METHODS FOR BATCH PROCESSING OF DATA MINING QUERIES

Marek Wojciechowski and Maciej Zakrzewicz
Institute of Computing Science
Poznan University of Technology
ul. Piotrowo 3a
Poznan, Poland

Abstract: Data mining is a useful decision support technique, which can be used to find trends and regularities in warehouses of corporate data. A serious problem of its practical applications is long processing time required by data mining algorithms. Current systems consume minutes or hours to answer single requests, while typically batches of the requests are delivered the systems. In this paper we present the problem of batch processing of data mining requests. We introduce methods that analyze similarities between separate requests to reduce the processing cost. We also perform a comparative performance analysis of the proposed methods.

Key words: data mining, multiple query optimization

1. INTRODUCTION

Data mining, also referred to as database mining or knowledge discovery in databases (KDD), aims at discovery of useful patterns from large databases or warehouses [1][2][4][6][10][11][12]. Currently we are observing the evolution of data mining environments from specialized tools to multi-purpose data mining systems offering some level of integration with existing database management systems. From a user's point of view data mining can be seen as advanced querying: a user specifies the source data set and the requested class of patterns, the system chooses the right data mining algorithm and returns discovered patterns to the user [3][5][7][8][9]. The most serious problem concerning data mining queries is a long response time. Current systems consume minutes or hours to answer single queries.

Data mining applications typically execute data mining queries during nights, when system activity is low. Sets of data mining queries are scheduled and then automatically evaluated by a data mining system. It is possible that the data mining queries delivered to the system are somehow similar, eg. their source data sets overlap. Unfortunately, none of the proposed data mining algorithms tried to employ such similarity of data mining requests to reduce their processing cost.

In this paper we present the problem of batch processing of data mining queries. We describe and analyze three methods of executing batches of data mining queries in a more efficient way. We illustrate our methods with many examples expressed in *MineSQL*, which is a declarative, multi-purpose *SQL*-like language for interactive and iterative data mining in relational databases, developed by us over the last couple of years [8][9].

1.1 Basic Definitions

Frequent itemsets. Let $L=\{l_1, l_2, ..., l_m\}$ be a set of literals, called items. Let a non-empty set of items T be called an *itemset*. Let D be a set of variable length itemsets, where each itemset $T \subseteq L$. We say that an itemset T *supports* an item $x \in L$ if x is in T. We say that an itemset T *supports* an itemset $X \subseteq L$ if T supports every item in the set X. The *support* of the itemset X is the percentage of T in D that support X. The problem of mining frequent itemsets in D consists in discovering all itemsets whose support is above a user-defined support threshold.

Apriori algorithm. Apriori is an example of a level-wise algorithm for association discovery. It makes multiple passes over the input data to determine all frequent itemsets. Let L_k denote the set of frequent itemsets of size k and let C_k denote the set of candidate itemsets of size k. Before making the k-th pass, Apriori generates C_k using L_{k-1}. Its candidate generation process ensures that all subsets of size $k-1$ of C_k are all members of the set L_{k-1}. In the k-th pass, it then counts the support for all the itemsets in C_k. At the end of the pass all itemsets in C_k with a support greater than or equal to the minimum support form the set of frequent itemsets L_k. Figure 1 provides the pseudocode for the general level-wise algorithm, and its Apriori implementation. The *subset(t, k)* function gives all the subsets of size k in the set t.

This method of pruning the C_k set using L_{k-1} results in a much more efficient support counting phase for Apriori when compared to the earlier algorithms. In addition, the usage of a hash-tree data structure for storing the candidates provides a very efficient support-counting process.

```
$C_1$ = {all 1-itemsets from $D$}
for ($k$=1; $C_k \neq \varnothing$; $k$++)
        count($C_k$, $D$);
    $L_k = \{c \in C_k \mid c.count \geq minsup\}$;
    $C_{k+1}$ = generate_candidates($L_k$);
$Answer = \bigcup_k L_k$;
```

```
$L_1$ = {frequent 1-itemsets}
for ($k$ = 2; $L_{k-1} \neq \varnothing$; $k$++)
    $C_k$ = generate_candidates($L_{k-1}$);
    forall tuples $t \in D$
            $C_t$=$C_k \cap$ subset($t$, $k$);
            forall candidates $c \in C_t$
                $c.count$++;
    $L_k = \{c \in C_k \mid c.count \geq minsup\}$
$Answer = \bigcup_k L_k$;
```

Figure -1. Level-wise algorithm for association discovery and its Apriori implementation

1.2 MineSQL Data Mining Query Language

MineSQL is a SQL language extension we presented in [9] as a tool to formulate data mining queries. The main MineSQL statement is *MINE*, designed to discover frequent patterns from a result of a *SELECT* query. The discovered patterns may be filtered by means of user-defined conditions. We also introduced new datatypes to allow to store itemsets in database relations: *SET OF CHAR*, *SET OF INTEGER*, etc., the *SET()* grouping function, as well as the *CONTAINS* operator used to determine if one set of items contains another set of items. In [8][13] we extended MineSQL with data mining materialized views and sequential pattern processing operators.

The following example statements illustrate MineSQL capabilities to create a database relation to hold sets of integers and to discover all frequent itemsets with support greater than 10 in the first 100 tuples of the relation.

```
create table mysets                mine itemset
(i integer,                        from (select s
 s set of integer)                       from   mysets
                                         where  i<=100)
                                   where support(itemset) > 10
```

The next example illustrates MineSQL capabilities to store results of a data mining query:

```
create table mypatterns            insert into mypatterns
(s set of integer)                 mine itemset
                                   from (select s
                                         from   mysets
                                         where  i<=100)
                                   where support(itemset) > 10
```

2. PRELIMINARIES AND PROBLEM STATEMENT

Data mining query. A data mining query is a tuple $DMQ = (R, a, \Sigma, \Phi)$, where R is a relation, a is an attribute of R, Σ is a condition involving the attributes of the relation R, Φ is a condition involving discovered patterns.

The result of the data mining query is a set of patterns discovered in $\pi_a\sigma_\Sigma$ and satisfying Φ.

Example. Given the relation R_1 shown in Fig. 1a, the result of the data mining query DMQ_1=(R_1, *"iset", "id>5 AND id<10", "minsup ≥ 3"*) is shown in Fig. 1b.

```
mine itemset from (select items from R1 where id>5 and id<10)
where support(itemset)>=3;
```

```
R1:     id  iset
        --------
        1   a,b,c
        4   a,c                           result of DMQ1:
        6   d,f,g
        7   f,g,k,m                            {f}
        8   e,f,g                              {g}
        15  a,f                               {f,g}
```

Figure -2a Example relation R_1 *Figure -2b* DMQ_1 query result

Problem statement. Given a set $S = \{DMQ_1, DMQ_2, ..., DMQ_n\}$ of data mining queries, where $DMQ_i = (R_i, a_i, \Sigma_i, \Phi_i)$ and $\forall_i \exists_{j\neq i}\, \sigma_{\Sigma i} (R_i) \cap \sigma_{\Sigma j} (R_j) \neq \emptyset$, the goal is to minimize the I/O cost and the CPU cost of executing S.

2.1 Motivating example

Consider a relation *Sales(uad, basket, time)* to store purchases made by users of an internet shop. Since data sets of this kind tend to be very large, there is a need for automated analysis of their contents. Assume a shop manager is interested in finding sets of products that were frequently co-occurring in the users' purchases. The shop manager plans to create two reports: one showing the frequent sets that appeared in more than 350 purchases in Jan 2002 and one showing the frequent sets that appeared in more than 20 purchases made by clients from France. Two required data mining queries are shown below.

```
DMQA                                       DMQB
mine itemset                               mine itemset
from (select basket from   sales           from (select basket
      where  time between '01-01-02'             from   sales
                      and '01-31-02')            where  uad like '%.fr')
where support(itemset) > 350               where support(itemset) > 20
```

If the size of the *Sales* relation is very large, each of the above data mining queries can take a significant amount of time to execute. Part of this time will be spent on reading the *Sales* relation from disk in order to count occurrences of candidate itemsets. Notice that the sets of bloks to be read by the two data mining queries may overlap. If we try to merge the processing of the two data mining queries, we can reduce redundancy resulting from this overlapping. In the remaining of this paper we will use this example to illustrate particular methods.

3. MODEL OF A LEVEL-WISE ASSOCIATION DISCOVERY ALGORITHM

In order to describe methods for batch processing of data mining queries, we first need to introduce a notation to express steps of a level-wise association discovery algorithm. We decided to use the extended relational algebra to model the level-wise algorithm processing in the following way. Each candidate counting step is represented as a relational join, followed by grouping and selection operations. Figure 3 shows the SQL query and the relational algebra graph for the candidate counting step; *C(s)* is the candidates relation, *R(s)* is the database relation. The candidate generation step is represented as a simple relational join. Figure 4 shows the SQL query and the relational algebra graph for this case.

```
select   c.s, count(r.s)
from     c, r
where    r.s contains c.s
group by c.s
having   count(r.s)>=minsup
```

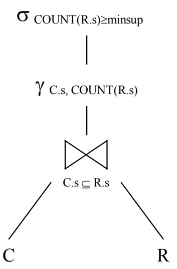


Figure -3. Candidate counting-pruning step modeled with relational algebra

```
select   union(l1.s, l2.s) as cand
from     l l1, l l2
where   size(difference(l1.s,l2.s)) = 1
group by cand
having   count(*) = k*(k-1)/2
```

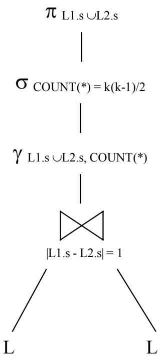


Figure -4. Candidate generation step for C_k modeled with relational algebra

In order to analyze the general cost model of the level-wise association discovery algorithm, we make the following assumptions: (1) the size of the database is much larger than the size of all candidate itemsets, (2) the size of all candidate itemsets is larger than the memory size, and (3) frequent itemsets fit in memory. The notation we use is given in Table 1.

Table 1. Notation used in cost models

M	main memory size (blocks)
$\|D\|$	number of itemsets in the database
$\|\|D\|\|$	size of the database (blocks)
$\|C_i\|$	number of candidate itemsets for step *I*
$\|\|C_i\|\|$	size of all candidate itemsets for step *i* (blocks), $\|\|C_i\|\|<<\|\|D\|\|$, $\|\|C_i\|\|<M$
$\|L_i\|$	number of frequent itemsets for step *i*, $\|L_i\|<\|C_i\|$
$\|\|L_i\|\|$	size of all frequent itemsets for step *i* (blocks), $\|\|L_i\|\|<M$

The cost of performing the general level-wise association discovery algorithm is as follows:

1. **Candidate counting-pruning.** Candidate itemsets must be read from disk in portions equal to the available memory size. For each portion, the database must be scanned to join itemsets from C_i with itemsets from *D*. Next, the candidate itemsets with support greater or equal to *minsup* become frequent itemsets and must be written to disk. The I/O cost of a single iteration *i* is the following:

$$\text{cost}_{I/O} = \|C_i\| + \frac{\|C_i\|}{M}\|D\| + \|L_i\|$$

The dominant part of the CPU cost is join condition verification. For the simplicity, we assume the cost of comparing two itemsets does not depend on their sizes and equals 1. Thus, the CPU cost of a single iteration *i* is the following:

$$\text{cost}_{CPU} = |C_i||D|$$

2. **Candidate generation.** Frequent itemsets from the previous iteration must be read from disk, joined in memory, and saved as new candidate itemsets. The I/O cost of a single iteration *i* is the following:

$$\text{cost}_{I/O} = \|L_i\| + \|C_{i+1}\|$$

The CPU cost of this phase of the algorithm is the following:

$$\text{cost}_{CPU} = |L_i||L_i|$$

Therefore, if *K* is the number of iterations, the overall cost of the level-wise algorithm is as follows:

$$\text{cost}_{I/O} = \sum_{i=1}^{K}\left(\|C_i\| + \frac{\|C_i\|}{M}\|D\| + 2\|L_i\| + \|C_{i+1}\|\right)$$

$$\text{cost}_{CPU} = \sum_{i=1}^{K}\left(|C_{i+1}||D| + |L_i|^2\right)$$

4. METHODS FOR BATCH PROCESSING OF DATA MINING QUERIES

In this Section we present three methods for processing batches of data mining queries. The first one represents a trivial approach, where we execute each DMQ separately. We call this method *Sequential Processing*. The second method, called *Common Counting*, integrates the counting phase of the level-wise algorithm to reduce I/O. The third method, called *Mine Merge*, splits DMQs into a new set of disjoint DMQs. Their results are used to answer the original queries.

4.1 Sequential Processing

In the *Sequential Processing* method, each DMQ is executed separately. We do not try to benefit from using common disk blocks by two separate data mining queries. Figure 5 gives the model and pseudocode for this method (C_i^A means C_i generated for DMQ^A, etc.). The cost of this method is equal to the sum of independent execution of each of the queries:

$$\text{cost}_{I/O} = \sum_{i=1}^{K^A}\left(\left\|C_i^A\right\| + \frac{\left\|C_i^A\right\|}{M}\left\|D^A\right\| + 2\left\|L_i^A\right\| + \left\|C_{i+1}^A\right\| \right) +$$

$$\sum_{i=1}^{K^B}\left(\left\|C_i^B\right\| + \frac{\left\|C_i^B\right\|}{M}\left\|D^B\right\| + 2\left\|L_i^B\right\| + \left\|C_{i+1}^B\right\| \right)$$

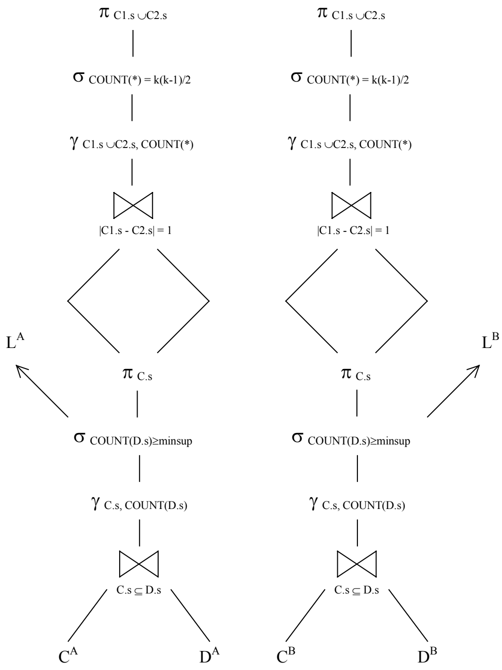
$$\text{cost}_{CPU} = \sum_{i=1}^{K^A}\left(\left|C_{i+1}^A\right|\left|D^A\right| + \left|L_i^A\right|^2 \right) + \sum_{i=1}^{K^B}\left(\left|C_{i+1}^B\right|\left|D^B\right| + \left|L_i^B\right|^2 \right)$$

4.2 Common Counting

When two or more different DMQs count their candidate itemsets in the same part of the database, the common part of their counting steps is integrated and requires only one scan of the involved part of the database. A model of a single step of the *Common Counting* algorithm and its procedural implementation are shown in Fig 6.

Example. Using the original database selection conditions, we construct three separate dataset definitions:

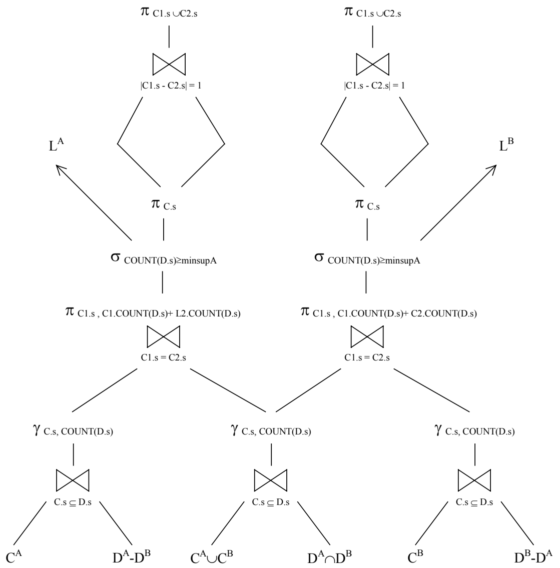
1.
   ```
   select basket from   sales
   where  time between ’01-01-02’ and ’01-31-02’
     and  NOT uad like ‘%.fr’
   ```
2.
   ```
   select basket from   sales
   where  time between ’01-01-02’ and ’01-31-02’
     and  uad like ‘%.fr’
   ```



```
$C_1^A$ = {all 1-itemsets from $D^A$}
for ($k$=1; $C_k^A \cup C_k^B \neq \emptyset$; $k$++)
        count($C_k^A$, $D^A$);
    $L_k^A = \{c \in C_k^A \mid c.count \geq minsup^A\}$;
    $C_{k+1}^A$ = generate_candidates($L_k^A$);
$Answer^A = \bigcup_k L_k^A$;

$C_1^B$ = {all 1-itemsets from $D^B$}
for ($k$=1; $C_k^B \neq \emptyset$; $k$++)
        count($C_k^B$, $D^B$);
    $L_k^B = \{c \in C_k^B \mid c.count \geq minsup^B\}$;
    $C_{k+1}^B$ = generate_candidates($L_k^B$);
$Answer^B = \bigcup_k L_k^B$;
```

Figure -5 Model of the Sequential Processing method



```
$C_1^A$ = {all 1-itemsets from $D^A$}
$C_1^B$ = {all 1-itemsets from $D^B$}
for ($k$=1; $C_k^A \cup C_k^A \neq \emptyset$; $k$++)
if $C_k^A \neq \emptyset$ count($C_k^A$, $D^A$ - $D^B$);
        if $C_k^B \neq \emptyset$ count($C_k^B$, $D^B$ - $D^A$);
        count($C_k^A \cup C_k^B$, $D^A \cap D^A$);
    $L_k^A = \{c \in C_k^A \mid c.count \geq minsup^A\}$;
    $L_k^B = \{c \in C_k^B \mid c.count \geq minsup^B\}$;
    $C_{k+1}^A$ = generate_candidates($L_k^A$);
    $C_{k+1}^B$ = generate_candidates($L_k^B$);
$Answer^A = \bigcup_k L_k^A$;
$Answer^B = \bigcup_k L_k^B$;
```

Figure -6. Model of the Common Counting method

3.
```
select basket
from   sales
where  NOT time between '01-01-02'
                    and '01-31-02'
  and  uad like '%.fr'
```

Next, we scan the first query's result in order to count *DMQ*^{A} candidate itemsets, then we scan the second query's result in order to count both *DMQ*^{A} and *DMQ*^{B} candidate itemsets, finally we scan the third query's result in order to count *DMQ*^{B} candidate itemsets. Notice that none of the database blocks nedeed to be read twice, if the candidate itemsets fit in memory.

Let us analyze the cost of this method. Candidate itemsets of *DMQ*^{A} must be read, joined with D^A-D^B, counted, and saved to disk. Also, candidate itemsets of *DMQ*^{B} must be read, joined with D^B-D^A, counted, and saved to disk. Next, all candidates of *DMQ*^{A} and *DMQ*^{B} must be read, joined with $D^A \cap D^B$, counted, and saved to disk. The candidate itemsets with support greater or equal to, respectively, *minsup*^{A} or *minsup*^{B}, become frequent itemsets and are written to disk. In order to generate new candidate itemsets, all frequent itemsets must be read from disk and new candidate itemsets must be written to disk. Therefore, the I/O cost of this method is the following:

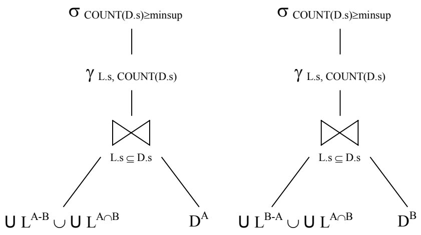
$$\text{cost}_{I/O} = \sum_{i=1}^{\max(K^A,K^B)} \left(\begin{gathered} 3\left\|C_i^A\right\| + \frac{\left\|C_i^A\right\|}{M}\left\|D^A - D^B\right\| + 3\left\|C_i^B\right\| + \frac{\left\|C_i^B\right\|}{M}\left\|D^B - D^A\right\| \\ + \frac{\left\|C_i^A\right\| + \left\|C_i^B\right\|}{M}\left\|D^A \cap D^B\right\| + 2\left\|L_i^A\right\| + 2\left\|L_i^B\right\| + \left\|C_{i+1}^A\right\| + \left\|C_{i+1}^B\right\| \end{gathered} \right)$$

Similarly, the CPU cost is as follows:

$$\text{cost}_{CPU} = \sum_{i=1}^{K^A}\left(\left|C_{i+1}^A\right|\left|D^A - D^B\right| + \left|L_i^A\right|^2\right) + \sum_{i=1}^{K^B}\left(\left|C_{i+1}^B\right|\left|D^B - D^A\right| + \left|L_i^B\right|^2\right) +$$
$$\sum_{i=1}^{\max(K^A,K^B)}\left(\left(\left|C_{i+1}^A\right| + \left|C_{i+1}^B\right|\right)\left|D^B \cap D^A\right|\right)$$

4.3 Mine Merge

This method employs the property that an itemset which is frequent in a whole data set, must also be frequent in at least one portion of it [4,13]. In the *Mine Merge* method, each pair of overlapping DMQs is divided into three separate DMQs. Next, the new DMQs are executed sequentially. The results of the new DMQs are candidates to determine the results of the original DMQs. Therefore, an additional counting step is needed to finally answer the original DMQs. The pseudocode of the method and a model of the additional step are given in Fig. 7.



```
$C_1^{A-B}$ = {all 1-itemsets from $D^A - D^B$ }
for ($k$=1; $C_k^{A-B} \neq \emptyset$; $k$++)
count($C_k^{A-B}$, $D^A - D^B$);
    $L_k^{A-B} = \{c \in C_k^{A-B} \mid c.count \geq minsup^A\}$;
    $C_{k+1}^{A-B}$ = generate_candidates($L_k^{\mathrm{A-B}}$);
$Answer^{A-B} = \bigcup_k L_k^{A-B}$;

$C_1^{B-A}$ = {all 1-itemsets from $D^B - D^A$ }
for ($k$=1; $C_k^{B-A} \neq \emptyset$; $k$++)
count($C_k^{B-A}$, $D^B - D^A$);
    $L_k^{B-A} = \{c \in C_k^{B-A} \mid c.count \geq minsup^B\}$;
    $C_{k+1}^{B-A}$ = generate_candidates($L_k^{B-A}$);
$Answer^{B-A} = \bigcup_k L_k^{B-A}$;

$C_1^{A \cap B}$ = {all 1-itemsets from $D^A \cap D^B$ }
for ($k$=1; $C_k^{A \cap B} \neq \emptyset$; $k$++)
count($C_k^{A \cap B}$, $D^A \cap D^B$);
 $L_k^{A \cap B}$ =
  $\{c \in C_k^{A \cap B} \mid c.count \geq \min(minsup^A , minsup^B)\}$;
 $C_{k+1}^{A \cap B}$ = generate_candidates($L_k^{A \cap B}$);
$Answer^{A \cap B} = \bigcup_k L_k^{A \cap B}$;

count($Answer^{A-B} \cup Answer^{A \cap B}$, $D^A$);
$Answer^A$ =
 $\{c \in Answer^{A-B} \cup Answer^{A \cap B} \mid c.count \geq minsup^A$);

count($Answer^{B-A} \cup Answer^{A \cap B}$, $D^B$);
$Answer^B$ =
 $\{c \in Answer^{B-A} \cup Answer^{A \cap B} \mid c.count \geq minsup^B$);
```

Figure –7. Model of the Mine Merge method

Example. Using the original database selection conditions, we construct three new data mining queries. Assume the intermediate results are written to the relation *Intermediate(label,itemset)*.

DMQ1:

```
insert into intermediate mine ‘DMQ1’, itemset
from ( select basket from sales
       where  time between ’01-01-02’ and ’01-31-02’
       and  NOT uad like ‘%.fr’)
where support(itemset)>350
```

DMQ2:

```
insert into intermediate mine ‘DMQ2’, itemset
from (select basket from sales
      where  time between ’01-01-02’ and ’01-31-02’
      and  uad like ‘%.fr’)
where support(itemset)>20
```

DMQ3:

```
insert into intermediate mine ‘DMQ3’, itemset
from (select basket from sales
      where  NOT time between ’01-01-02’ and ’01-31-02’
      and  uad like ‘%.fr’)
where support(itemset)>20
```

The above queries discover frequent itemsets in the three partitions of the original data sets. In the next step, we have to merge the partitions and verify the itemsets' final supports:

```
1. select itemset
   from   (select  distinct itemset from intermediate) i,
          sales s
   where  label in ('DMQ1','DMQ2')
   and    s.itemset contains i.itemset
   group by i.itemset
   having count(*)>350;
2.  select itemset
   from   (select  distinct itemset from intermediate) i,
          sales s
   where  label in ('DMQ2','DMQ3')
   and    s.itemset contains i.itemset
   group by i.itemset
   having count(*)>20;
```

The itemsets selected by the first Select query form the result of DMQ^A, and the itemsets selected by the second Select query form the result of DMQ^B.

Let us analyze the cost of this method. The I/O cost of executing the three new data mining queries is the following:

$$\text{cost}_{I/O} = \sum_{i=1}^{K^{A-B}} \left(\left\| C_i^{A-B} \right\| + \frac{\left\| C_i^{A-B} \right\|}{M} \left\| D^A - D^B \right\| + 2\left\| L_i^{A-B} \right\| + \left\| C_{i+1}^{A-B} \right\| \right) +$$

$$\sum_{i=1}^{K^{A\cap B}} \left(\left\| C_i^{A\cap B} \right\| + \frac{\left\| C_i^{A\cap B} \right\|}{M} \left\| D^A \cap D^B \right\| + 2\left\| L_i^{A\cap B} \right\| + \left\| C_{i+1}^{A\cap B} \right\| \right) +$$

$$\sum_{i=1}^{K^{B-A}} \left(\left\| C_i^{B-A} \right\| + \frac{\left\| C_i^{B-A} \right\|}{M} \left\| D^B - D^A \right\| + 2\left\| L_i^{B-A} \right\| + \left\| C_{i+1}^{B-A} \right\| \right)$$

The I/O cost of verifying the discovered itemsets' supports is the cost of performing the join operation:

$$\text{cost}_{I/O} = \frac{\left\| \bigcup \left(L^{A-B} \cup L^{A\cap B} \right) \right\|}{M} \left\| D^A \right\| + \frac{\left\| \bigcup \left(L^{B-A} \cup L^{A\cap B} \right) \right\|}{M} \left\| D^B \right\|$$

The CPU cost of the complete method is the following:

$$\text{cost}_{CPU} = \sum_{i=1}^{K^{A-B}} \left(\left| C_{i+1}^{A-B} \right\| D^{A-B} \right| + \left| L_i^{A-B} \right|^2 \right) + \sum_{i=1}^{K^{A\cap B}} \left(\left| C_{i+1}^{A\cap B} \right\| D^{A\cap B} \right| + \left| L_i^{A\cap B} \right|^2 \right) +$$

$$\sum_{i=1}^{K^{B-A}} \left(\left| C_{i+1}^{B-A} \right\| D^{B-A} \right| + \left| L_i^{B-A} \right|^2 \right) + \left| \bigcup (L^{A-B} \cup L^{A\cup B}) \right\| D^A \right| + \left| \bigcup (L^{B-A} \cup L^{A\cup B}) \right\| D^B \right|$$

5. CONCLUSIONS

In this paper we have presented the problem of efficient executing batches of data mining queries. We have built a relational algebra model for a level-wise association discovery algorithm and we used this model to describe our methods of executing batched data mining queries. For the three described methods, we analyzed their performance in terms of I/O cost and CPU cost.

REFERENCES

1. Agrawal R., Imielinski T., Swami A.: Mining Association Rules Between Sets of Items in Large Databases. Proc. of the 1993 ACM SIGMOD Conf. on Management of Data (1993)
2. Agrawal R., Srikant R.: Fast Algorithms for Mining Association Rules. Proc. of the 20th Int'l Conf. on Very Large Data Bases (1994)
3. Ceri S., Meo R., Psaila G.: A New SQL-like Operator for Mining Association Rules. Proc. of the 22nd Int'l Conference on Very Large Data Bases (1996)
4. Cheung D.W., Han J., Ng V., Wong C.Y.: Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique. Proc. of the 12th ICDE (1996)
5. Han J., Fu Y., Wang W., Chiang J., Gong W., Koperski K., Li D., Lu Y., Rajan A., Stefanovic N., Xia B., Zaiane O.R.: DBMiner: A System for Mining Knowledge in Large Relational Databases. Proc. of the 2nd KDD Conference (1996)
6. Imielinski T., Mannila H.: A Database Perspective on Knowledge Discovery. Communications of the ACM, Vol. 39, No. 11 (1996)
7. Imielinski T., Virmani A., Abdulghani A.: Datamine: Application programming interface and query language for data mining. Proc. of the 2nd KDD Conference (1996)
8. Morzy T., Wojciechowski M., Zakrzewicz M.: Data Mining Support in Database Management Systems. Proc. of the 2nd DaWaK Conference (2000)
9. Morzy T., Zakrzewicz M.: SQL-like Language for Database Mining. ADBIS'97 Symposium (1997)
10. Nag B., Deshpande P.M., DeWitt D.J.: Using a Knowledge Cache for Interactive Discovery of Association Rules. Proc. of the 5th KDD Conference (1999)
11. Thomas S., Bodagala S., Alsabti K., Ranka S.: An Efficient Algorithm for the Incremental Updation of Association Rules in Large Databases. Proc. of the 3rd KDD Conference (1997)
12. Toivonen H.: Sampling Large Databases for Association Rules. Proc. of the 22nd Int'l Conference on Very Large Data Bases (1996)
13. Wojciechowski M., Zakrzewicz M.: Itemset Materializing for Fast Mining of Association Rules. Proc. of the 2nd ADBIS Conference (1998)