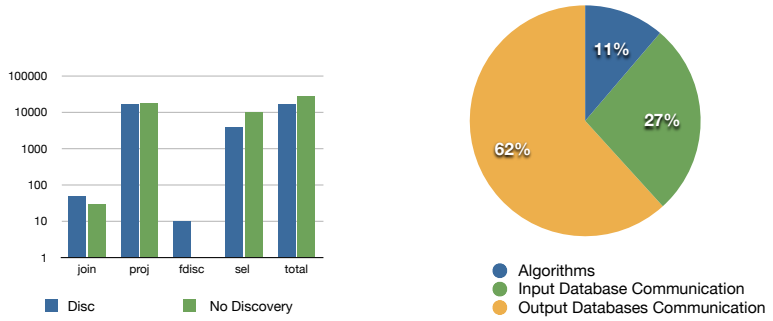


(a) QuizDB



(b) IMDB

Fig. 1: Results for Conqueror, Conqueror$^+$ with a set of Functional Dependencies given but no detection (given), Conqueror$^+$ with only detection of Functional Dependencies (disc), and Conqueror$^+$ both with given Functional Dependencies and detection.



(a) Time Analysis (b) Input/Output Time Analysis

Fig. 2: Time and I/O time analysis of a QuizDB experiment.

which is not the case in previous work. The approach of [13] has been generalised in [14] to databases defined over multiple relations, organised according to a star schema.

All approaches other than those discussed just above and dealing with mining frequent queries ([5, 9, 16, 17]) are far more restrictive than ours. Indeed, whereas our approach considers several tables and all possible ways to count supports as distinct values over all possible attribute sets, all these approaches consider a fixed relation to be mined, along with a fixed characterisation of how to count supports. For instance, in [9, 16] tuples are counted, Turmeaux et al. [17] characterize counting by tuple values over a given attributes, whereas Diop et al. [5] characterize counting by a query, called the reference. Notice that all these approaches (except for [17]) are also restricted to conjunctive queries, as is the case in this paper.

6 Concluding Remarks

The contribution of this paper is threefold. First, we combined the results of different prior work resulting in a new algorithm for mining association rules over simple conjunctive queries in arbitrary relational databases, over which functional dependencies are assumed. The algorithm makes use of the functional dependencies of the database to optimise the generation of frequent queries and prune redundant queries. Second, our new algorithm is capable of detecting new functional dependencies that were not given but that hold on the database relations or on any join of these relations. Third, these newly detected dependencies are used to prune even more redundant queries. We implemented our algorithm, and we showed that it greatly outperforms our previous methods and efficiently reduces the amount of queries generated.

Several new opportunities for future work exist. First, the additional use of key and foreign key contraints is an issue that we are currently investigating. Other appealing related constraints are *conditional functional dependency*, introduced by Fan et al. [3]. As these constraints generalise standard functional dependencies using selections, it seems interesting to investigate how they could be used and discovered in our framework.

References

1. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 309–328. AAAI-MIT Press, 1996.
2. R. Bocklandt. http://www.persecondewijzer.net.
3. Ph. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, pages 746–755, 2007.
4. L. Dehaspe and L. De Raedt. Mining association rules in multiple relations. In *7th International Workshop on Inductive Logic Programming*, volume 1297 of *LNCS*, pages 125–132. Springer Verlag, 1997.
5. C.T. Diop, A. Giacometti, D. Laurent, and N. Spyratos. Composition of mining contexts for efficient extraction of association rules. In *EDBT'02*, volume 2287 of *LNCS*, pages 106–123. Springer Verlag, 2002.
6. B. Goethals and J. Van den Bussche. Relational association rules: getting warmer. In *ESF Exploratory Workshop on Pattern Detection and Discovery in Data Mining*, volume 2447 of *LNCS*, pages 125–139. Springer, 2002.
7. B. Goethals, E. Hoekx, and J. Van den Bussche. Mining tree queries in a graph. In *ACM KDD*, pages 61–69, 2005.
8. B. Goethals, W. Le Page, and H. Mannila. Mining association rules of simple conjunctive queries. In *SIAM-SDM*, pages 96–107, 2008.
9. J. Han, Y. Fu, W. Wang, K. Koperski, and O. Zaiane. Dmql : A data mining query language for relational databases. In *SIGMOD-DMKD'96*, pages 27–34, 1996.
10. E. Hoekx and J. Van den Bussche. Mining for tree-query associations in a graph. In *IEEE ICDM)*, pages 254–264, 2006.
11. IMDB. http://imdb.com. 2008.
12. A. Inokuchi, T. Washio, and H. Motoda. An Apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, volume 1910 of *LNCS*, pages 13–23. Springer, 2000.
13. T.Y. Jen, D. Laurent, and N. Spyratos. Mining all frequent selection-projection queries from a relational table. In *EDBT'08*, pages 368–379. ACM Press, 2008.
14. T.Y. Jen, D. Laurent, and N. Spyratos. Mining frequent conjunctive queries in star schemas. In *International Database Engineering and Applications Symposium (IDEAS)*, pages 97–108. ACM Press, 2009.
15. M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *IEEE ICDM*, pages 313–320, 2001.
16. R. Meo, G. Psaila, and S. Ceri. An extension to sql for mining association rules. *Data Mining and Knowledge Discovery*, 9:275–300, 1997.
17. T. Turmeaux, A. Salleb, C. Vrain, and D. Cassard. Learning caracteristic rules relying on quantified paths. In *PKDD*, volume 2838 of *LNCS*, pages 471–482. Springer Verlag, 2003.
18. E.W. Weisstein. Restricted growth string. In *From MathWorld – A Wolfram Web Resource (http://mathworld.wolfram.com/RestrictedGrowthString.html)*, 2009.
19. X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *IEEE ICDM*, page 721, 2002.
20. M.J. Zaki. Efficiently mining frequent trees in a forest. In *ACM KDD*, pages 71–80, 2002.