An Efficient Computation of Frequent Queries in a Star Schema

Cheikh Tidiane Dieng, Tao-Yuan Jen, Dominique Laurent

An Efficient Computation of Frequent Queries in a Star Schema

Cheikh Tidiane Dieng[1,2], Tao-Yuan Jen[1], Dominique Laurent[1]

[1] ETIS - CNRS - ENSEA, Université de Cergy Pontoise, F-95000, FRANCE
[2] Laboratoire d'Analyse Numérique et Informatique, Université Gaston-Berger, Saint-Louis, SENEGAL

Abstract. Although the problem of computing frequent queries in relational databases is known to be intractable, it has been argued in our previous work that using functional and inclusion dependencies, computing frequent conjunctive queries becomes feasible for databases operating over a star schema. However, the implementation considered in this previous work showed severe limitations for large fact tables. The main contribution of this paper is to overcome these limitations using appropriate auxiliary tables. We thus introduce a novel algorithm, called Frequent Query Finder (FQF), and we report on experiments showing that our algorithm allows for an effective and efficient computation of frequent queries.

Keywords: Frequent Queries, Functional Dependencies, Inclusion Dependencies, Query Comparison, Star Schemas.

1 Introduction

The problem of discovering frequent patterns in a (relational) database is one of a main topics in data mining. However, even when patterns are restricted to conjunctive queries, this problem is known to be intractable, because the size of the search space is exponential in the size of the database. Nonetheless, it is argued in [9, 10] that mining all frequent conjunctive queries (*i.e.,* conjunctive queries whose answers have a cardinality greater than or equal to a predefined threshold) becomes tractable if the underlying database operates over a star schema, and if constraints such as functional and inclusion dependencies, are taken into account.

Indeed, it has been shown in [9, 10] that such dependencies allow for comparing queries according to a pre-ordering with respect to which the support measure is anti-monotonic (the support of a query being the number of tuples in the answer of that query). As a consequence, a level-wise algorithm such as Apriori ([1]) can be used, with the basic additional feature that the considered pre-ordering induces an equivalence relation for which two equivalent queries have the same support. Consequently, one computation per equivalence class allows to determine the support of all queries of this class.

In the present paper, similarly to [10], we consider a relational database operating over a star schema and we follow the approach of [10] for mining frequent projection-selection-join queries in which joins are performed along keys and foreign keys. In this setting, our contribution is to provide an *efficient* computation of such frequent queries.

Indeed, it is shown in [10] that the number of scans of the database in our algorithms is in $O(N \times |U|)$ (where N is the number of dimensions of the underlying star schema and $|U|$ is the number of attributes in this schema), and that, in order to have an efficient implementation, an appropriate indexing technique should be used to count the supports safely. The problem to solve is finding an *at most linear* technique for counting only once the duplicates occurring in the answers to projection queries; a problem which basically requires a *quadratic* scan of the table. Unfortunately, as such an indexing technique must work for *all* possible attribute sets, no solution could be found. In fact, in the implementation of [10], auxiliary data are stored in main memory, so as to keep track of the tuples computed so far for a given query. Therefore, this technique is somehow still quadratic (since duplicates are still checked against the auxiliary data), and more importantly, experiments result in main memory overflow for large fact tables.

In order to cope with this important limitation, we propose a novel efficient and scalable algorithm, called *Frequent Query Finder* (FQF), for the computation of frequent queries. According to FQF, every table r to be mined is associated with an auxiliary table $\mathrm{AUX}(r)$, whose role is to associate every tuple t of r with all attribute sets S such that $t.S = t'.S$ for some tuple t' occurring in r *before* t, with respect to the scanning order of r. Assuming that these auxiliary tables are computed, it turns out that counting the supports becomes *linear* in the size of the table to be mined. We are thus provided with an *efficient* and *scalable* implementation, in the sense that runtime keeps very low and that no main memory overflow occurs, even for large datasets (up to 100,000 tuples in our experiments). We refer to Section 5 for experiments.

We also emphasize that, although the computation of an auxiliary table $\mathrm{AUX}(r)$ is still quadratic in the size of r, our experiments show that our new implementation, even when involving the computation of the auxiliary tables, outperforms that of [10] (when the comparison is possible). Moreover, it is also argued in the end of the paper that the computation of the auxiliary tables can be seen as a pre-processing phase, when mining frequent queries.

The paper is organized as follows: In Section 2, we briefly overview related work, and in Section 3, we recall from [10] the basic definitions and properties of our approach. Then, in Section 4, we present our algorithm FQF for mining conjunctive queries and in Section 5, we report on experiments showing that our algorithms are efficient. Section 6 concludes the paper and discusses future work.

2 Related Work

Early approaches dealing with frequent queries [2, 3, 8, 12] consider a *fixed* set of "objects" to be counted when computing supports, meaning that

in these approaches, all queries of interest are projections over a *fixed* attribute set. Moreover, apart from [4], none of these approaches consider constraints on the data, such as functional dependencies, in order to optimize the computation. In [4], equivalent attribute sets with respect to functional dependencies are used for query optimization, based on materialized views, which is not the case in our approach.

To the best of our knowledge, [5] is the first approach for mining frequent queries in the general context where the set of objects to be counted is *not* fixed. However, in [5], equivalent queries are generated, which can not be tested efficiently (a problem that does not exist in our approach); and moreover, data dependencies are not taken into account.

The work of [6], dealing with mining tree queries in a graph, is also closely related to ours. Indeed, in [6], a graph is seen as a binary relation, and frequent tree queries are expressed as SQL projection-selection-join queries. This work is somehow generalized in [7] to the case of projection-selection-join queries, with the restriction that a given relation cannot occur more than once in the joins. Queries considered in [7] are *more general* than ours, since (*i*) all possible joins in which base relations occur at most once are considered in [7], whereas we only consider such joins along keys and foreign keys, and (*ii*) selection conditions of the form $(Y = Y')$ where Y and Y' are relation schemas are allowed in [7], which is not the case in our approach. However, in [7], dependencies are not taken into account, thus resulting in redundant computations.

In our previous work [9–11], we have considered conjunctive query mining in a star schema, focussing successively on projection queries ([11]), projection-selection queries ([9]), and projection-selection-join queries ([10]). The main contribution of this previous work is to show that taking dependencies into account in query comparison results in an efficient computation of frequent conjunctive queries. In particular, in [10], it is shown that if the database schema is a star schema, then the problem of mining frequent projection-selection-join queries where joins are performed along keys and foreign keys becomes tractable. However as previously mentioned, in [10], experiments show severe limitations, and the contribution of this paper is to propose an efficient and scalable implementation that overcomes these limitations.

3 Formal Model

3.1 Queries

We first recall that a database Δ over a *star schema* consists of a distinguished table φ with schema F, called the *fact table*, together with a set of other tables $\delta_1, \ldots, \delta_N$ with schemas $D_1, \ldots, D_N$, called the *dimension tables*, such that:

1. If $K_1, \ldots, K_N$ are the (primary) keys of $\delta_1, \ldots, \delta_N$, respectively, then, denoting by K the union of these keys (*i.e.,* $K = K_1 \ldots K_N$), K is the key of φ. In other words, for every $i = 1, \ldots, N$, δ_i satisfies $K_i \rightarrow D_i$ and φ satisfies $K \rightarrow F$. We denote by $\mathcal{F}$ the set of these functional dependencies.

2. For every $i = 1, \ldots, N$, $\pi_{K_i}(\varphi) \subseteq \pi_{K_i}(\delta_i)$ (thus each K_i is a foreign key in the fact table φ). The attribute set $M = F \setminus K$ is called the *measure* of the star schema.

As usual, we denote by $\mathcal{F}^+$ the set of all functional dependencies that can be inferred from $\mathcal{F}$, using the Armstrong's axioms, and we denote by X^+ the set of all attributes A such that $X \to A$ is in $\mathcal{F}^+$ ([13]).

In what follows, we consider a *fixed* database $\Delta = (\delta_1, \ldots, \delta_N, \varphi)$, along with projection-selection-join queries with the following specificities:

- the tuple in selection condition is either the *empty tuple*, denoted by $\top$ (in which case all tuples are selected), or a tuple y over Y (in which case all tuples t such that $t.Y = y$ are selected);
- the joins are performed along keys and foreign keys, that is, either the join is reduced to a single table, or it involves the fact table φ.

Definition 1. *Let $\Delta = (\delta_1, \ldots, \delta_N, \varphi)$ be a database over a star schema. The considered set of queries, denoted by $\mathcal{Q}$, is the set of all queries of the form $q = \pi_X(\sigma_y(r))$, or more simply $\pi_X\sigma_y(r)$, such that $XY \subseteq R$ (R denotes the schema of r), and where:*

- *r is either a table in Δ or a join of such tables containing φ;*
- *y is either the empty tuple $\top$ or a tuple over relation schema Y.*

For every query q in $\mathcal{Q}$, the support of q in Δ*, denoted by $sup(q)$, is the cardinality of the answer to q. Given a support threshold min-sup, a query q is said to be* frequent *if $sup(q) \geq$ min-sup.*

We illustrate our approach using the following example, borrowed from [10], and that we shall use as a running example throughout the paper.

Example 1. Consider the database Δ consisting of three tables and a set of functional and inclusion dependencies, as shown in Figure 1. The meaning of the attributes is as follows:

- *Cid*, *Cname* and *Caddr* stand for Customer Identifier, Customer Name and Customer Address,
- *Pid* and *Ptype* stand for Product Identifier and Product Type,
- *Qty* stands for Quantity (*i.e.,* number of products sold).

The schema of Δ is clearly a star schema, with *Sales* as its fact table, and *Cust* and *Prod* as its dimensional tables.

The queries $q_1 = \pi_{Cid}\sigma_{\texttt{Paris}}(Cust)$ and $q_2 = \pi_{Cid}\sigma_{\texttt{Paris beer}}(Cust \bowtie Prod \bowtie Sales)$ are in $\mathcal{Q}$, the answers of which being $\{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3\}$ and $\{\texttt{c}_1, \texttt{c}_2\}$, respectively. Thus, if *min-sup* $= 2$, these queries are frequent.

On the other hand, $\pi_{Cid}(Cust)$ and $\pi_{Cid}(Cust \bowtie Sales)$ are also in $\mathcal{Q}$, and are written as $\pi_{Cid}\sigma_\top(Cust)$ and $\pi_{Cid}\sigma_\top(Cust \bowtie Sales)$, respectively. Their answers are respectively $\{\texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4\}$ and $\{\texttt{c}_1, \texttt{c}_2\}$.

3.2 Query Comparison

Definition 2. *Let $q = \pi_X\sigma_y(r)$ and $q_1 = \pi_{X_1}\sigma_{y_1}(r_1)$ be queries in $\mathcal{Q}$. Then q_1 is said to be* more specific than q in Δ*, denoted by $q \preceq q_1$, if one of the following holds:*

1. *$y_1 \notin \pi_{Y_1}(r_1)$*
2. *$y \in \pi_Y(r)$, $y_1 \in \pi_{Y_1}(r_1)$, and $Y_1 \to X_1 \in \mathcal{F}^+$*
3. *All of the following hold:*

Cust	*Cid*	*Cname*	*Caddr*
	$\mathtt{c}_1$	John	Paris
	$\mathtt{c}_2$	Mary	Paris
	$\mathtt{c}_3$	Jane	Paris
	$\mathtt{c}_4$	Anne	Tours

Prod	*Pid*	*Ptype*
	$\mathtt{p}_1$	milk
	$\mathtt{p}_2$	beer

Sales	*Cid*	*Pid*	*Qty*
	$\mathtt{c}_1$	$\mathtt{p}_1$	10
	$\mathtt{c}_2$	$\mathtt{p}_2$	5
	$\mathtt{c}_2$	$\mathtt{p}_1$	1
	$\mathtt{c}_1$	$\mathtt{p}_2$	10

$\mathcal{F}$: $Cid \rightarrow CnameCaddr$
$Pid \rightarrow Ptype$
$CidPid \rightarrow Qty$

$\mathcal{I}$: $\pi_{Cid}(Sales) \subseteq \pi_{Cid}(Cust)$
$\pi_{Pid}(Sales) \subseteq \pi_{Pid}(Prod)$

Fig. 1. The database of the running example

(a) either $r = r_1$ or r_1 involves the fact table φ,
(b) $y \in \pi_Y(r)$, $y_1 \in \pi_Y(r_1)$, $Y_1 \rightarrow X_1 \notin \mathcal{F}^+$,
(c) $XY_1 \rightarrow X_1 \in \mathcal{F}^+$ and $Y_1 \rightarrow Y \in \mathcal{F}^+$,
(d) $yy_1 \in \pi_{YY_1}(r \bowtie r_1)$.

Example 2. In the context of Example 1, consider again the queries $q_1 = \pi_{Cid}\sigma_{\mathtt{Paris}}(Cust)$ and $q_2 = \pi_{Cid}\sigma_{\mathtt{Paris\,beer}}(Cust \bowtie Prod \bowtie Sales)$. Referring to Definition 2.3, we have: (*a*) $Cust \bowtie Prod \bowtie Sales$ involves the fact table $Sales$, (*b*) $\mathtt{Paris} \in \pi_{Caddr}(Cust)$, $\mathtt{Paris\,beer} \in \pi_{Caddr\,Ptype}(Cust \bowtie Prod \bowtie Sales)$ and $Caddr\,Ptype \rightarrow Cid \notin \mathcal{F}^+$, (*c*) $Cid\,Caddr\,Ptype \rightarrow Cid \in \mathcal{F}^+$, $Caddr\,Ptype \rightarrow Caddr \in \mathcal{F}^+$, and (*d*) $\mathtt{Paris\,beer} \in \pi_{Caddr\,Ptype}(Cust \bowtie Prod \bowtie Sales)$. Therefore, $q_1 \preceq q_2$.

Consider now $q_2' = \pi_{Cname}\sigma_{\mathtt{c_2\,beer}}(Cust \bowtie Prod \bowtie Sales)$. Then, by Definition 2.2, $q_1 \preceq q_2'$, because $\mathtt{Paris} \in \pi_{Caddr}(Cust)$, $\mathtt{c_2\,beer} \in \pi_{Cid\,Ptype}(Cust \bowtie Prod \bowtie Sales)$ and $Cid\,Ptype \rightarrow Cname \in \mathcal{F}^+$.

For $q_3 = \pi_{Cid\,Cname}\sigma_{\top}(Cust \bowtie Sales)$, as above, $q_3 \preceq q_2$ holds. For $q_3' = \pi_{Cid\,Cname\,Caddr}\sigma_{\top}(Cust \bowtie Prod \bowtie Sales)$, applying again Definition 2.3, it can be seen that $q_3 \preceq q_3'$ and $q_3' \preceq q_3$ hold.

For $q_4 = \pi_{Qty}\sigma_{\mathtt{beer\,15}}(Prod \bowtie Sales)$, by Definition 2.1, we find $q_1 \preceq q_4$, $q_2 \preceq q_4$ and $q_3 \preceq q_4$, because $\mathtt{beer\,15} \notin \pi_{Ptype\,Qty}(Prod \bowtie Sales)$.

For $q_5 = \pi_{Qty}\sigma_{\mathtt{beer\,5}}(Prod \bowtie Sales)$, we have $q_1 \preceq q_5$, $q_2 \preceq q_5$ and $q_3 \preceq q_5$. Indeed, by Definition 2.2, $\mathtt{Paris} \in \pi_{Caddr}(Cust)$, $\mathtt{beer\,5} \in \pi_{Ptype\,Qty}(Prod \bowtie Sales)$ and $Ptype\,Qty \rightarrow Qty$ is in $\mathcal{F}^+$. Notice that, by Definition 2.1, we also have $q_5 \preceq q_4$.

It has been shown in [10] that the relation $\preceq$ is indeed a pre-ordering (*i.e.,* reflexive and transitive), with respect to which the support is anti-monotonic, *i.e.,* $(\forall q, q_1 \in \mathcal{Q})(q \preceq q_1 \Rightarrow sup(q_1) \leq sup(q))$.

Clearly, this property is required when mining patterns according to a level-wise algorithm, such as Apriori ([1]). Moreover, the pre-ordering $\preceq$ induces an equivalence relation defined as follows: two queries q and q_1 in $\mathcal{Q}$ are said to be *equivalent*, denoted by $q \equiv q_1$, if $q \preceq q_1$ and $q_1 \preceq q$ hold. The equivalence class of a query q is denoted by $[q]$.

Referring back to Example 2, the queries q_3 and q_3' are equivalent, since it has been seen that $q_3 \preceq q_3'$ and $q_3' \preceq q_3$ both hold.

As a consequence of the anti-monotonicity property mentioned above, it turns out that equivalent queries have the *same* support. Therefore, instead of computing the supports of individual queries, we consider only *one* query per equivalence class.

The pre-ordering $\preceq$ is extended to the set of equivalence classes $\mathcal{C}$, and then becomes an *ordering* (*i.e.,* reflexive, anti-symmetric and transitive) over $\mathcal{C}$. Moreover, a class $[q]$ is said to be *frequent* if its support (*i.e.,* the support of all queries in $[q]$) is greater than or equal to *min-sup*.

It is easy to see that all queries $q = \pi_X \sigma_y(r)$ in $\mathcal{Q}$ such that $y \notin \pi_Y(r)$ are equivalent and have a support equal to 0, a value meant to be less than the support threshold *min-sup*. Similarly, all queries $q = \pi_X \sigma_y(r)$ in $\mathcal{Q}$ such that $y \in \pi_Y(r)$ and $Y \rightarrow X \in \mathcal{F}^+$ are equivalent, and have a support equal to 1, another value meant to be less than *min-sup*. Thus, these equivalence classes, respectively denoted by C_0 and C_1, are not considered in the computation of frequent queries.

Equivalence classes different than C_0 and C_1, whose set is denoted by $\mathcal{C}^*$, have been characterized in [10]. We simply recall that, given a query $q = \pi_X \sigma_y(r)$ such that $[q]$ is in $\mathcal{C}^*$, we consider the representative $q^+ = \pi_{X'} \sigma_{y'}(r')$ of $[q]$ such that:

1. $X' = (XY)^+$ and $Y' = Y^+$,
2. $r' = r$ if r is a dimension table, otherwise, $r' = J$ where J is the join of all tables in Δ,
3. y' is the tuple over Y^+ such that y is a subtuple of y' and $y' \in \pi_{Y^+}(r')$.

In the remainder of the paper, all considered queries are assumed to satisfy the properties above, and stand for their equivalence classes.

Example 3. In Example 1, we have $J = (Cust \bowtie Prod \bowtie Sales)$. As seen in Example 2, $q_3 = \pi_{Cid\,Cname}\sigma_\top(Cust \bowtie Sales)$ and $q_3' = \pi_{Cid\,Cname\,Caddr}\sigma_\top(J)$ are equivalent. As $(Cid\,Cname)^+ = (Cid\,Cname\,Caddr)$ and $\emptyset^+ = \emptyset$, $[q_3]$ is represented by q_3'. It can be seen that $[q_3]$ is the set of all queries $\pi_X \sigma_\top(r)$ such that $Cid \subseteq X \subseteq (Cid\,Cname\,Caddr)$, and either $r = J$ or $r = (Cust \bowtie Sales)$.

For $q = \pi_{Cname\,Ptype}\sigma_{\mathtt{p}_2}(J)$, we have $(Cname\,Pid\,Ptype)^+ = (Cname\,Pid\,Ptype)$, $Pid^+ = (Pid\,Ptype)$, and $\mathtt{p}_2\,\mathtt{beer} \in \pi_{Pid\,Ptype}(J)$. Thus, $[q]$ is represented by $\pi_{Cname\,Pid\,Ptype}\sigma_{\mathtt{p}_2\,\mathtt{beer}}(J)$, and this class is the set of all queries $\pi_X \sigma_y(r)$ such that $Cname \subseteq X \subseteq (Cname\,Pid\,Ptype)$, and

- either $y = \mathtt{p}_2$ and $r = (Cust \bowtie Sales)$ or $r = J$
- or $y = \mathtt{p}_2\,\mathtt{beer}$ and $r = J$.

4 Algorithms

4.1 Main Algorithm: FQF

As in [10], frequent classes in $\mathcal{C}^*$ are computed by a level-wise algorithm, called *Frequent Query Finder* (FQF), whose main steps are shown in Figure 2: all dimension tables are first mined, and then the join J of all tables in Δ is mined. Moreover, as in [10], we define the notion of *generic class* to avoid generating classes that are processed in the same way.

Algorithm FQF

Input: The database Δ associated to an N-dimensional star schema and a support threshold *min-sup*.
Output: The set $Freq$ of all frequent classes.
Method:
$Freq = \emptyset$
`for` $i = 1, \ldots, N$ `do`
  `mine`(δ_i, $Freq(\delta_i)$)
  $Freq = Freq \cup Freq(\delta_i)$
`compute` $J = \delta_1 \bowtie \ldots \bowtie \delta_N \bowtie \varphi$
`mine`(J, $Freq(J)$)
$Freq = Freq \cup Freq(J)$
`return` $Freq$

Fig. 2. The main algorithm FQF

Definition 3. *Given a class $q = \pi_X \sigma_y(r)$ in $\mathcal{C}^*$, the* generic class *associated to q, denoted by $\langle X, Y, r\rangle$, is the set of all classes $\pi_X \sigma_{y'}(r)$ in $\mathcal{C}^*$ such that y' is a tuple in $\pi_Y(r)$,* i.e., $\langle X, Y, r\rangle = \{\pi_X \sigma_{y'}(r) \in \mathcal{C}^* \mid y' \in \pi_Y(r)\}$.

Algorithm `mine`, shown in Figure 3, follows a level-wise strategy ([1]). Namely, starting with the less specific generic class, that is r, the following steps are iterated until no frequent classes are generated:

1. Generate and prune the set C of candidate generic classes, based on the current set L of frequent generic classes (see [10]);
2. Compute the supports of all classes associated with the remaining candidate generic classes in C;
3. Discard all classes whose support is less than the support threshold;
4. Assign L to the set of all generic classes that contain at least one frequent class.

However, the steps above require more attention than in Apriori, because (i) we are dealing with equivalence classes, instead of individual itemsets, (ii) the ordering over $\mathcal{C}^*$ is more difficult to handle than set inclusion, and (iii) computing the supports requires to efficiently scan the database.

Consequently, the main difficulties are first, generating and pruning generic classes, and second, computing efficiently the supports of classes in $\mathcal{C}^*$. The first point has been addressed in [10] (see Proposition 7 in [10]), but not the second one, which is the main contribution of the present paper, and which we address below.

4.2 Algorithm `scan`

When scanning a given table r, the main difficulty is that every tuple in the answer to a query must be counted only *once*, whereas, due to projection, it might occur several times when scanning r. In order to cope with this difficulty, it is argued in [10] that indexing techniques are required. Unfortunately, considering such indexing techniques, which have to work for *all* possible attribute sets, is not realistic. In order to cope with this problem, in [9, 10], each scan is associated with huge

Algorithm `mine`

Input: A table r (either a dimension table δ_i or the join J) defined over R.
Output: The set $Freq(r)$ of all frequent classes in $\mathcal{C}^*$ of the form $\pi_X\sigma_y(r)$.
Method:

```
if |r| < min-sup then
    //no computation since, for every q in C* of the form π_X σ_y(r), |r| ≥ sup(q)
    Freq(r) = ∅
else //the computation starts with the generic class ⟨R, ∅, r⟩
    L = {⟨R, ∅, r⟩} ; Freq(r) = {π_R σ_⊤(r)}
    while L ≠ ∅ do
        //L is the set of frequent generic classes from the previous level
        C = generate(L, r)
        C = prune(C, L, r)
        scan(C, AUX(r), L, L_Freq(r))
        //L contains all frequent generic classes of the current level, and
        //L_Freq(r) is the corresponding set of frequent classes
        Freq(r) = Freq(r) ∪ L_Freq(r)
return Freq(r)
```

Fig. 3. Computing frequent queries on a table r

volumes of auxiliary data, resulting in main memory overflow for large fact tables.

Instead, in the present paper, before scanning r, we build an auxiliary table, denoted by AUX(r), as follows. Assuming that r contains n tuples $t_1, \ldots, t_n$, the first row AUX(r)[1] of AUX(r) is set to the empty set, and for every $i = 2, \ldots, n$, the ith element of AUX(r), denoted by AUX(r)[i], contains all maximal (with respect to set inclusion) attribute sets S for which there exists $j < i$ such that $t_j.S = t_i.S$. Therefore, when considering t_i during a scan of r, knowing that S is in AUX(r)[i] ensures that for every $X \subseteq S$, $t_i.X$ has already been processed.

The corresponding algorithm is shown in Figure 4, where $match(t_i, t_j)$ stands for the set of all attributes A such that $t_i.A = t_j.A$. We note that computing $match(t_i, t_j)$ amounts to compare t_i and t_j, which does not require using any index.

Example 4. We illustrate the construction of the auxiliary table AUX(J) in the context of Example 1, for $J = Cust \bowtie Prod \bowtie Sales$. The tables J and AUX(J) are shown in Figure 5.

Since $match(t_2, t_1) = Caddr$, we obtain AUX(J)[2] = $Caddr$. Similarly, since $match(t_3, t_1) = (Pid\, Caddr\, Ptype)$ and $match(t_3, t_2) = (Cid\ Cname\ Caddr)$, AUX($J$)[3] is the set of these two attribute sets.

The computation for AUX(J)[4] is similar, but although $match(t_4, t_3) = Caddr$, this schema does not appear in AUX(J)[4]. This is so because $Caddr$ is a subset of $(Cid\, Cname\, Caddr\, Qty)$ and $(Pid\, Caddr\, Ptype)$ that both belong to AUX(J)[4].

Now, given a table r and assuming that AUX(r) has been computed, the supports of equivalence classes over r are computed through parallel scans of r and AUX(r). The corresponding algorithm `scan` is shown in

Algorithm `aux`

Input: A table r to be scanned containing tuples $t_1, \ldots, t_n$.
Output: The table AUX(r).
Method:
AUX[1] = $\emptyset$
`for each` $i = 2, \ldots, n$ `do`
  AUX(r)[i] = $\emptyset$
  `for each` $j = 1, \ldots, i-1$ `do`
    `compute` $match(t_i, t_j)$
    `if` AUX(r)[i] contains no super set of $match(t_i, t_j)$ `then`
      AUX(r)[i] = AUX(r)[i] $\cup$ $match(t_i, t_j)$
`return` AUX(r)

Fig. 4. Computing the auxiliary table AUX(r)

J	*Cid*	*Pid*	*Cname*	*Caddr*	*Ptype*	*Qty*
t_1	c1	p1	John	Paris	milk	10
t_2	c2	p2	Mary	Paris	beer	5
t_3	c2	p1	Mary	Paris	milk	1
t_4	c1	p2	John	Paris	beer	10

AUX(J)	i	AUX(J)[i] ($1 \leq i \leq 4$)
	1	$\emptyset$
	2	*Caddr*
	3	(*Pid Caddr Ptype*), (*Cid Cname Caddr*)
	4	(*Cid Cname Caddr Qty*), (*Pid Caddr Ptype*)

Fig. 5. The table J and the associated table AUX(J) of Example 1

Figure 6. The input of Algorithm `scan` is a set C of candidate generic classes of the form $\langle X, Y, r\rangle$ for which r contains the tuples $t_1, \ldots, t_n$. All frequent classes associated with all generic candidate classes in C are computed as follows: For every $i = 1, \ldots, n$, the following actions are performed, for every $\langle X, Y, r\rangle$ in C:

1. If AUX(r)[i] contains a super schema of X, then $t_i.X$ has been encountered for some $j < i$. Thus $t_i.X$ has already been processed for all classes with a projection over X. Otherwise, $t_i.X$ is encountered for the first time, and thus, has to be processed.
2. In the latter case, $t_i.X$ has to be counted for the support of $q = \pi_X \sigma_{t_i.Y}(r)$. Two cases are then possible:
   (a) If AUX(r)[i] contains a super schema of Y then q has been processed previously, and thus is already associated with $\langle X, Y, r\rangle$. In this case, the support of q is incremented.
   (b) Otherwise, q is processed for the first time, and so, is not associated with $\langle X, Y, r\rangle$. In this case, we check if q can be pruned (see below), and if not, its support is initialized to 1 and q is associated with $\langle X, Y, r\rangle$.

Algorithm `scan`

Input: The set C of candidate generic classes, the table AUX(r).
Output: The set L of frequent generic classes in C, and the associated frequent classes $L_{Freq}(r)$.
Method:

$L = \emptyset$; $L_{Freq}(r) = \emptyset$
`for each` $\langle X, Y, r\rangle \in C$ `do`
  $L(\langle X, Y, r\rangle) = \emptyset$
`for each` $i = 1, \ldots, n$ `do` //r contains tuples $t_1, \ldots, t_n$
  `for each` $\langle X, Y, r\rangle \in C$ `do`
    `if` $\exists\ X' \in$ AUX(r)[i] such that $X \subseteq X'$ `then`
      //$t_i.X$ has been encountered before, and thus has been counted
      `nothing to do`
    `else`
      `if` $\exists\ Y' \in$ AUX(r)[i] such that $Y \subseteq Y'$ `then`
        //$\pi_X\sigma_{t_i.Y}(r)$ has already been encountered, thus
        //$t_i.X$ must be counted for the support of $\pi_X\sigma_{t_i.Y}(r)$
        $sup(\pi_X\sigma_{t_i.Y}(r)) = sup(\pi_X\sigma_{t_i.Y}(r)) + 1$
      `else`
        //$\pi_X\sigma_{t_i.Y}(r)$ has not been encountered before,
        //thus either prune it or initialize its support
        `if not(pruneQuery`$(\pi_X\sigma_{t_i.Y}(r))$`) then`
          $sup(\pi_X\sigma_{t_i.Y}(r)) = 1$
          $L(\langle X, Y, r\rangle) = L(\langle X, Y, r\rangle) \cup \{\pi_X\sigma_{t_i.Y}(r)\}$
`for each` $\langle X, Y, r\rangle \in C$ `do`
  $L(\langle X, Y, r\rangle) = L(\langle X, Yr\rangle) \setminus \{\pi_X\sigma_y(r) \mid sup(\pi_X\sigma_y(r)) < min\text{-}sup\}$
  `if` $L(\langle X, Y, r\rangle) \neq \emptyset$ `then`
    $L_{Freq}(r) = L_{Freq}(r) \cup L(\langle X, Y, r\rangle)$
    $L = L \cup \{\langle X, Y, r\rangle\}$
`return` L `and` $L_{Freq}(r)$

Fig. 6. Scanning the table r

Once these actions are performed, all supports of all classes that have to be computed are known. All classes whose support is greater than or equal to *min-sup* are put in $L_{Freq}(r)$ and the set L of frequent generic classes is output.

In our algorithms, pruning is performed at two distinct levels: for generic classes in Algorithm `mine`, and for classes in Algorithm `scan`. In Algorithm `mine`, a generic class $\langle X, Y, r\rangle$ is pruned if at least one of its predecessors (according to $\preceq$) contains no frequent classes, which entails that no class in $\langle X, Y, r\rangle$ can be frequent. However, if $\langle X, Y, r\rangle$ is not pruned, it may happen that a particular class $\pi_X\sigma_y(r)$ of $\langle X, Y, r\rangle$ can be pruned. This is checked in Algorithm `scan` as mentioned in item 2(b) above, according to Algorithm `pruneQuery` shown in Figure 7.

It is important to note that Proposition 7 of [10] shows that this latter pruning is *partial*, in the sense that not all predecessors of the class $\pi_X\sigma_y(r)$ are tested. We opted for such a partial pruning for efficiency reasons, as processing a complete pruning would damage performance.

Algorithm `pruneQuery`

Input: A class $q = \pi_X \sigma_y(r)$.
Output: boolean.
Method:
`if` there exist $A \in Y$ and $a \in dom(A)$ such that
$q = \pi_X \sigma_{y'}(r) \notin L_{Freq}(r)$ and $y = y'a$ `then`
`return` true;
`return` false;

Fig. 7. Class Pruning

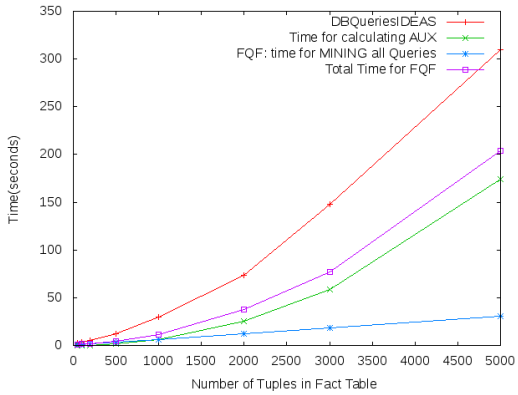

Fig. 8. Runtime over the size of the fact table for FQF and the implementation of [10].

5 Experiments

We performed experiments on an Pentium Duo Core with 2Go main memory running on Ubuntu Linux 2.6. The algorithms are implemented in Java using JDBC to communicate with MySql. Datasets have been generated using our own generator, adapted from the IBM data generator (`www.almaden.ibm.com`).

The generated databases over star schemas are denoted by db`d`Da`a`T`t`M`m` where `d` is the number of dimensions, `a` is the total number of attributes, `t` is the number of tuples in the fact table, and `m` is the number of measure attributes. In all our experiments, except those reported in Figures 11 and 12, the support threshold is set to 0.6 times the number of tuples in the fact table, that is $0.6 \times$ `t`. We also mention that all runtimes reported below include the computation time of the construction of auxiliary tables. Moreover, in Figures 8 and 9, the runtimes excluding the computation of the auxiliary tables are also shown.

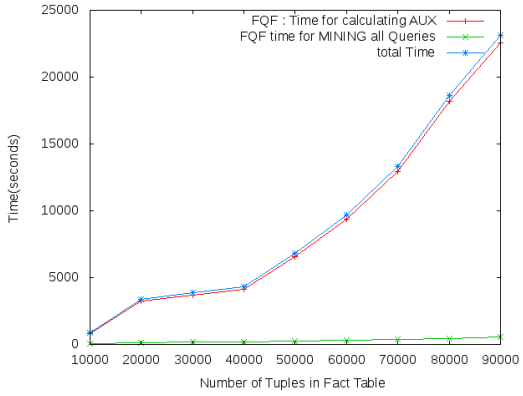

Fig. 9. Runtime over the size of the fact table.

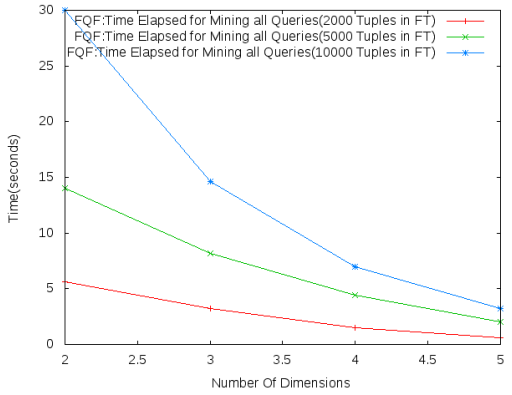

Fig. 10. Runtime over the number of dimensions.

Figure 8 shows the runtimes of FQF compared to those presented in [10] for db2D12TtM1, with `t` between 50 and 5000. Clearly, FQF outperforms our previous implementation presented in [10]: the reduction of runtime between the implementation in [10] and FQF is always greater than 33%. In should also be noticed from Figure 8 that the runtime for only mining frequent classes is very low, since less than 40 seconds.

Similarly, as shown in Figure 9, the time spent in mining the frequent classes for the databases db2D12TtM1 with `t` between 10,000 and 90,000 is very low compared to that for calculating the auxiliary tables. Moreover, this runtime increases slowly with the size of the fact table. We also emphasize that, in these experiments, we had no main memory overflow,

contrary to what happened with the previous implementation presented in [10], when `t` exceeds 5000.

Figure 10 reports on runtime over the number of dimensions, for the databases db`d`D12T`t`M1 where `d` ranges from 2 to 5 and for `t` equal to 2000, 5000 and 10,000. This figure clearly shows that the time spent for mining frequent classes decreases significantly when the number of dimensions increases. This is so because, given a number of attributes (12 in our case), when `d` increases, more functional dependencies are available, and so, less classes have to be processed. It is important to note that, according to our previous statement that the number of scans of the database is in $O(N \times |U|)$ (where N is the number of dimensions and $|U|$ the total number of attributes), one would rather expect an *increase* of runtime when N increases. However, what these experiments show is that, although the increase of the number of dimension tables entails more scans, this is compensated by a drastic reduction of the number of generic classes.

Figure 11 shows the runtime over the support threshold (expressed as a ratio of the size of the fact table), for databases db5D25T`t`M1 with `t` equal to 2000, 5000 and 10,000. Clearly, runtime decreases rapidly when the support threshold increases.

We recall from Section 2 that the only other work aiming at mining all frequent queries from a relational database is that in [7], and thus, we could compare our algorithm only to the Conqueror algorithm ([7]). This has been done using the IMDB database (`http://www.imdb.com`), for various support thresholds (expressed in numbers of tuples). To do so, we first transformed the IMDB database into a star schema having 3 dimensions, 6 attributes and no measure. In this experiment, the fact table contains 158,441 tuples.

As shown in Figure 12, our algorithm performs better than the Conqueror algorithm. This is so because, in [7], functional and inclusion dependencies are not taken into account, as we do in our approach. However, we recall in this respect that selection conditions of the form $Y = Y'$ (where Y and Y' are attribute sets) are considered in [7], which is not the case in our approach.

We end this section by two important remarks regarding the computation of the auxiliary tables.

1. The computation of auxiliary tables can be seen as a pre-processing, because it has to be computed only *once* for all runs of FQF, provided that, meanwhile, the database has not been updated. This remark is very important regarding runtime, because, as shown in Figures 8 and 9, assuming that auxiliary tables are available, the runtime of FQF is very low even for large fact tables.
2. When a database table r is updated, maintaining up to date the associated auxiliary table AUX(r) can be achieved efficiently. Indeed, in the case of insertion of a new tuple t in r, and assuming that t becomes the last tuple of r, a new row is added to AUX(r) and the associated schemas are obtained through one scan of r. If a tuple t_i is deleted from r, then AUX(r)[i] must be deleted from AUX(r), and only the rows AUX(r)[j] such that $j > i$ and $match(t_j, t_i) \in$ AUX(r)[j] have to be updated.

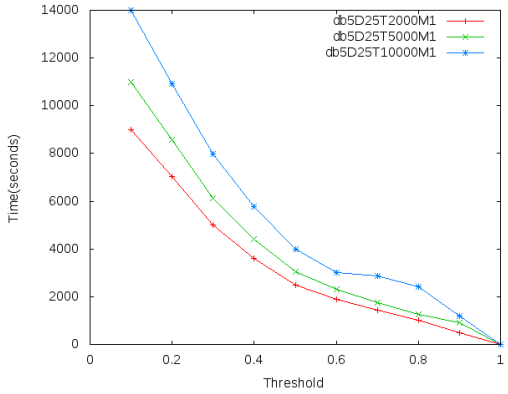

Fig. 11. Runtime over support.

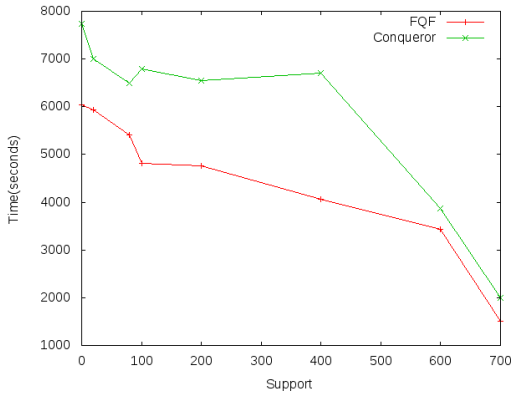

Fig. 12. FQF versus Conqueror.

6 Conclusion and Further Work

We presented new algorithms for mining frequent queries in databases over a star schema, based on theoretical results introduced in our previous work [10]. We showed through experiments that, in this particular case, mining frequent conjunctive queries becomes tractable. Our approach relies on the computation of auxiliary tables that can be seen as a pre-processing phase. An important point in this respect is that, assuming these auxiliary tables are available, the time for mining frequent queries becomes very low, as shown in our experiments.

Future work consists in processing further tests and optimizing our algorithms. We also plan to generalize our approach to database schemas

other than star schemas, and to study the possible rules that can be obtained based on frequent queries.

References

1. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*. MIT Press, 1996.
2. L. Dehaspe and L. D. Raedt. Mining association rules in multiple relations. In *7th Intl Workshop on Inductive Logic Programming, LNCS 1297*. Springer Verlag, 1997.
3. C. Diop, A. Giacometti, D. Laurent and N. Spyratos. Composition of mining contexts for efficient extraction of association rules. In *EDBT'02, LNCS 2287*. Springer Verlag, 2002.
4. R. Esposito, R. Meo and M. Botta. Answering constraint-based mining queries on itemsets using previous materialized results. *J. of Intelligent Information Systems*, 26(1), 2006.
5. B. Goethals and J. V. den Bussche. Relational association rules: getting warmer. In *ESF Exploratory Workshop on Pattern Detection and Discovery in Data Mining, LNCS 2447*. Springer-Verlag, 2002.
6. B. Goethals, E. Hoekx, and J. V. den Bussche. Mining tree queries in a graph. In *11th ACM SIGKDD Intl Conference on Knowledge Discovery and Data Mining (KDD)*, 2005.
7. B. Goethals, W. L. Page, and H. Mannila. Mining association rules of simple conjunctive queries. In *SIAM*, 2008.
8. J. Han, Y. Fu, W. Wang, K. Koperski, and O. Zaiane. Dmql : A data mining query language for relational databases. In *SIGMOD-DMKD'96*, 1996.
9. T.-Y. Jen, D. Laurent, and N. Spyratos. Mining all frequent selection-projection queries from a relational table. In *EDBT'08*. ACM Press, 2008.
10. T.-Y. Jen, D. Laurent, N. Spyratos. Mining Frequent Queries in Relational Databases. In *IDEAS*. ACM Press, 2009.
11. T.-Y. Jen, D. Laurent, N. Spyratos, and O. Sy. Towards mining frequent queries in star schemes. In *Intl Workshop on Knowledge Discovery in Databases (KDID), LNCS 3933*. Springer Verlag, 2005.
12. R. Meo, G. Psaila, and S. Ceri. An extension to sql for mining association rules. *Data Mining and Knowledge Discovery*, 9, 1997.
13. J. Ullman. *Principles of Databases and Knowledge-Base Systems*. Comp. Sc. Press, 1988.