# Mining Frequent Patterns with Counting Inference


Yves Bastide[*] stid@acm.org

Rafik Taouil[†] Rafik.taouil@loria.fr

Nicolas Pasquier[‡] Nicolas.Pasquier@unice.fr

Gerd Stumme[§] stumme@aifb.uni-karlsruhe.de

Lotfi Lakhal[¶] lakhal@lim.univ-mrs.fr



## ABSTRACT

In this paper, we propose the algorithm PASCAL which introduces a novel optimization of the well-known algorithm Apriori. This optimization is based on a new strategy called *pattern counting inference* that relies on the concept of *key patterns*. We show that the support of frequent non-key patterns can be inferred from frequent key patterns without accessing the database. Experiments comparing PASCAL to the three algorithms Apriori, Close and Max-Miner, show that PASCAL is among the most efficient algorithms for mining frequent patterns.


## 1. INTRODUCTION

Knowledge discovery in databases (KDD) is defined as the non-trivial extraction of valid, implicit, potentially useful and ultimately understandable information in large databases [11]. For several years, a wide range of applications in various domains have benefited from KDD techniques and many work has been conducted on this topic. The problem of mining frequent patterns arose first as a sub-problem of mining association rules [1], but it then turned out to be present in a variety of problems [12]: mining sequential patterns [3], episodes [16], association rules [2], correlations [8; 23], multi-dimensional patterns [13; 14], maximal patterns [7; 26; 15], closed patterns [24; 20; 19; 21]. Since the complexity of this problem is exponential in the size of the binary database input relation and since this relation has to be scanned several times during the process, efficient algorithms for mining frequent patterns are required.

### 1.1 Related work

Three approaches have been proposed for mining frequent patterns. The first is traversing iteratively the set of all patterns in a levelwise manner. During each iteration corresponding to a level, a set of candidate patterns is created by joining the frequent patterns discovered during the previous iteration, the supports of all candidate patterns are counted and infrequent ones are discarded. The most prominent algorithm based on this approach is the Apriori algorithm [2]. A variety of modifications of this algorithm arose [9; 17; 22; 25] in order to improve different efficiency aspects. However, all of these algorithms have to determine the supports of *all* frequent patterns and of some infrequent ones in the database.

The second approach is based on the extraction of maximal[1] frequent patterns, from which all supersets are infrequent and all subsets are frequent. This approach combines a levelwise bottom-up traversal with a top-down traversal in order to quickly find the maximal frequent patterns. Then, all frequent patterns are derived from these ones and one last database scan is carried on to count their support. The most prominent algorithm using this approach is Max-Miner [7]. Experimental results have shown that this approach is particularly efficient for extracting maximal frequent patterns, but when applied to extracting all frequent patterns performances drastically decrease because of the cost of the last scan which requires roughly an inclusion test between each frequent pattern and each object of the database. As for the first approach, algorithms based on this approach have to extract the supports of all frequent patterns from the database.

The third approach, represented by the Close algorithm [20], is based on the theoretical framework introduced in [18] that uses the closure of the Galois connection [10]. In this approach, the frequent closed patterns (and their support) are extracted from the database in a levelwise manner. A closed pattern is the greatest pattern common to a set of objects of the database; and each non-closed pattern has the same properties (i.e. the same set of objects containing it and thus the same support) as its closure, the smallest closed pattern containing it. Then, all frequent patterns as well as their support are derived from the frequent closed patterns and their support without accessing the database. Hence not all patterns are considered during the most expensive part of the algorithm (counting the supports of the patterns) and the search space is drastically reduced, especially for strongly correlated data. Experiments have shown that this approach is much more efficient than the two previous ones on such data.


[*]LIMOS, université Blaise Pascal, complexe scientifique des Cézeaux, 24 av. des Landais, 63177 Aubière Cedex, France

[†]INRIA Lorraine, 54506 Vandœuvre-lès-Nancy Cedex, France

[‡]I3S (CNRS UPRESA 6070) - université de Nice, 06903 Sophia Antipolis, France

[§]Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe (TH), D-76128 Karlsruhe, Germany

[¶]LIM (CNRS FRE 2246) - université de la Méditerranée, 13288 Marseille Cedex 09, France


[1]'Maximal' means 'maximal with respect to set inclusion'.

## 1.2 Contribution

We present the PASCAL[2] algorithm, introducing a novel, effective and simple optimization of the Apriori algorithm. This optimization is based on *pattern counting inference* that relies on the new concept of *key patterns*. A key pattern is a minimal pattern of an equivalence class gathering all patterns that have the same objects[3]. The pattern counting inference allows to determine the supports of *some* frequent and infrequent patterns (the key patterns) in the database only. The supports of all other frequent patterns are derived from the frequent key patterns. This allows to reduce, at each database pass, the number of patterns considered, and, even more important, to reduce the number of passes in total. This optimization is valid since key patterns have a property that is compatible with the original Apriori (deterministic) heuristic; we show that all subsets of a key pattern are key patterns and all supersets of a non-key pattern are non-key patterns. Then, the counting inference is performed in a levelwise manner: If a candidate pattern of size $k$ which support has to be determined is a non-key pattern, then its support is equal to the minimal support among the patterns of size $k$-1 that are its subsets. In comparison to most other modifications of Apriori, this induces a minimal impact on the understandability and simplicity of implementation of the algorithm. The important difference is to determine as much support counts as possible without accessing the database by information gathered in previous passes. As shown by the experiments, the efficiency gain is up to one order of magnitude on correlated data.

## 1.3 Organization of the paper

In the next section, we recall the problem of mining frequent patterns. The essential notions and definitions of key patterns and pattern counting inference are given in Section 3. The PASCAL algorithm is described in Section 4 and experimental results for comparing its efficiency to those of Apriori, Max-Miner and Close are presented in Section 5. Section 6 concludes the paper.

# 2. STATEMENT OF THE PROBLEM

Let $\mathbb{P}$ be a finite set of *items*, $\mathbb{O}$ a finite set of *objects* (e. g., transaction ids) and $\mathbb{R} \subseteq \mathbb{O} \times \mathbb{P}$ a binary relation (where $(o, p) \in \mathbb{R}$ may be read as "item $p$ is included in transaction $o$"). The triple $\mathbb{D} = (\mathbb{O}, \mathbb{P}, \mathbb{R})$ is called *dataset*.

Each subset $P$ of $\mathbb{P}$ is called a *pattern*. We say that a pattern $P$ is *included* in an object $o \in \mathbb{O}$ if $(o, p) \in \mathbb{R}$ for all $p \in P$. Let $f$ be the function which assigns to each pattern $P \subseteq \mathbb{P}$ the set of all objects that include this pattern: $f(P) = \{o \in \mathbb{O} \mid o \text{ includes } P\}$.

The *support* of a pattern $P$ is given by: $\sup(P) = \operatorname{card}(f(P))/\operatorname{card}(\mathbb{O})$. For a given threshold *minsup* $\in [0, 1]$, a pattern $P$ is called *frequent pattern* if $\sup(P) \geq \mathit{minsup}$.

The task of mining frequent patterns consists in determining all frequent patterns together with their supports for a given threshold minsup.

[2]The French mathematician Blaise Pascal (* Clermont-Ferrand 1623, † 1662 Paris) invented an early computing device.

[3]A similar notion of equivalence classes was also recently proposed by R. Bayardo and R. Agrawal [6] to characterize "a-maximal" rules (i.e., association rules with maximal antecedent).

# 3. PATTERN COUNTING INFERENCE

In this section, we give the theoretical basis of the new PASCAL algorithm. This basis provides at the same time the proof of correctness of the algorithm. In Section 4, these theorems will be turned into pseudo-code.

Like Apriori, PASCAL will traverse the powerset of $\mathbb{P}$ levelwise: At the $k^{\text{th}}$ iteration, the algorithm generates first all *candidate $k$-patterns*.

*Definition 1.* A $k$-pattern $P$ is a subset of $\mathbb{P}$ such that $\operatorname{card}(P) = k$. A *candidate $k$-pattern* is a $k$-pattern where all its proper sub-patterns are frequent.

Given the set of candidate $k$-patterns, one database pass is used to determine their support. Infrequent patterns are then pruned. This approach works because of the well-known fact that a pattern cannot be frequent if it has an infrequent sub-pattern.

## 3.1 Key Patterns

Our approach is based on the observation that frequent patterns can be considered as "equivalent" if they are included in exactly the same objects. We describe this fact by the following equivalence relation $\theta$ on the frequent patterns.

*Definition 2.* Given two patterns $P, Q \subseteq \mathbb{P}$, let $P \,\theta\, Q$ if and only if $f(P) = f(Q)$. The set of patterns which are *equivalent* to a pattern $P$ is given by $[P] = \{Q \subseteq \mathbb{P} \mid P \,\theta\, Q\}$.

In the case of patterns $P$ and $Q$ with $P \,\theta\, Q$, both patterns obviously have the same support:

LEMMA 1. *Let $P$ and $Q$ be two patterns.*
$(i)$ $\quad P \,\theta\, Q \implies \sup(P) = \sup(Q)$
$(ii)$ $\quad P \subseteq Q \wedge \sup(P) = \sup(Q) \implies P \,\theta\, Q$

PROOF. $(i)$ $\; P \,\theta\, Q \iff f(P) = f(Q) \implies \sup(P) = \operatorname{card}(f(P))/\operatorname{card}(\mathbb{O}) = \operatorname{card}(f(Q))/\operatorname{card}(\mathbb{O}) = \sup(Q)$.
$(ii)$ Since $P \subseteq Q$ and $f$ is monotonous decreasing, we have $f(P) \supseteq f(Q)$. $\sup(P) = \sup(Q)$ is equivalent to $\operatorname{card}(f(P)) = \operatorname{card}(f(Q))$ which implies with the former $f(P) = f(Q)$ and thus $P \,\theta\, Q$. ☐

Hence if we knew the relation $\theta$ in advance, we would need to count the support of only one pattern in each equivalence class. Of course this is not the case; but we can construct it step by step[4]. Thus, we will (in general) need to determine the support of more than one pattern in each class, but not of all of them. If we already have determined the support of a pattern $P$ in the database and encounter later a pattern $Q \in [P]$, then we need not access the database for it because we know that $\sup(Q) = \sup(P)$.

The first patterns of an equivalence class that we reach using a levelwise approach are exactly the minimal[5] patterns in the class:

*Definition 3.* A pattern $P$ is a *key pattern* if $P \in \min[P]$; that is, if no proper subset of $P$ is in the same equivalence class. A *candidate key pattern* is a pattern such that all its proper sub-patterns are frequent key patterns.

[4]In the algorithm, the equivalence relation is not explicitly generated, but is—as the algorithm is based on the following theorems—implicitly used.

[5]'Minimal' means 'minimal with respect to set inclusion'.

Observe that all candidate key patterns are also candidate patterns.
Figure 1 presents the lattice of frequent patterns found on an example database with a minimum support *minsup* = 2/5, highlighting key patterns and equivalence classes (the database is similar to the one described in the *running example* subsection of the PASCAL algorithm, except that the item $F$ was removed). In this database, for instance, every object containing $A$ also contains $C$; but not all of them contain $B$. Hence $\{A\}$ and $\{AC\}$ are in the same equivalence class—but not $\{AB\}$.

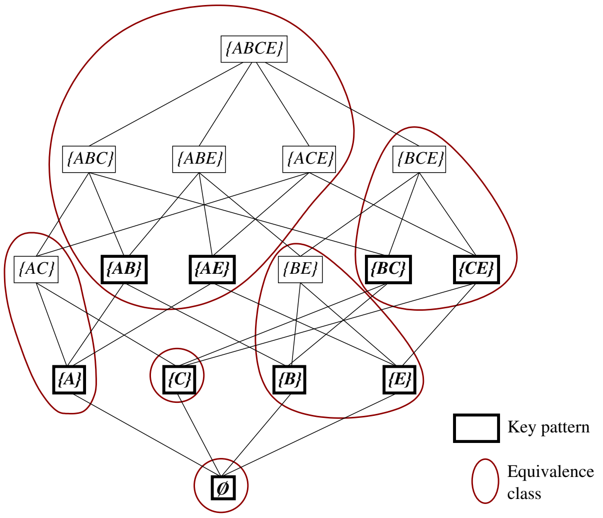


Figure 1: Example lattice of frequent patterns

## 3.2 Counting Inference

In the algorithm, we apply the pruning strategy to both candidate patterns and candidate key patterns. This is justified by the following theorem:

THEOREM 2. $(i)$ *If $Q$ is a frequent key pattern and $P \subseteq Q$, then $P$ is also a frequent key pattern.*
$(ii)$ *If $P$ is not a frequent key pattern and $P \subseteq Q$, then $Q$ is not a frequent key pattern either.*[6]

PROOF. We will first prove a more general statement of this theorem with no hypothesis on the support.
$(ii)$ Let $P$ and $Q$ be two patterns with $P \subseteq Q$ and $P$ not being a key pattern. Then there exists $P' \in \min[P]$ with $P' \subset P$. We need to show that $f(Q) = f(Q \setminus (P \setminus P'))$. We can rewrite $f(Q)$ as $f((Q \setminus (P \setminus P')) \cup (P \setminus P'))$, or $f(Q \setminus (P \setminus P')) \cap f(P \setminus P')$. We know that $f(P') \subseteq f(P \setminus P')$, because $f(P') = f(P)$ and $f$ is a decreasing function[7], so $f(Q) \supseteq f(Q \setminus (P \setminus P')) \cap f(P')$. This formula is equivalent to $f(Q) \supseteq f((Q \setminus (P \setminus P')) \cup P')$. Since naturally $P' \subseteq (Q \setminus (P \setminus P'))$, it follows that $f(Q) \supseteq f(Q \setminus (P \setminus P'))$. The opposite inequality is obvious. Hence, $Q$ is not minimal in $[Q]$ and thus by definition not a key pattern. $(i)$ is a direct logical consequence of $(ii)$.
When taking the support into account, it is sufficient to notice that all subsets of a frequent pattern are frequent patterns, so the proof remains valid. $\square$

[6] In mathematical terms, $(i)$ and $(ii)$ state that the set of frequent key patterns is an order ideal (or down-set) of $(2^{\mathbb{P}}, \subseteq)$.
[7] That is, $P_1 \subseteq P_2 \Rightarrow f(P_2) \subseteq f(P_1)$.

The algorithm determines, at each iteration, the key patterns among the candidate key patterns by using $(ii)$ of the following theorem:

THEOREM 3. *Let $P$ be a frequent pattern.*
$(i)$ *Let $p \in P$. Then $P \in [P \setminus \{p\}]$ iff* $\sup(P) = \sup(P \setminus \{p\})$.
$(ii)$ *$P$ is a key pattern iff* $\sup(P) \neq \min_{p \in P}(\sup(P \setminus \{p\}))$.

PROOF. $(i)$ The 'if' part follows from Lemma 1 $(ii)$. The 'only if' part is obvious. $(ii)$ From $(i)$ we deduce that $P$ is a key pattern iff $\sup(P) \neq \sup(P \setminus \{p\})$, for all $p \in P$. Since sup is a monotonous decreasing function, this is equivalent to $(ii)$. $\square$

As all candidate key patterns are also candidate patterns, when generating all candidate patterns for the next level we can at the same time determine the candidate key patterns among them.
If we reach a candidate $k$-pattern which is not a candidate key pattern, then we already passed along at least one of the key patterns in its equivalence class in an earlier iteration. Hence we already know its support. Using the following theorem, we determine this support *without accessing the database*:

THEOREM 4. *If $P$ is a non-key pattern, then*

$$\sup(P) = \min_{p \in P}(\sup(P \setminus \{p\})).$$

PROOF. "$\leq$" follows from the fact that sup is a monotonous decreasing function. "$\geq$": If $P$ is not a key pattern then exists $p \in P$ with $P \,\theta\, P \setminus \{p\}$. Hence $\sup(P) = \sup(P \setminus \{p\}) \geq \min_{q \in P}(\sup(P \setminus \{q\})$. $\square$

Thus the database pass needs to count the supports of the candidate key patterns only. Knowing this, we can summarize PASCAL as follows: It works exactly as Apriori, but counts only those supports in the database pass which cannot be derived from supports already computed. We can thus, on each level, restrict the expensive count in the database to some of the candidates. Better yet, from some level on, all candidate pattern may be known to be non-key patterns. Then all remaining frequent patterns and their support can be derived without accessing the database any more. In the worst case (i. e., in weakly correlated data), all candidate patterns are also candidate key patterns. The algorithm behaves then exactly as Apriori, with no significant overhead.

# 4. THE PASCAL ALGORITHM

In this section, we transform the theorems from the last section into an algorithm. The pseudo-code is given in Algorithm 1. A list of notations is provided in Table 1. We assume that $\mathbb{P}$ is linearly ordered, e. g., $\mathbb{P} = \{1, \ldots, n\}$. This will be used in PASCAL-GEN.
The algorithm starts with the empty set, which always has a support of 1 and which is (by definition) a key pattern (step

Table 1: Notations used in PASCAL

| | |
|---|---|
| $k$ | is the counter which indicates the current iteration. In the $k$th iteration, all frequent $k$-patterns and all key patterns among them are determined. |
| $\mathcal{P}_k$ | contains after the $k$th iteration all frequent $k$-patterns $P$ together with their support $P.\sup$, and a boolean variable $P$.key indicating if $P$ is a (candidate) key pattern. |
| $\mathcal{C}_k$ | stores the candidate $k$-patterns together with their support (if known), the boolean variable $P$.key, and a counter $P$.pred_sup which stores the minimum of the supports of all $(k-1)$-sub-patterns of $P$. |

**Algorithm 1** PASCAL

1) $\emptyset.\sup \leftarrow 1$; $\emptyset$.key $\leftarrow$ true;
2) $\mathcal{P}_0 \leftarrow \{\emptyset\}$;
3) $\mathcal{P}_1 \leftarrow$ {frequent 1-patterns};
4) **forall** $p \in \mathcal{P}_1$ **do begin**
5)   $p$.pred_sup $\leftarrow 1$; $p$.key $\leftarrow (p.\sup \neq 1)$;
6) **end**;
7) **for** $(k = 2;\ \mathcal{P}_{k-1} \neq \emptyset;\ k{+}{+})$ **do begin**
8)   $\mathcal{C}_k \leftarrow$ PASCAL-GEN$(\mathcal{P}_{k-1})$;
9)   **if** $\exists(c \in \mathcal{C}_k$ where $c$.key $=$ true$)$ **then**
10)     **forall** $o \in \mathbb{D}$ **do begin**
11)       $\mathcal{C}_o \leftarrow$ subset$(\mathcal{C}_k, o)$;
12)       **forall** $c \in \mathcal{C}_o$ where $c$.key $=$ true **do**
13)         $c.\sup{+}{+}$;
14)     **end**;
15)   **forall** $c \in \mathcal{C}_k$ **do**
16)     **if** $c.\sup \geq$ *minsup* **then begin**
17)       **if** $c$.key and $c.\sup = c$.pred_sup **then**
18)         $c$.key $\leftarrow$ false;
19)       $\mathcal{P}_k \leftarrow \mathcal{P}_k \cup \{c\}$;
20)     **end**;
21) **end**;
22) **return** $\bigcup_k \mathcal{P}_k$.

1 and 2). In step 3, frequent 1-patterns are determined. They are marked as key patterns unless their support is 1 (steps 4–6). The main loop is similar to the one in Apriori (steps 7 to 21). First, PASCAL-GEN is called to compute the candidate patterns. The support of key ones is determined via a database pass (steps 10–14). The 'subset' function (step 11) is the same as in Apriori: it returns all the candidate patterns included in the object $o$. Candidate patterns are stored in a data structure allowing fast retrieval, like the hash-tree in Apriori [2]. In our implementation, we used a trie with hash nodes [18].

Then (steps 15–20) the 'traditional' pruning is done. At the same time, for all remaining candidate key patterns, it is determined whether they are key or not (step 17 and 18).

The way that PASCAL-GEN operates is basically known from the generator function Apriori-Gen which was introduced in [2]. When called at the $k$th iteration, it uses as input the set of frequent $(k-1)$-patterns $\mathcal{P}_{k-1}$. Its output is the set of candidate $k$-patterns. Additionally to Apriori-Gen's *join* and *prune* steps, PASCAL-GEN makes the new candidates inherit the fact of being or not a candidate key pattern (step 9) by using Theorem 2; and it determines at the same time the support of all non-key candidate patterns (step 12) by using Theorem 4.

**Algorithm 2** PASCAL-GEN

Input: $\mathcal{P}_{k-1}$, the set of frequent $(k-1)$-patterns $p$ with their support $p.\sup$ and the $p$.key flag.

Output: $\mathcal{C}_k$, the set of candidate $k$-patterns $c$ each with the flag $c$.key, the value $c$.pred_sup, and the support $c.\sup$ if $c$ is not a key pattern.

1) **insert into** $\mathcal{C}_k$
   **select** $p.\text{item}_1, p.\text{item}_2, \ldots, p.\text{item}_{k-1}, q.\text{item}_{k-1}$
   **from** $\mathcal{P}_{k-1}\ p, \mathcal{P}_{k-1}\ q$
   **where** $p.\text{item}_1 = q.\text{item}_1, \ldots,$
   $p.\text{item}_{k-2} = q.\text{item}_{k-2}, p.\text{item}_{k-1} < q.\text{item}_{k-1}$;
2) **forall** $c \in \mathcal{C}_k$ **do begin**
3)   $c$.key $\leftarrow$ true; $c$.pred_sup $\leftarrow +\infty$;
4)   **forall** $(k-1)$-subsets $s$ of $c$ **do begin**
5)     **if** $s \notin \mathcal{P}_{k-1}$ **then**
6)       **delete** $c$ from $\mathcal{C}_k$;
7)     **else begin**
8)       $c$.pred_sup $\leftarrow \min(c.\text{pred\_sup}, s.\sup)$;
9)       **if** not $s$.key **then** $c$.key $\leftarrow$ false;
10)     **end**;
11)   **end**;
12)   **if** not $c$.key **then** $c.\sup \leftarrow c$.pred_sup;
13) **end**;
14) **return** $\mathcal{C}_k$.

*Running example.* We illustrate the PASCAL algorithm on the following dataset for *minsup* $= 2/5$:

| ID | Items | | | | |
|---|---|---|---|---|---|
| 1 | $A$ | $C$ | $D$ | $F$ | |
| 2 | $B$ | $C$ | $E$ | $F$ | |
| 3 | $A$ | $B$ | $C$ | $E$ | $F$ |
| 4 | $B$ | $E$ | $F$ | | |
| 5 | $A$ | $B$ | $C$ | $E$ | $F$ |

The algorithm performs first one database pass to count the support of the 1-patterns. The candidate pattern $\{D\}$ is pruned because it is infrequent. As $\{F\}$ has the same support as the empty set, $\{F\}$ is marked as a non-key pattern:

| $\mathcal{P}_1$ | sup | key |
|---|---|---|
| $\{A\}$ | 3/5 | t |
| $\{B\}$ | 4/5 | t |
| $\{C\}$ | 4/5 | t |
| $\{E\}$ | 4/5 | t |
| $\{F\}$ | 1 | f |

At the next iteration, all candidate 2-patterns are created and stored in $\mathcal{C}_2$. At the same time, the support of all patterns containing $\{F\}$ as a sub-pattern is computed. Then a database pass is performed to determine the supports of the remaining six candidate patterns:

| $\mathcal{C}_2$ | pred_sup | key | sup |
|---|---|---|---|
| $\{AB\}$ | 3/5 | t | ? |
| $\{AC\}$ | 3/5 | t | ? |
| $\{AE\}$ | 4/5 | t | ? |
| $\{AF\}$ | 3/5 | f | 3/5 |
| $\{BC\}$ | 4/5 | t | ? |
| $\{BE\}$ | 4/5 | t | ? |
| $\{BF\}$ | 4/5 | f | 4/5 |
| $\{CE\}$ | 4/5 | t | ? |
| $\{CF\}$ | 4/5 | f | 4/5 |
| $\{EF\}$ | 4/5 | f | 4/5 |

| $\mathcal{P}_2$ | sup | key |
|---|---|---|
| $\{AB\}$ | 2/5 | t |
| $\{AC\}$ | 3/5 | f |
| $\{AE\}$ | 2/5 | t |
| $\{AF\}$ | 3/5 | f |
| $\{BC\}$ | 3/5 | t |
| $\{BE\}$ | 4/5 | f |
| $\{BF\}$ | 4/5 | f |
| $\{CE\}$ | 3/5 | t |
| $\{CF\}$ | 4/5 | f |
| $\{EF\}$ | 4/5 | f |

At the third iteration, it turns out in Pascal-Gen that each newly generated candidate pattern contains at least one sub-pattern which is not a key pattern. Hence all new candidate patterns are not candidate key pattern, and all their supports are determined directly in Pascal-Gen. *From there on, the database will not be accessed any more.*

| $\mathcal{C}_3$ | pred_sup | key | sup |
|---|---|---|---|
| $\{ABF\}$ | 2/5 | f | 2/5 |
| $\{ABC\}$ | 2/5 | f | 2/5 |
| $\{ABE\}$ | 2/5 | f | 2/5 |
| $\{ACE\}$ | 2/5 | f | 3/5 |
| $\{ACF\}$ | 3/5 | f | 3/5 |
| $\{AEF\}$ | 2/5 | f | 2/5 |
| $\{BCE\}$ | 3/5 | f | 3/5 |
| $\{BCF\}$ | 3/5 | f | 3/5 |
| $\{BEF\}$ | 4/5 | f | 4/5 |
| $\{CEF\}$ | 3/5 | f | 3/5 |

| $\mathcal{P}_3$ | sup | key |
|---|---|---|
| $\{ABF\}$ | 2/5 | f |
| $\{ABC\}$ | 2/5 | f |
| $\{ABE\}$ | 2/5 | f |
| $\{ACE\}$ | 2/5 | f |
| $\{ACF\}$ | 3/5 | f |
| $\{AEF\}$ | 2/5 | f |
| $\{BCE\}$ | 3/5 | f |
| $\{BCF\}$ | 3/5 | f |
| $\{BEF\}$ | 4/5 | f |
| $\{CEF\}$ | 3/5 | f |

In the fourth and fifth iteration, all supports are determined directly in Pascal-Gen. In the sixth iteration, Pascal-Gen generates no new candidate patterns, thus no frequent 6-patterns are computed and the algorithm stops:

| $\mathcal{C}_4$ | pred_sup | key | sup |
|---|---|---|---|
| $\{ABCE\}$ | 2/5 | f | 2/5 |
| $\{ABCF\}$ | 2/5 | f | 2/5 |
| $\{ABEF\}$ | 2/5 | f | 2/5 |
| $\{ACEF\}$ | 2/5 | f | 3/5 |
| $\{BCEF\}$ | 3/5 | f | 3/5 |

| $\mathcal{P}_4$ | sup | key |
|---|---|---|
| $\{ABCE\}$ | 2/5 | f |
| $\{ABCF\}$ | 2/5 | f |
| $\{ABEF\}$ | 2/5 | f |
| $\{ACEF\}$ | 2/5 | f |
| $\{BCEF\}$ | 3/5 | f |

| $\mathcal{C}_5$ | pred_sup | key | sup |
|---|---|---|---|
| $\{ABCEF\}$ | 2/5 | f | 2/5 |

| $\mathcal{P}_5$ | sup | key |
|---|---|---|
| $\{ABCEF\}$ | 2/5 | f |

Hence Pascal needs two database passes in which the algorithm counts the supports of $6 + 6 = 12$ patterns. Apriori would have needed five database passes for counting the supports of $6 + 10 + 10 + 5 + 1 = 32$ patterns for the same dataset. All other current algorithms (with the only exception of Close) may need less than five passes, but they all have to perform the 32 counts.

## 5. EXPERIMENTAL EVALUATION

We evaluated Pascal against the algorithms Apriori, Close, and Max-Miner. Max-Miner was extended to retrieve the frequent patterns with their support by a pass over the databases; the two phases are shown in the tables below[8]. Pascal, Apriori, Close and this final step to Max-Miner all shared the same data structures and general organization. Optimizations such as special handling of pass two or items reordering were disabled. Experiments were conducted on a PC Pentium 3 600MHz with 512MiB of RAM.

Characteristics of the datasets used are given in Table 2. These datasets are the C20D10K and C73D10K census datasets from the PUMS sample file[9], the T25I10D10K and T25I20D100K[10] synthetic dataset that mimics market basket data, and the Mushrooms[11] dataset describing mushrooms characteristics [5]. In all experiments, we attempted to choose significant minimum support threshold values.

| Name | # of objects | Avg. size | # of items |
|---|---|---|---|
| T20I6D100K | 100,000 | 20 | 1,000 |
| T25I10D10K | 10,000 | 25 | 1,000 |
| T25I20D100K | 100,000 | 25 | 10,000 |
| C20D10K | 10,000 | 20 | 386 |
| C73D10K | 10,000 | 73 | 2,178 |
| Mushrooms | 8,416 | 23 | 128 |

Table 2: Datasets

Related work have shown that the behavior of algorithms for extracting frequent patterns depends mainly on the dataset characteristics. Weakly correlated data, such as synthetic data, constitute easy cases for the extraction since few patterns are frequent. For such data, all algorithms give acceptable response times as we can observe in Section 5.1 in which experimental results obtained for the T20I6D100K, T25I10D10K and T25I20D100K datasets are presented. On the contrary, correlated data constitute far more difficult cases for the extraction due to the important proportion of patterns that are frequent among all patterns. Such data represent a huge part of real-life datasets, and differences between extraction times obtained widely vary depending on the algorithm used. Experimental results obtained for the C20D10K, C73D10K and Mushrooms datasets, that are made up of correlated data, are given in Section 5.2.

### 5.1 Weakly correlated data

The T20I6D100K, T25I10D10K and T25I20D100K synthetic datasets are constructed according to the properties of market basket data that are typical weakly correlated data. In these datasets, the number of frequent patterns is small compared to the total number of patterns and, in most cases, nearly all the frequent patterns are also key patterns.

Response times for the T20I6D100K dataset are presented numerically in Table 3 and graphically in Figure 2. In this dataset, all frequents patterns are key patterns and Apriori and Pascal behave identically and response times obtained from them and Max-Miner are similar. The Close algorithm gives higher response times due to the number of intersection operations needed for computing the closures of the

[8] This second phase is by far the most costly part of the 'Max-Miner+' run-times.
[9] ftp://ftp2.cc.ukans.edu/pub/ippbr/census/pums/pums90ks.zip
[10] http://www.almaden.ibm.com/cs/quest/syndata.html
[11] ftp://ftp.ics.uci.edu/pub/machine-learning-databases/mushroom/agaricus-lepiota.data

candidate patterns.

| Sup. | # freq. | Pascal | Apriori | Close | Max-Miner$^+$ | |
|---|---|---|---|---|---|---|
| 1.00 | 1,534 | 13.14 | 13.51 | 25.91 | 2.60 | 5.03 |
| 0.75 | 4,710 | 20.41 | 20.67 | 35.29 | 4.44 | 11.06 |
| 0.50 | 26,950 | 44.00 | 44.38 | 67.82 | 6.87 | 35.37 |
| 0.25 | 155,673 | 117.97 | 117.79 | 182.95 | 15.64 | 109.14 |

Table 3: Response times for T20I6D100K

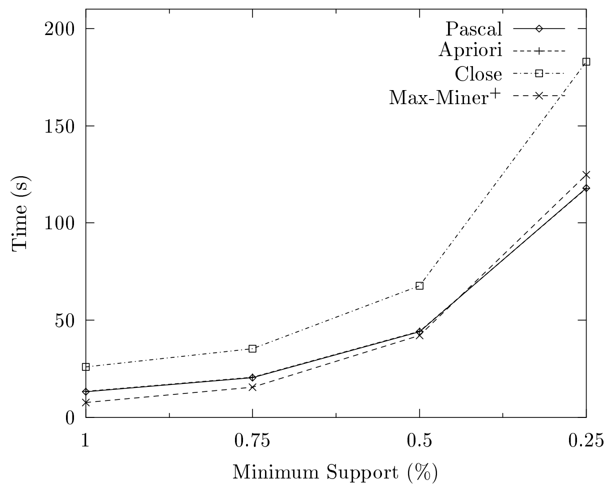


Figure 2: Experimental results for T20I6D100K

| Sup. | # freq. | Pascal | Apriori | Close | Max-Miner$^+$ | |
|---|---|---|---|---|---|---|
| 1.00 | 583 | 5.15 | 5.76 | 11.15 | 1.24 | 1.3 |
| 0.75 | 1,155 | 9.73 | 11.13 | 35.67 | 1.99 | 1.77 |
| 0.50 | 1,279,254 | 968.64 | 935.14 | 2,151.34 | 24.94 | 879.85 |

Table 4: Response times for T25I20D100K

Results for the T25I20D100K dataset are presented numerically in Table 4 and graphically in Figure 3. For this dataset, nearly all frequent patterns are key patterns, and results are similar to those obtained for the T20I6D100K dataset: Pascal and Apriori give identical response times and suffer a slight performance loss over Max-Miner while Close is the worst performer.

In Table 5 and Figure 4, execution times for the T25I10D10K dataset are presented. In this dataset, the proportion of frequent patterns that are not key patterns is much more important than for the T25I20D100K dataset. For the 1.00 and 0.75 minsup thresholds, Max-Miner performs better than Apriori and Pascal that themselves perform better than Close. For the lower 0.50 and 0.25 minsup thresholds, Pascal becomes the best performer and is slightly better than Close whereas they both clearly outperform Apriori and Max-Miner: When the proportion of frequent patterns that are not key is significant, the mechanism used by Pascal (resp. Close) to consider only key (resp. closed) patterns enables to reduce considerably the number of support counts performed.

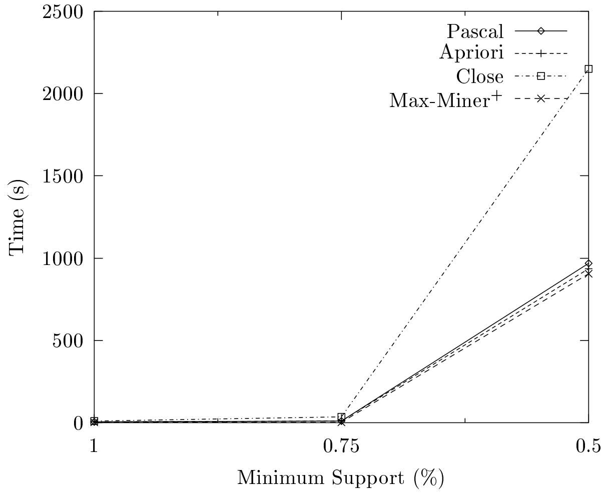


Figure 3: Experimental results for T25I20D10K

| Sup. | # freq. | Pascal | Apriori | Close | Max-Miner$^+$ | |
|---|---|---|---|---|---|---|
| 1.00 | 3,300 | 3.24 | 3.62 | 6.67 | 0.63 | 1.05 |
| 0.75 | 17,583 | 5.17 | 6.95 | 9.38 | 1.09 | 3.83 |
| 0.50 | 331,280 | 17.82 | 41.06 | 26.43 | 2.76 | 35.53 |
| 0.25 | 2,270,573 | 70.37 | 187.92 | 86.08 | 6.99 | 154.89 |

Table 5: Response times for T25I10D10K

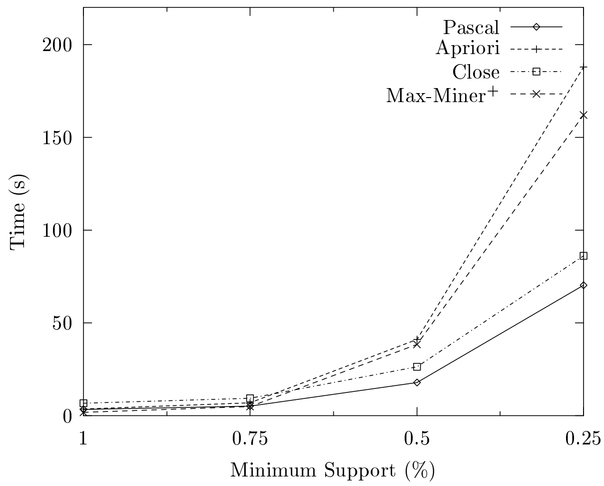


Figure 4: Experimental results for T25I10D10K

## 5.2 Correlated data

Response times obtained from the C20D10K and C73D10K census datasets are given numerically in Tables 6 and 7, and graphically in Figures 5 and 6. Results for the MUSHROOMS dataset are presented in Table 8 and Figure 7. In these three datasets, constituted of correlated data, the proportion of patterns that are frequent is important but few of them are also key patterns. Hence, using pattern counting inference, PASCAL has to perform much fewer support counts than the Apriori and the Max-Miner algorithms. The same observation stands for the Close algorithm, that uses the closure mechanism to reduce the number of support counts, and both PASCAL and Close are an order of magnitude faster than Apriori and Max-Miner. Differences between the execution times of PASCAL and Close and those of Apriori and Max-Miner can be counted in tens of minutes for C20D10K and MUSHROOMS and in hours for C73D10K. Moreover, pattern counting inference and closure mechanism allow to reduce the number of passes on the datasets since the supports of all candidate patterns of some iteration are all deduced from the supports of key, or closed, patterns of previous iterations. On C73D10K with *minsup* = 60%, for instance, PASCAL and Close both make 13 passes while the largest frequent patterns are of size 19. For this dataset and this threshold value, frequent patterns could not be derived from the maximal frequent patterns extracted with Max-Miner since we did not implement memory management for this phase and it required in this case more memory space than available.

| Sup. | # freq. | Pascal | Apriori | Close | Max-Miner$^+$ | |
|---|---|---|---|---|---|---|
| 20.0 | 20,239 | 9.44 | 57.15 | 14.36 | 0.17 | 77.40 |
| 15.0 | 36,359 | 12.31 | 85.35 | 18.99 | 0.26 | 113.22 |
| 10.0 | 89,883 | 19.29 | 164.81 | 29.58 | 0.34 | 201.33 |
| 7.5 | 153,163 | 23.53 | 232.40 | 36.02 | 0.35 | 268.80 |
| 5.0 | 352,611 | 33.06 | 395.32 | 50.46 | 0.48 | 428.65 |
| 2.5 | 1,160,363 | 55.33 | 754.64 | 78.63 | 0.81 | 775.56 |

Table 6: Response times for C20D10K

| Sup. | # freq. | Pascal | Apriori | Close | Max-Miner$^+$ | |
|---|---|---|---|---|---|---|
| 80 | 109,159 | 177.49 | 3,661.27 | 241.91 | 0.87 | 3,717.99 |
| 75 | 235,271 | 392.80 | 7,653.58 | 549.27 | 1.06 | 7,730.36 |
| 70 | 572,087 | 786.49 | 17,465.10 | 1,112.42 | 2.28 | 17,618.40 |
| 60 | 4,355,543 | 3,972.10 | 109,204.00 | 5,604.91 | 7.72 | (*) |

(*) Not enough memory.

Table 7: Response times for C73D10K

| Sup. | # freq. | Pascal | Apriori | Close | Max-Miner$^+$ | |
|---|---|---|---|---|---|---|
| 20.0 | 53,337 | 6.48 | 115.82 | 9.63 | 0.31 | 134.31 |
| 15.0 | 99,079 | 9.81 | 190.94 | 14.57 | 0.50 | 218.93 |
| 10.0 | 600,817 | 23.12 | 724.35 | 29.83 | 0.89 | 745.72 |
| 7.5 | 936,247 | 32.08 | 1,023.24 | 41.05 | 1.25 | 1,035.48 |
| 5.0 | 4,140,453 | 97.12 | 2,763.42 | 98.81 | 1.99 | 2,752.05 |

Table 8: Response times for MUSHROOMS

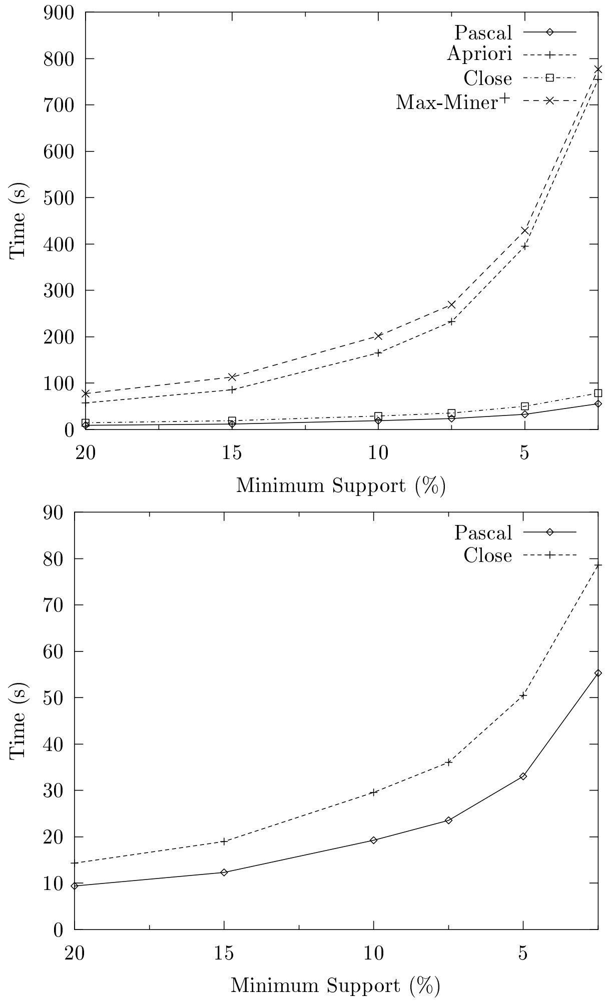


Figure 5: Experimental results for C20D10K

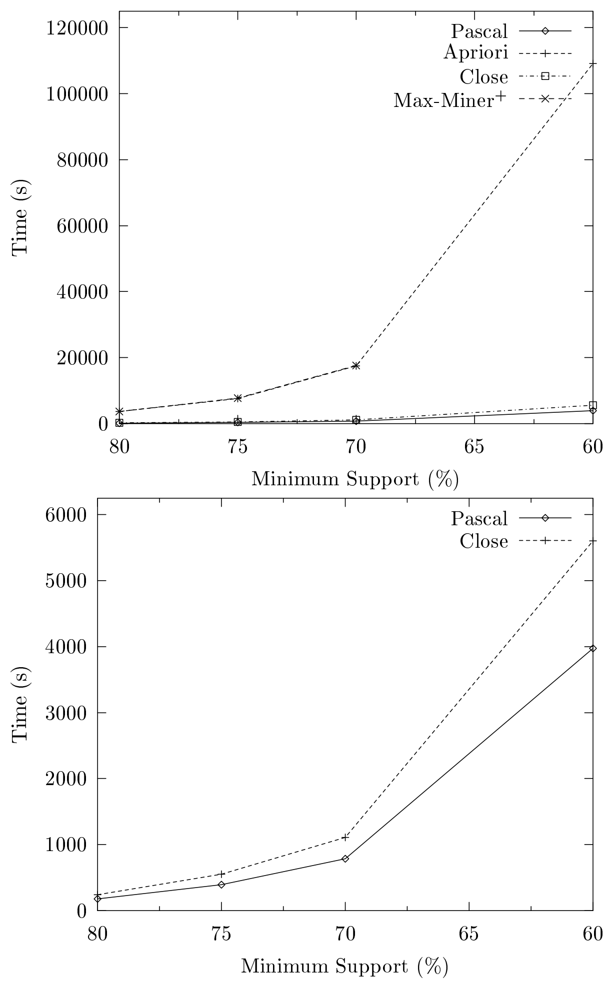


Figure 6: Experimental results for C73D10K

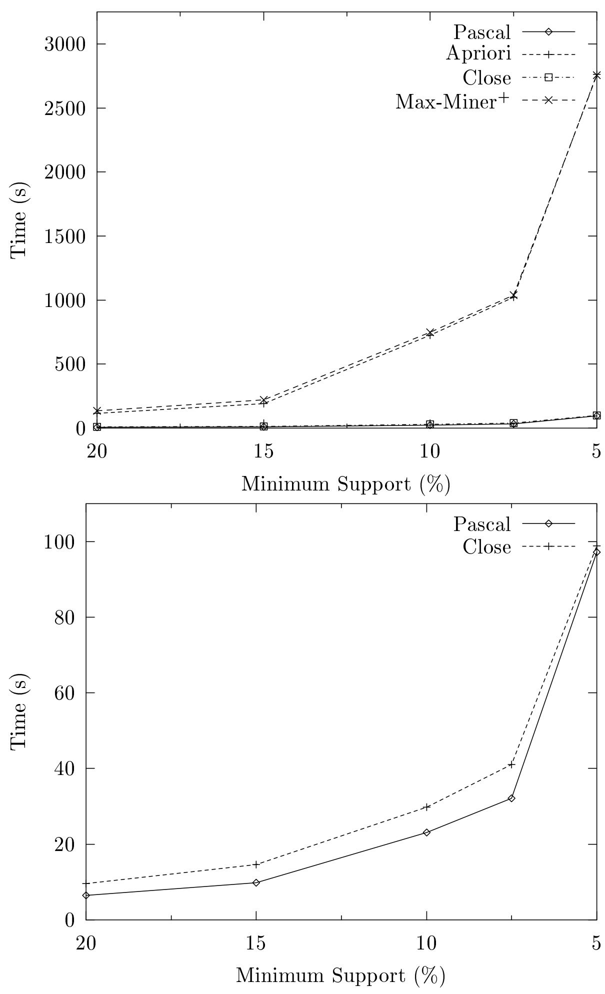


Figure 7: Experimental results for Mushrooms

## 6. CONCLUSION

We presented PASCAL, a novel optimization of the Apriori algorithm for fast discovery of frequent patterns. PASCAL is both effective and easy to implement or to integrate in existing implementations based on the Apriori approach. This optimization uses pattern counting inference, using the key patterns in equivalence classes to reduce the number of patterns counted and database passes.

We conducted performance evaluations to compare the efficiency of PASCAL with those of Apriori, Max-Miner and Close. The results showed that PASCAL gives response times equivalent to those of Apriori and Max-Miner when extracting all frequent patterns and their support from weakly correlated data, and that it is the most efficient among the four algorithms when data are dense or correlated.

Frequent key patterns are also used for simplifying rule generation, as they can be seen as the left hand sides of minimal non-redundant association rules [4].

## 7. ACKNOWLEDGEMENTS

We would like to thank Roberto Bayardo, who provided us with the implementation of Max-Miner used in the tests and commented on an earlier version of this work.

## 8. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 207–216, May 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. of the 20th Int'l Conf. on Very Large Data Bases (VLDB)*, pages 478–499, June 1994.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of the 11th Int'l Conf. on Data Engineering (ICDE)*, pages 3–14, Mar. 1995.

[4] Y. Bastide, N. Pasquier, R. Taouil, G. Stumme, and L. Lakhal. Mining minimal non-redundant rules using frequent closed itemsets. In *Proc. of the 1st Int'l Conf. on Computational Logic (6th Int'l Conf. on Database Systems – DOOD)*, pages 972–986. Springer, July 2000.

[5] S. D. Bay. The UCI KDD Archive [`http://kdd.ics.uci.edu`]. Irvine, CA: University of California, Department of Information and Computer Science.

[6] R. Bayardo and R. Agrawal. Mining the most interesting rules. In *Proc. of the 5th Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 145–154, Aug. 1999.

[7] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 85–93, June 1998.

[8] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: Generalizing association rules to correlation. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 265–276, May 1997.

[9] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 255–264, May 1997.

[10] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.

[11] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, Sept. 2000.

[12] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 1–12, May 2000.

[13] M. Kamber, J. Han, and Y. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proc. of the 3rd KDD Int'l Conf.*, Aug. 1997.

[14] B. Lent, A. Swami, and J. Widom. Clustering association rules. In *Proc. of the 13th Int'l Conf. on Data Engineering (ICDE)*, pages 220–231, Mar. 1997.

[15] D. Lin and Z. M. Kedem. Pincer-Search: A new algorithm for discovering the maximum frequent set. In *Proc. of the 6th Int'l Conf.on Extending Database Technology (EDBT)*, pages 105–119, Mar. 1998.

[16] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, Sept. 1997.

[17] J. S. Park, M. S. Chen, and P. S. Yu. An efficient hash based algorithm for mining association rules. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 175–186, May 1995.

[18] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Pruning closed itemset lattices for association rules. In *Actes des 14$^e$ journées « Bases de données avancées »*, pages 177–196, Oct. 1998.

[19] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. of the 7th Int'l Conf. on Database Theory (ICDT)*, pages 398–416, Jan. 1999.

[20] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Efficient mining of association rules using closed itemset lattices. *Journal of Information Systems*, 24(1):25–46, Mar. 1999.

[21] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *Proc. Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD)*, pages 21–30, May 2000.

[22] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the 21th Int'l Conf. on Very Large Data Bases (VLDB)*, pages 432–444, Sept. 1995.

[23] C. Silverstein, S. Brin, and R. Motwani. Beyond market baskets: Generalizing association rules to dependence rules. *Data Mining and Knowledge Discovery*, 2(1), Jan. 1998.

[24] R. Taouil, N. Pasquier, Y. Bastide, and L. Lakhal. Mining basis for association rules using closed sets. In *Proc. of the 16th Int'l Conf. on Data Engineering (ICDE)*, page 307, Feb.–Mar. 2000.

[25] H. Toivonen. Sampling large databases for association rules. In *Proc. of the 22nd Int'l Conf. on Very Large Data Bases (VLDB)*, pages 134–145, Sept. 1996.

[26] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proc. of the 3rd Int'l Conf. on Knowledge Discovery in Databases (KDD)*, pages 283–286, Aug. 1997.