# Frequent Query Computation in a Star Schema

Cheikh Tidiane Dieng[*,**], Tao-Y. Jen[*], Dominique Laurent[*]

[*] ETIS - CNRS - ENSEA - Université de Cergy Pontoise FRANCE
{jen,dlaurent}@u-cergy.fr
[**] Laboratoire d'Analyse Numérique et Informatique
Université Gaston-Berger - Saint-Louis SENEGAL
tidiane.dieng@gmail.com

**RÉSUMÉ.** Le problème de la recherche de requêtes fréquentes dans les bases de données relationnelles est connu comme étant très difficile, même si on se limite aux requêtes conjonctives sur une seule table. Cependant, nous montrons qu'en utilisant les dépendances fonctionnelles et les dépendances d'inclusion, le calcul des requêtes conjonctives fréquentes est envisageable dans le cas des bases de données organisées selon un schéma étoile. Plus précisément, nous montrons que les requêtes fréquentes de la forme projection-sélection-jointure, pour lesquelles la jointure contient la table de faits, peuvent être calculées efficacement par un algorithme par niveau de type Apriori.
Pour ce faire, nous présentons dans cet article un nouvel algorithme, appelé Frequent Query Finder (FQF), et nous montrons par des tests que cet algorithme permet le calcul des requêtes fréquentes de manière très efficace.

**ABSTRACT.** The problem of computing frequent queries in relational databases is known to be intractable, even when only conjunctive queries are considered. However, we show that using functional and inclusion dependencies, computing frequent conjunctive queries becomes feasible for databases operating over a star schema. More precisely, it is shown that frequent projection-selection-join queries, for which the join contains the fact table, can be efficiently computed, using a level-wise algorithm such as Apriori.
To this end, we introduce in this paper, a novel algorithm, called Frequent Query Finder (FQF), and we report on experiments showing that our algorithm allows for a particularly efficient computation of frequent queries.

**MOTS-CLÉS :** Requêtes Fréquentes, Dépendances Fonctionnelles, Dépendances d'Inclusion, Comparaison de Requêtes.

**KEYWORDS :** Frequent Queries, Functional Dependencies, Inclusion Dependencies, Query Comparison.

## 1. Introduction

The problem of discovering frequent queries in a (relational) database is one of a main topics in data mining. However, this problem is known to be intractable, even when only conjunctive queries are considered, due to the fact that the size of the search space is exponential in the size of the database.

On the other hand, as argued in [4, 5], mining all frequent conjunctive queries from a given database becomes tractable if constraints on the database, such as functional dependencies and inclusion dependencies, are taken into account. Indeed, it has been shown in [4, 5] that such dependencies allow for comparing queries according to a pre-ordering with respect to which the support measure is anti-monotonic. As a consequence, a level-wise algorithm such as Apriori ([1]) can be used, with the basic additional following feature : the considered pre-ordering between queries 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.

We refer to [3, 4, 5] for related work. We simply mention here that the work in [2, 3] is closely related to ours, because in [2, 3], frequent conjunctive queries are mined in relational databases with several relations. However, in this work, no dependency is involved in query comparison, as opposed to our work.

The main contribution of the paper is, based on the approach presented in [5], to propose a novel algorithm for the computation of frequent queries, called *Frequent Query Finder* (FQF) in the case where the database to be mined operates on a star schema.

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

## 2. Formal Model

### 2.1. Queries

We first recall that a database $\Delta$ over a *star schema* consists of a distinguished table $\varphi$ 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 $\varphi$. In other words, for every $i = 1, \ldots, N$, $\delta_i$ satisfies the functional dependency $K_i \rightarrow D_i$ and $\varphi$ satisfies the functional dependency $K \rightarrow F$. 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 $\varphi$). 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 $X \rightarrow A$ is in $\mathcal{F}^+$ ([6]).

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 selection condition is either the *empty tuple*, denoted by $\top$ (meaning that all tuples are selected), or a tuple $y$ over the (intended) relation schema $Y$ (meaning that 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 $\varphi$.

**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 $\Delta$ or a join of tables in $\Delta$ containing the table $\varphi$ ;*

*– $y$ is either the empty tuple $\top$ or a tuple over the relation schema $Y$.*

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

## 2.2. Query Comparison

As in [5], queries in $\mathcal{Q}$ are compared according to the following pre-ordering.

**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 $\Delta$*, 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 $Y_1 \rightarrow X_1 \in \mathcal{F}^+$*

*3) All of the following hold :*

*a) either $r = r_1$ or $r_1$ involves the fact table $\varphi$,*

*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) $yy_1 \in \pi_{YY_1}(r \bowtie r_1)$.*

It has been seen in [5] that the relation $\preceq$ is indeed a pre-ordering (*i.e.,* reflexive and transitive), with respect to which the support is anti-monotonic. That is, for all queries $q$ and $q_1$ in $\mathcal{Q}$, we have : $q \preceq q_1 \Rightarrow sup(q_1) \leq sup(q)$.

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 $[q]$.

As a consequence of the anti-monotonicity property, 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. On the other hand, 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 $\mathcal{C}$. Consequently, 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*.

Moreover, 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, denoting these equivalence classes by $C_0$ and $C_1$, respectively, these classes are not considered in the computation.

**Algorithm FQF**

**Input :** The database $\Delta$ 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`($\delta_i$, *min-sup*, $Freq(\delta_i)$)
    $Freq = Freq \cup Freq(\delta_i)$
`compute` $J = \delta_1 \bowtie \ldots \bowtie \delta_N \bowtie \varphi$
`mine`($J$, *min-sup*, $Freq(J)$)
$Freq = Freq \cup Freq(J)$
`return` $Freq$

**Figure 1.** *The main algorithm FQF*

On the other hand, equivalence classes different than $C_0$ and $C_1$, whose set is denoted by $\mathcal{C}^*$, have been characterized in [5]. We simply recall that, given a query $q = \pi_X \sigma_y(r)$ such that $[q]$ is in $\mathcal{C}^*$, we consider the representative $q^+ = \pi_{X'} \sigma_{y'}(r')$ such that :

1) $X' = (XY)^+$ and $Y' = Y^+$,

2) $r' = r$ if $r$ is a dimension table or, otherwise, $r'$ is the join of all tables in $\Delta$,

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.

# 3. Algorithms

## 3.1. Main Algorithm : FQF

As in [5], frequent classes in $\mathcal{C}^*$ are computed by a level-wise algorithm, called *Frequent Query Finder* (FQF) whose main steps are shown in Figure 1 : all dimension tables are first mined, and then the join $J$ of all tables in $\Delta$ is mined. Moreover, as in [5], we define the notion of *generic class* in order to avoid generating classes that are processed in the same way.

**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 2, follows a level-wise strategy ([1]). Namely, starting with the less specific generic class, that is $r$, iterate the following steps until no frequent classes are generated : $(i)$ generate and prune the set of candidate classes (see [5]), $(ii)$ compute the supports of the remaining candidate classes, and $(iii)$ eliminate all classes whose support is less than the support threshold.

## 3.2. Algorithm `scan`

We note that scanning a relation for the computation of frequent classes over a given table $r$ has to be efficient. This is so because such a scan is required at each level of

**Algorithm** `mine`

**Input :** A table $r$ (either a dimension table $\delta_i$ 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
        C = generate(L, r)
        C = prune(C, L, r)
        scan(C, L, L_Freq(r))
        //all generic classes of L are instanciated and L_Freq(r) is the set of frequent classes in L
        Freq(r) = Freq(r) ∪ L_Freq(r)
return Freq(r)
```

**Figure 2.** *Computing frequent queries on a table* $r$

candidate generation, and this for all dimension tables and for the join table $J$. The main difficulty is that every tuple in the answer to a query must be counted only *once*, whereas it might occur several times when scanning $r$.

In order to cope with this difficulty, before scanning $r$, we associate $r$ with an auxiliary table, denoted by AUX($r$), and that contains a set of schemas associated to each tuple in $r$. Assuming that $r$ contains $n$ tuples $t_1, \ldots, t_n$, the first element AUX$(r)[1]$ of AUX($r$) is set to the empty set, and for every $i = 2, \ldots, n$, the $i$th element of AUX($r$), denoted by AUX$(r)[i]$, contains all schemas $S$ such that $(i)$ AUX$(r)[i]$ contains no subset of $S$, and $(ii)$ there exists $j < i$ such that $t_j.S = t_i.S$. The corresponding algorithm is shown in Figure 3, where $match(t_i, t_j)$ stands for the set of all attributes $A$ such that $t_i.A = t_j.A$.

**Algorithm 1**

**Input :** A table $r$ to be scanned containing tuples $t_1, \ldots, t_n$.
**Output :** The table AUX($r$).
**Method :**

```
AUX[1] = ∅
for each i = 2, ..., n do
    AUX(r)[i] = ∅
    for each j = 1, ..., 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] ∪ match(t_i, t_j)
return AUX(r)
```

**Figure 3.** *Computing the auxiliary table AUX*$(r)$

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 Figure 4. At a given level of Algorithm `scan`,

we have a set $C$ of candidate generic classes of the form $\langle X, Y, r\rangle$ for which $r$ contains the tuples $t_1, \ldots, t_n$. The goal of Algorithm `scan` is then to compute all frequent classes associated with a generic class in $C$. To do so, for each $i = 1, \ldots, n$, 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, if AUX$(r)[i]$ contains a super schema of $Y$ then the query $q = \pi_X \sigma_{t_i.Y}(r)$ has been processed previously. Two cases are then possible :

a) Either $q$ is associated with $\langle X, Y, r\rangle$, in which case its support is incremented.

b) Or $q$ is not associated with $\langle X, Y, r\rangle$, in which case $q$ is processed for the first time. We check if $q$ can be pruned (see below), and if not, its support is initialized to $1$.

3) At this stage, 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 for the next level is updated.

Regarding the pruning, we recall that some pruning is performed in Algorithm `mine`, but for generic classes, whereas the pruning in Algorithm `scan` operates on classes. A generic class $\langle X, Y, r\rangle$ is pruned if at least one its predecessor generic classes (according to $\preceq$) contains no frequent classes. On the other hand, if $\langle X, Y, r\rangle$ is not pruned, it might happen that $\pi_X \sigma_y(r)$ in $\langle X, Y, r\rangle$ can be pruned. This is checked in Algorithm `scan` as mentioned in item 2(b) above, according to Algorithm `pruneQuery` shown in Figure 5.

## 4. Experiments

We performed experiments on an Pentium Duo Core with 2Go main memory running on Ubuntu Linux 2.6. The algorithm is written in Java using JDBC to communicate with MySql. Datasets have been generated using our generator, based on the one by IBM. The generated databases over star schemas, are denoted by $db$-$D$-$A$-$M$-$T$ where $D$ is the number of dimensions, $A$ is the number of attributes, $M$ is the number of measure attributes, and $T$ is the number of tuples in the fact table. Figure 6 shows the runtimes of FQF compared to those presented in [5] for $db$-2-12-1-$T$, with $T$ between $50$ and $5000$.

As shown in Figure 7, the time spent in mining the queries for the databases $db$-2-12-1-$T$ with $T$ between $1000$ and $100000$ is very low compared to that for calculating the auxiliary table. It is important to note that, in our experiments, we had no Out Of Memory exceptions, as with the algorithm presented in [5] when $T$ exceeds 6000. Moreover, as seen in Figure 8, the time spent for mining queries decreases when the number of dimensions increases. This is so because, given a number of attributes (12 in our case), when $T$ increases, more functional dependencies are available, and so, less classes are to be considered. We also point out that if the number of tuples increases, the time spent for mining queries is almost constant.

## 5. Conclusion and Further Work

We presented algorithms for mining frequent queries in database over a star schema, and we showed that they outperform those proposed in [5]. Our experiments show that the performance gap increases with the number of tuples, and that by using the AUX table,

**Algorithm** `scan`

**Input :** The set $C$ of candidate generic classes, the table AUX$(r)$, a given threshold *min-sup*.
**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`
  $L(\langle X, Y, r\rangle) = \emptyset$
`end for each`
`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` //$t_i.X$ has not been encountered before, and thus must considered
      `if` $\exists\ Y' \in$ AUX$(r)[i]$ such that $Y \subseteq Y'$ `then`
        //$t_i.X$ must be counted for the support of $\pi_X\sigma_{t_i.Y}(r)$
        `if` $\langle X, t.Y, r\rangle \in L(\langle X, Y, r\rangle)$ `then`
          //$\pi_X\sigma_{t_i.Y}(r)$ has already been encountered, thus update its support
          $sup(\pi_X\sigma_{t_i.Y}(r)) = sup(\pi_X\sigma_{t_i.Y}(r)) + 1$
        `else` //$\pi_X\sigma_{t_i.Y}(r)$ has not been encountered, thus prune 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-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` $L_{Freq}(r)$

**Figure 4.** *Scanning the table* $r$

**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 ;

**Figure 5.** *Class Pruning*

the time for mining queries become very low. Further work consists in processing further tests and optimizing the computation of AUX table by an incremental approach. We also plan to generalize our approach to database schemas other than star schemas.

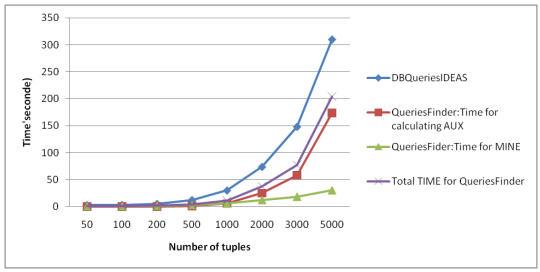


**Figure 6.** *Runtime Comparison.*

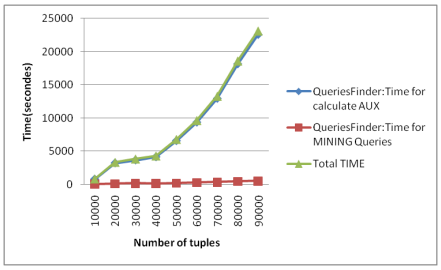


**Figure 7.** *Runtime over the number of tuples in fact table.*

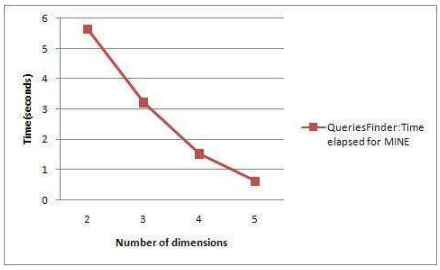


**Figure 8.** *Runtime over the number of dimensions (with 2000 tuples in fact table)*

## 6. Bibliographie


[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] 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.

[3] B. Goethals, W. L. Page, and H. Mannila. Mining association rules of simple conjunctive queries. In *SIAM*, 2008.

[4] T. Jen, D. Laurent, and N. Spyratos. Mining all frequent selection-projection queries from a relational table. In *EDBT'08*. ACM Press, 2008.

[5] T. Jen, D. Laurent, N. Spyratos. Mining Frequent Queries in Relational Databases. In *IDEAS*. ACM Press, 2009.

[6] J. Ullman. *Principles of Databases and Knowledge-Base Systems*. Comp. Sc. Press, 1988.