Processing Frequent Itemset Discovery Queries by Division and Set Containment Join Operators

Ralf Rantzau
Department of Computer Science, Electrical Engineering and Information Technology
University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart, Germany
rrantzau@acm.org

ABSTRACT

SQL-based data mining algorithms are rarely used in practice today. Most performance experiments have shown that SQL-based approaches are inferior to main-memory algorithms. Nevertheless, database vendors try to integrate analysis functionalities to some extent into their query execution and optimization components in order to narrow the gap between data and processing. Such a database support is particularly important when data mining applications need to analyze very large datasets or when they need access current data, not a possibly outdated copy of it.

We investigate approaches based on SQL for the problem of finding frequent itemsets in a transaction table, including an algorithm that we recently proposed, called *Quiver*, which employs universal and existential quantifications. This approach employs a table schema for itemsets that is similar to the commonly used vertical layout for transactions: each item of an itemset is stored in a separate row. We argue that expressing the frequent itemset discovery problem using quantifications offers interesting opportunities to process such queries using set containment join or set containment division operators, which are not yet available in commercial database systems. Initial performance experiments reveal that Quiver cannot be processed efficiently by commercial DBMS. However, our experiments with query execution plans that use operators realizing set containment tests suggest that an efficient processing of Quiver is possible.

Keywords

association rule discovery, relational division, set containment join

1. INTRODUCTION

The frequent itemset discovery algorithms used in today's data mining applications typically employ sophisticated in-memory data structures, where the data is stored into and retrieved from flat files. This situation is driven by the need for high-performance ad-hoc data mining in the areas of health-care or e-commerce, see for example [12] for an overview. However, ad-hoc mining often comes at a high cost because huge investments into powerful hardware and sophisticated software are required.

1.1 Why Data Mining with SQL?

From a performance perspective, data mining algorithms that are implemented with the help of SQL are usually considered inferior to algorithms that process data outside the database system [21]. One of the most important reasons is that offline algorithms employ sophisticated in-memory data structures and try to scan the data as few times as possible while SQL-based algorithms either require several scans over the data or require many and complex joins between the input tables. However, DBMS already provide techniques to deal with huge datasets. These techniques need to be re-implemented in part if offline algorithms are used to analyze data that is so large that the in-memory data structures grow beyond the size of the computer's main-memory.

One can classify data mining model generation algorithms along the following list of features:

- On-line vs. off-line mining: While the algorithm is running, the user may change parameters and retrieve intermediate or approximate results as early as possible.
- In-place vs. out-of-place mining: The algorithm retrieves original data from the database itself or it operates on a copy of the original data, perhaps with a data layout that is optimal for the algorithm.
- Database-coupled vs. database-decoupled mining: The algorithm uses the DBMS merely as a file system, i.e., it stores and loads data using trivial queries or it exploits the query processor capabilities of the DBMS for nontrivial mining queries.

Obviously, some algorithms lie somewhere in-between these categories. For example, there are implementations of the frequent itemset discovery method that allow a user interaction only to some extent: a user can increase the minimum support threshold but maybe not decrease it because this would require revisiting some previously seen transactions. Hence, such an implementation realizes on-line mining only up to a certain degree.

We believe that there are datasets of high potential value that are so large that the only way to analyze them in their entirety (without sampling) is to employ algorithms that exploit the scalable query processing capability of a database system, i.e., these datasets require in-place, database-coupled mining, however, at the potential cost of allowing only off-line mining. On-line mining can be achieved if the query execution plans are based on non-blocking operators along the "critical paths" of the data streams from the input tables to the root operator of the plan. A framework for such operators supporting queries within the knowledge discovery process has been investigated for example in [5].

1.2 The Problem of Frequent Itemset Discovery

We briefly introduce the widely established terminology for frequent itemset discovery. An *item* is an object of analytic interest, like a product of a shop or a URL of a document on a web site. An *itemset* is a set of items and a k*-itemset* contains k items. A *transaction* is an itemset representing a fact, like a basket of distinct products purchased together or a collection of documents requested by a user during a web site visit.

Given a set of transactions, the *frequent itemset discovery problem* is to find itemsets within the transactions that appear at least as frequently as a given threshold, called *minimum support*. For example, a user can define that an itemset is frequent if it appears in at least 2% of all transactions.

Almost all frequent itemset discovery algorithms consist of a sequence of steps that proceed in a bottom-up manner. The result of the kth step is the set of frequent k-itemsets, denoted as F_k. The first step computes the set of frequent items or 1-itemsets F_1. Each of the following step consists of two phases:

1. The *candidate generation phase* computes a set of potential frequent k-itemsets from F_{k-1}. The new set is called C_k, the set of candidate k-itemsets. It is a superset of F_k.

2. The *support counting phase* filters out those itemsets from C_k that appear more frequently in the given set of transactions than the minimum support and stores them in F_k.

All known SQL-based algorithms follow this "classical" two-phase approach. There are other, non-SQL-based approaches such as *frequent-pattern growth* [7], which do not require a candidate generation phase. However, the frequent-pattern growth algorithm employs a (relatively complex) main-memory data structure called frequent-pattern tree, which disqualifies it for a straightforward comparison with SQL-based algorithms.

1.3 Set Containment Tests

The *set containment join* (SCJ) $\bowtie_{\subseteq}$ is a join between two relations R and S on set-valued attributes $R.a$ and $S.b$, the join condition being that $a \subseteq b$. It is considered an important operator for queries involving set-valued attributes [8; 9; 11; 15; 14; 17; 18; 25]. For example, set containment test operations have been used for optimizing a workload of continuous queries, in particular for checking if one query is a subquery of another [4]. Another application area is content-based retrieval in document databases, when one tries to find a collection of documents containing a set of keywords.

Frequent itemset discovery, one of the most important data mining operations, is an excellent example of a problem that can be expressed using set containment joins: Given a set of transactions T and a single (candidate) itemset i, how many transactions $t \in T$ fulfill the condition $i \subseteq t$? If this number is beyond the minimum support threshold the itemset is considered frequent. In general, we would like to test a whole set of itemsets I for containment in T: *Find the number of tuples in* $I \bowtie_{\subseteq} T$ *for each distinct transaction in* T.

transaction	*items*
1001	$\{A, D\}$
1002	$\{A, B, C, D\}$
1003	$\{A, C, D\}$

(a) *Transaction*

itemset	*items*
101	$\{A, B, D\}$
102	$\{A, C\}$

(b) *Itemset*

itemset	*items*	*transaction*	*items*
101	$\{A, B, D\}$	1002	$\{A, B, C, D\}$
102	$\{A, C\}$	1002	$\{A, B, C, D\}$
102	$\{A, C\}$	1003	$\{A, C, D\}$

(c) *Contains*

Figure 1: An example set containment join: *Itemset* $\bowtie_{items \subseteq items}$ *Transaction* $=$ *Contains*

transaction	*item*
1001	*A*
1001	*D*
1002	*A*
1002	*B*
1002	*C*
1002	*D*
1003	*A*
1003	*C*
1003	*D*

(a) *Transaction* (single dividend)

itemset	*item*
101	*A*
101	*B*
101	*D*
102	*A*
102	*C*

(b) *Itemset* (several divisors)

transaction	*itemset*
1002	101
1002	102
1003	102

(c) *Contains* (several quotients)

Figure 2: An example set containment division: *Transaction* $\div_{\{item\} \supseteq \{item\}}$ *Itemset* $=$ *Contains*

The set containment join takes unnormalized relations as input, i.e., the relations are not in the first normal form (1NF), which would require that the attributes have only atomic values. We have recently introduced the *set containment division* (SCD) operator $\div_{\supseteq}$, which is equivalent to SCJ but the relations are in 1NF [20]. It is a generalization of the well-known relational division operator. Given two relations $R(A \cup B)$ and $S(C \cup D)$, where $A = \{a_1, \ldots, a_m\}$, $B = \{b_1, \ldots, b_n\}$, $C = \{c_1, \ldots, c_n\}$, and $D = \{d_1, \ldots, d_p\}$ are sets of attributes and the attribute domains of b_i are compatible with c_i for all $1 \leq i \leq n$. The sets A, B, and C have to be nonempty, while D may be empty. The set containment division is defined by the algebra expression

$$\begin{aligned} R \div_{B \supseteq C} S &= \bigcup_{x \in \pi_D(S)} \left(\left(R \div \pi_C \left(\sigma_{D=x} \left(S \right) \right) \right) \times \left(x \right) \right) \\ &= T \left(A \cup D \right). \end{aligned}$$

Note that the superset symbol ($\supseteq$) is used to contrast SCD from the traditional division operator and to show its similarity to the SCJ operator, which has a subset relation ($\subseteq$). It does not mean that SCD compares set-valued attributes, unlike SCJ.

If $D = \emptyset$, i.e., if S consists of a single group or "set" of elements, then the set containment division becomes the normal division operator:

$$R \div_{B \supseteq C} S = R \div S = T\,(A)\,,$$

where R is called *dividend*, S *divisor*, and T *quotient*.

To compare the two operators SCJ and SCD, Figures 1 and 2 sketch a simple example that uses equivalent input data for the operators. SCJ is based on unnormalized data while SCD requires input relations in 1NF. Each tuple of the result relation *Contains* delivers the information on which itemset is contained in which transaction.

In this paper, we argue that if SQL would allow expressing SCJ and SCD problems in an intuitive manner and if several algorithms implementing these operators were available in a DBMS, this would greatly facilitate the processing of queries for frequent itemset discovery.

1.4 Paper Overview

The remainder of this paper is organized as follows. In Section 2, we discuss frequent itemset algorithms that employ SQL queries. Query execution plans that realize the queries are discussed in Section 3. Section 4 explains algorithms for the SCJ and SCD operators and their implementation based on an open source Java class library for building query processors. Several experiments that assess the performance of the algorithms based on both synthetic and real-life datasets are presented in Section 5. We conclude the paper in Section 6 and give a brief outlook on future work.

2. FREQUENT ITEMSET DISCOVERY WITH SQL

Algorithms for deriving association rules with SQL have been studied in great detail in the past. Frequent itemset discovery, a preprocessing phase of association rule discovery, is more time-consuming than the subsequent rule generation phase [2]. Therefore, all association rule discovery algorithms described in the literature strive to optimize the frequent itemset generation phase.

2.1 Related Work

The *SETM* algorithm is the first SQL-based approach [10] described in the literature. Subsequent work suggested improvements of SETM. For example, in [24] views are used instead of some of the tables employed in SETM. The authors also suggest a reformulation of SETM using subqueries. The performance of SETM on a parallel DBMS has been studied [16]. The results have shown that SETM does not perform well on large datasets and new approaches have been devised, like for example *K-Way-Join*, *Three-Way-Join*, *Subquery*, and *Two-Group-Bys* [22]. These new algorithms differ only in the statements used for support counting. They use the same statement for generating C_k, as shown in Figure 3 for the example value $k = 4$. The statement creates a new candidate k-itemset by exploiting the fact that all of its k subsets of size $k-1$ have to be frequent. This condition is called *Apriori property* because it was originally introduced in the *Apriori* algorithm [2; 13]. Two frequent subsets are picked to construct a new candidate. These itemsets must have the same items from position 1 to $k-1$. The new candidate is further constructed by adding the kth items of both itemsets in a lexicographically ascending order. In addition, the statement checks if the $k-2$ remaining subsets of the new candidates are frequent as well, expressed by the two "skip item" predicates in Figure 3.

```
INSERT
INTO C4 (itemset, item1, item2, item3, item4)
SELECT newid(), item1, item2, item3, item4
FROM (
  SELECT a1.item1, a1.item2, a1.item3, a2.item3
  FROM   F3 AS a1, F3 AS a2, F3 AS a3, F3 AS a4
  WHERE  a1.item1 = a2.item1 AND
         a1.item2 = a2.item2 AND
         a1.item3 < a2.item3 AND
         -- Apriori property.
         -- Skip item1.
         a3.item1 = a1.item2 AND
         a3.item2 = a1.item3 AND
         a3.item3 = a2.item3 AND
         -- Skip item2.
         a4.item1 = a1.item1 AND
         a4.item2 = a1.item3 AND
         a4.item3 = a2.item3) AS temporary;
```

Figure 3: Candidate generation phase in SQL-92 for $k = 4$. Such a statement is used by all known algorithms that have a horizontal table layout.

Another well-known approach presented in [22] called *K-Way-Join* uses k instances of the transaction table and joins it k times with itself and with a single instance of C_k. The support counting phase of K-Way-Join is illustrated in Figure 4(a), where we contrast it to an equivalent approach using a vertical table layout in Figure 4(b) that is similar to our *Quiver* approach, discussed below.

The algorithms presented in [22] perform differently for different data characteristics. The authors report that Subquery is the best algorithm overall compared to the other approaches based on SQL-92. The reason is that it exploits common prefixes between candidate k-itemsets when counting the support.

More recently, an approach called *Set-oriented Apriori* has been proposed [23]. The authors argue that too much redundant computation is involved in each support counting phase. They claim that it is beneficial to save the information about which item combinations are contained in which transaction, i.e., Set-oriented Apriori generates an additional table $T_k(\mathit{transaction}, \mathit{item}_1, \ldots, \mathit{item}_k)$ in the kth step of the algorithm. The algorithm derives the frequent itemsets by grouping on the k items of T_k and it generates T_{k+1} using T_k. Their performance results have shown that Set-oriented Apriori performs better than Subquery, especially for high values of k.

2.2 The Quiver Algorithm

We have recently introduced a new SQL-based algorithm for deriving frequent itemsets called *Quiver (QUantified Itemset discoVERy)* [19]. The basic idea of the algorithm is to use a vertical layout for representing itemsets in a relation in the same way as transactions are represented, i.e., the relations storing candidate and frequent itemsets have the same schema $(\mathit{itemset}, \mathit{pos}, \mathit{item})$, similar to the schema for transactions $T(\mathit{transaction}, \mathit{item})$. Table 1 (taken from [19]) illustrates the difference between the horizontal and the vertical layout for transactions and itemsets.

Quiver's SQL statements employ *universal quantification* in both the candidate generation and the support counting phase. Quiver constructs a candidate $(k+1)$-itemset i from two frequent k-itemsets $f_1.\mathit{itemset}$ and $f_2.\mathit{itemset}$ by checking if for **all** *item* values of f_1 at position $1 \leq f_1.\mathit{pos} =$

```
INSERT
INTO     S4 (itemset, support)
SELECT   c1.itemset, COUNT(*)
FROM     C4 AS c1,
         T AS t1, T AS t2, T AS t3, T AS t4
WHERE    c1.item1       = t1.item        AND
         c1.item2       = t2.item        AND
         c1.item3       = t3.item        AND
         c1.item4       = t4.item        AND
         t1.transaction = t2.transaction AND
         t1.transaction = t3.transaction AND
         t1.transaction = t4.transaction
GROUP BY c.itemset
HAVING   COUNT(*) >= @minimum_support;

INSERT
INTO   F4 (itemset, item1, item2, item3, item4)
SELECT c1.itemset, c1.item1, c1.item2, c1.item3, c1.item4
FROM   C4 AS c1, S4 AS s1
WHERE  c1.itemset = s1.itemset;
```

(a) Original, horizontal version of K-Way-Join

```
INSERT
INTO     S4 (itemset, support)
SELECT   c1.itemset, COUNT(*)
FROM     C4 AS c1, C4 AS c2, C4 AS c3, C4 AS c4,
         T  AS t1, T  AS t2, T  AS t3, T  AS t4
WHERE    c1.itemset     = c2.itemset     AND
         c1.itemset     = c3.itemset     AND
         c1.itemset     = c4.itemset     AND
         t1.transaction = t2.transaction AND
         t1.transaction = t3.transaction AND
         t1.transaction = t4.transaction AND
         c1.item        = t1.item        AND
         c2.item        = t2.item        AND
         c3.item        = t3.item        AND
         c4.item        = t4.item        AND
         c1.pos         = 1              AND
         c2.pos         = 2              AND
         c3.pos         = 3              AND
         c3.pos         = 4
GROUP BY c1.itemset
HAVING   COUNT(*) >= @minimum_support;

INSERT
INTO   F4 (itemset, pos, item)
SELECT c1.itemset, c1.pos, c1.item
FROM   C4 AS c1, S4 AS s1
WHERE  c1.itemset = s1.itemset;
```

(b) Vertical version of K-Way-Join

Figure 4: Support counting phase of K-Way-Join for $k = 4$

$f_2.pos = i.pos \leq k - 1$ the condition $f_1.item = f_2.item = i.item$ holds. This is the prefix construction used in the Apriori algorithm mentioned in Section 2.1. Similar predicates are added to the WHERE clause of the SQL query to realize the Apriori property, i.e., the "skip item" predicates mentioned in the SQL query used for retrieving horizontal candidates in Figure 3. The SQL statement used for this phase is quite lengthy, therefore we do not present it in this paper. It is shown in [19] together with an equivalent query in tuple relational calculus to emphasize the use of universal quantification (the universal quantifier "$\forall$").

The frequent itemset counting phase of Quiver uses universal quantification as well. It is used to express the set containment test: For each candidate itemset i, find the number of transactions where for each transaction t there is a value $t.item = i.item$ for **all** values of $i.pos$. Note that this condition holds for itemsets of arbitrary size k. No further restriction involving the parameter k is necessary. Hence, the same SQL statement, depicted in Figure 5(a), can be used for any iteration of the frequent itemset discovery algorithm. The nested NOT EXISTS predicate realizes the universal quantifier.

<table>
<tr><th>Layout</th><th>Transactions</th><th>Itemsets</th></tr>
<tr><td>horizontal (single-row/ multi-column)</td><td><table><tr><th><i>transaction</i></th><th><i>item</i>₁</th><th><i>item</i>₂</th><th><i>item</i>₃</th></tr><tr><td>1001</td><td><i>A</i></td><td><i>C</i></td><td><i>NULL</i></td></tr><tr><td>1002</td><td><i>A</i></td><td><i>B</i></td><td><i>C</i></td></tr></table></td><td><table><tr><th><i>itemset</i></th><th><i>item</i>₁</th><th><i>item</i>₂</th></tr><tr><td>101</td><td><i>A</i></td><td><i>B</i></td></tr><tr><td>102</td><td><i>A</i></td><td><i>C</i></td></tr></table></td></tr>
<tr><td>vertical (multi-row/ single-column)</td><td><table><tr><th><i>transaction</i></th><th><i>item</i></th></tr><tr><td>1001</td><td><i>A</i></td></tr><tr><td>1001</td><td><i>C</i></td></tr><tr><td>1002</td><td><i>A</i></td></tr><tr><td>1002</td><td><i>B</i></td></tr><tr><td>1002</td><td><i>C</i></td></tr></table></td><td><table><tr><th><i>itemset</i></th><th><i>pos</i></th><th><i>item</i></th></tr><tr><td>101</td><td>1</td><td><i>A</i></td></tr><tr><td>101</td><td>2</td><td><i>B</i></td></tr><tr><td>102</td><td>1</td><td><i>A</i></td></tr><tr><td>102</td><td>2</td><td><i>C</i></td></tr></table></td></tr>
</table>

Table 1: Table layout alternatives for storing the items of transactions and itemsets

Unfortunately, there is no universal quantifier defined in the SQL standard, not even in the upcoming standard SQL:2003. It is difficult for an optimizer to recognize that the query for support counting in Figure 5(a) is actually a division problem. Therefore, we suggest a set containment division operator in SQL, whose syntax is illustrated in Figure 5(b). Its semantics is equivalent to that of Figure 5(a).

Note that we cannot simply write

```
T AS t1 SET_CONTAINMENT_DIVIDE BY C AS c1
```

because the layout of the divisor table $C(itemset, pos, item)$ has the attribute pos, the lexicographical position of an item within an itemset, that is needed when candidates are created and that would incorrectly lead to grouping the divisor table on the attributes $(itemset, pos)$ instead of $(itemset)$. Hence, we omit this attribute in the SELECT clause of the subquery.

3. QUERY EXECUTION STRATEGIES

In this section, we show query execution plans (QEPs) produced by a commercial DBMS for some example SQL statements of the SQL-based frequent itemset discovery algorithms K-Way-Join and Quiver. In addition, we show how to realize the Quiver query for frequent itemset counting using a SCDs operator.

We compare Quiver to K-Way-Join because of their structural similarity. Remember that Quiver uses a vertical table layout for both itemsets and transactions while K-Way-Join uses a horizontal table layout for itemsets and a vertical layout for transactions. We are aware of the fact that K-Way-Join is not the best algorithm based on SQL-92 overall, even if our preliminary tests with a synthetic dataset has shown the opposite (see Section 5). However, we decided to derive first results by comparing these two approaches. Further performance experiments with other approaches will follow.

In Figures 6(a) and (b), we sketch the query execution plans that we derived during performance experiments with Microsoft SQL Server 2000 for several SQL algorithms discussed in Section 2.1. The plans (a) and (b) illustrate the execution strategies chosen by the optimizer for a given transaction table T and a candidate table C_4 for the queries shown in Figures 4(a) and 5(a). The operators in use are sort, row count, retrieval of the first row only (Top_1), rename (ρ), group (γ), join ($\bowtie$), left anti-semi-join ($\overline{\ltimes}$), index scan ($IScan$), and index lookup ($ISeek$).

The QEP (a) for the K-Way-Join query uses four hash-joins to realize the Apriori trick. It joins the candidate table C_4 four times with the transaction table.

The QEP (b) for the Quiver query using quantifications employs anti-semi-joins. Left anti-semi-join returns all rows of the left input that have no matching row in the right input. The top left anti-semi-join operator test for each combination of values $(c1.itemset, c1.transaction)$ on the left if at

```
INSERT
INTO     S (itemset, support)
SELECT   itemset, COUNT(DISTINCT transaction) AS support
FROM     (
  SELECT c1.itemset, t1.transaction
  FROM   C AS c1, T AS t1
  WHERE  NOT EXISTS (
    SELECT *
    FROM   C AS c2
    WHERE  NOT EXISTS (
      SELECT *
      FROM   T AS t2
      WHERE  NOT (c1.itemset = c2.itemset) OR
             (t2.transaction = t1.transaction AND
              t2.item        = c2.item)))
         ) AS Contains
GROUP BY itemset
HAVING   support >= @minimum_support;

INSERT
INTO   F (itemset, pos, item)
SELECT c1.itemset, c1.pos, c1.item
FROM   C AS c1, S AS s1
WHERE  c1.itemset = s1.itemset;
```

(a) Quiver using quantifications

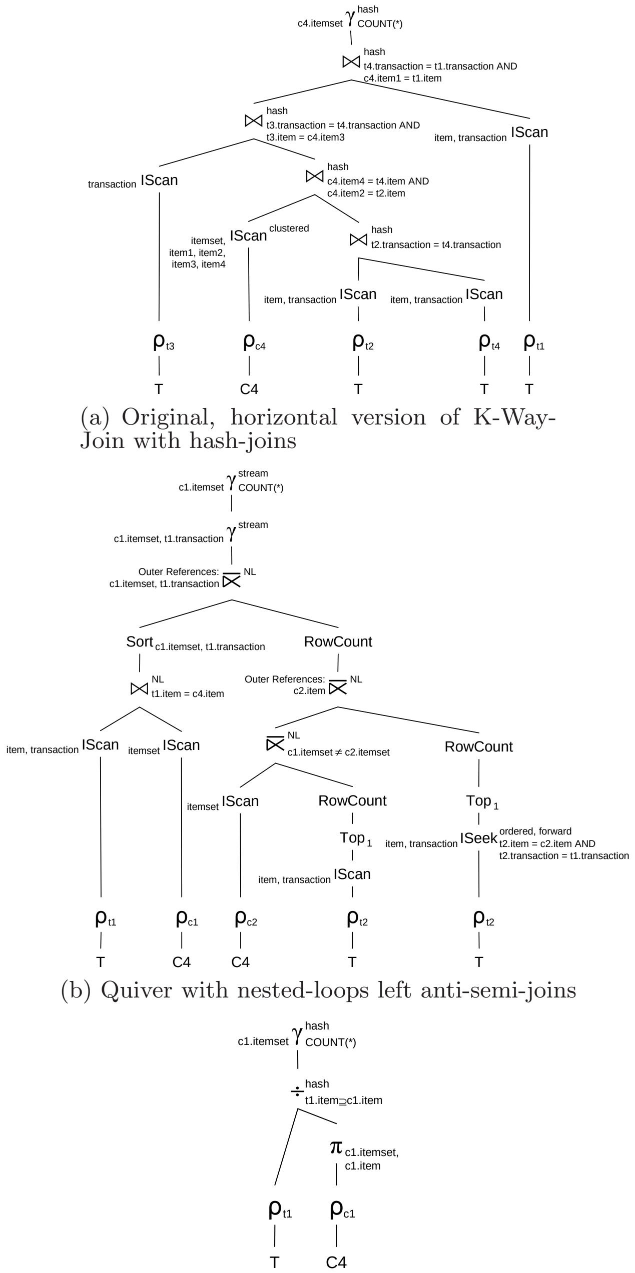
```
INSERT
INTO     S (itemset, support)
SELECT   c1.itemset, COUNT(*)
FROM     T AS t1 SET_CONTAINMENT_DIVIDE BY (
           SELECT itemset, item
           FROM   C
         ) AS c1
         ON (t1.item = c1.item)
GROUP BY c1.itemset
HAVING   COUNT(*) >= @minimum_support;

INSERT
INTO   F (itemset, pos, item)
SELECT c1.itemset, c1.pos, c1.item
FROM   C AS c1, S AS s1
WHERE  c1.itemset = s1.itemset;
```

(b) Quiver using a set containment division operation, specified by hypothetical SQL keywords

Figure 5: Support counting phase of Quiver for any value of k. We write C (F) instead of C_k (F_k) for the candidate (frequent) itemset table because of the independence of the parameter k.

least one row can be retrieved from the right input. If no, then the combination qualifies for the subsequent grouping and counting, otherwise the left row is skipped. A similar processing is done for the left anti-semi-join with the outer reference *c2.item*. An interesting point to note is that the index scan (not the seek) of *t2* on *item, transaction* is uncorrelated. Every access to this table is used to check if the transaction table is empty or not. For our problem of frequent itemset discovery this table is non-empty by default. In addition to the QEPs that have been derived for real SQL queries using a commercial DBMS, we illustrate a hypothetical QEP for the version of the Quiver algorithm that employs a set containment division operator in Figure 6(c). The corresponding query using hypothetical SQL keywords is specified in Figure 5(b).



(a) Original, horizontal version of K-Way-Join with hash-joins

(b) Quiver with nested-loops left anti-semi-joins

(c) Quiver with hash-based SCD

Figure 6: Example query execution plans computing F_4, generated for the SQL queries in Figures 4(a), 5(a), and 5(b)

4. ALGORITHMS FOR SCD AND SCJ

The term "hash" specified for the set containment division operator in the QEP in Figure 6(c) was used to illustrate that there may be several implementations (physical operators) in a DBMS realizing SCD, in this example based on the hash-division algorithm [6]. Since SCD is based on the division operator, as shown by the definition in Section 1.3, many implementations of division come into consideration for realizing SCD depending on the current data characteristics, e.g., the data may be grouped or even sorted on certain attributes, as discussed in [20].

The SCD operator that we used for the performance tests is based on the hash-division algorithm [6]. Hash-division uses two hash tables. The divisor hash table stores all rows of the divisor table (i.e., all rows of a single divisor group for SCD) and assigns to each row an integer number that is equal to $k-1$ for the kth row. The quotient hash table stores all distinct values of the dividend's quotient attributes (i.e., attribute set A in Section 1.3). These values are a superset of the final quotients, i.e., they are quotient candidates. For each quotient candidate value, a bitmap is kept, whose length is the number of divisor rows. For each divisor, the dividend table is scanned once. For each dividend row, the algorithm looks up the value i in the divisor hash table as an index for the the quotient bitmap. Then, the quotient candidate is looked up in the quotient hash table. If it is found, the bit at position i is set to *true*, otherwise a new quotient candidate and bitmap is inserted with all bits set to *false*. After the scan, the quotient candidates in the quotient hash table are returned whose bits are all *true*.

We have realized the QEPs shown in Figure 6 using the open source Java class library *XXL (eXtensible and fleXible Library for data processing)* for building query processors [3]. Some example classes are *BTree*, *Buffer*, *HashGrouper*, *Join*, and *Predicate*. Interestingly, they provide a class called *SortBasedDivision* that implements the merge-division algorithm. However, several necessary operators for our purposes like nested-loops anti-semi-join and hash-join had to be built from scratch.

Several algorithms for SCJs have been proposed, as mentioned in Section 1.3. It may seem curious that we only show a QEP and experiments using Quiver realized with our home-made SCD instead of the better-known SCJ operator. In fact, we have realized an implementation of SCJ using the *Adaptive Pick-and-Sweep Join* algorithm [15], which is claimed to be the most efficient SCJ algorithm to date and made initial performance tests, presented in Section 5.2. However, we cannot yet report on a thorough comparison of different implementations for SCD and SCJ as it is part of our ongoing work. Note that an interesting recent publication [11] on SCJs did not investigate this algorithm.

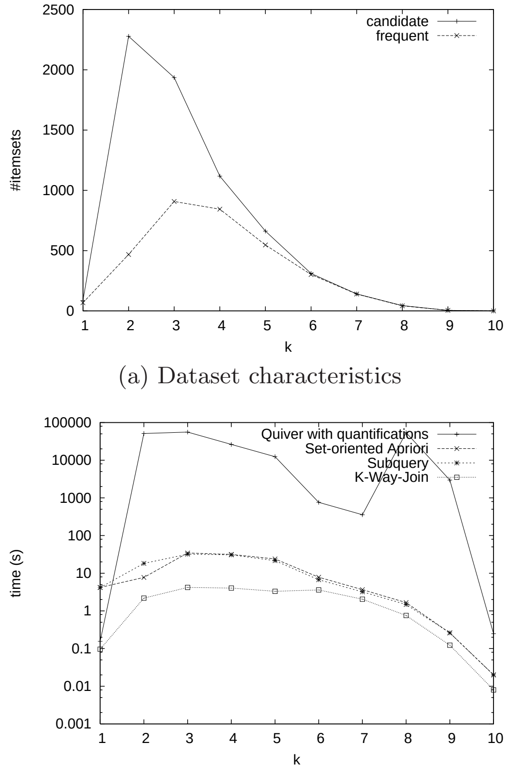
5. EXPERIMENTS

5.1 Commercial DBMS

In Figure 7, we highlight some results of experiments on a commercial database system to assess the query execution plans and performance of the SQL-based algorithms K-Way-Join, Subquery, Set-oriented Apriori, and Quiver using quantifications. We used Microsoft SQL Server 2000 Standard Edition on a 4-CPU Intel Pentium-III Xeon PC with 900 MHz, 4 GB main memory, and Microsoft Windows 2000 Server. Due to lack of space, we cannot give the details on the indexes provided on the tables and what primary keys were chosen. However, it was surprising that K-Way-Join performed best for this (admittedly small) dataset, unlike reports in related work mentioned in Section 2.1. We provide more information on this comparison in [19].

5.2 XXL Query Execution Plans

In addition to the experiments on a commercial DBMS, we have executed the manual implementation of the QEPs with Java and XXL using one synthetic and one real-life dataset as the transactions table. Table 2 summarizes the characteristics of the datasets. The synthetic data has been produced using the *SDSU Java Library*, based on the well-known IBM data generation tool mentioned in [2]. The real-life transaction dataset called *BMS-WebView-2* by Blue Martini Software [26] contain several months of click-stream data of an e-commerce web site.



(a) Dataset characteristics

(b) Execution times. Note the logarithmic scale of the y-axis.

Figure 7: Experiments with SQL-based algorithms on a commercial DBMS for the synthetic dataset T5.I5.D10k with minimum support of 1% (100 transactions)

Type of data	Dataset	Distinct items	Rows	Trans-actions	Trans. size	
					avg.	max.
synthetic	T5.I5.D5k	86	30,268	5,000	6.05	18
real-life	BMS-WebView-2	3,340	358,278	77,512	4.62	161

Table 2: Overview of transaction datasets

5.2.1 Set Containment Division

The plans were executed on a dual-CPU Intel Pentium-III PC with 600 MHz, 256 MB main memory, Microsoft Windows 2000 Personal Edition, and JRE 1.4.1. The data resided on a single local hard disk.

We used subsets of the transaction datasets and subsets of the candidate 4-itemsets to derive frequent 4-itemsets for certain minimum support values. The synthetic (real-life) data candidates were generated for a minimum support of 1% (0.04%). Figure 8 shows some results of our experiments. One can observe that the execution plan using SCD performed best for small numbers of candidate sets. Of course, this result is disappointing but it is simply the result of our straightforward implementation of set containment division. Our implementation follows strictly the logical operator's definition, which is a union of several divisions: each division takes the entire transaction table as dividend and a single itemset group as divisor. We plan to develop more efficient implementations of the operator in the future.

The plan chosen by the commercial DBMS based on anti-

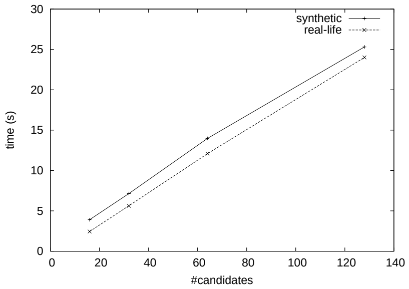


Figure 9: Experiments with set containment division for two types of data sets

semi-join was always a bad solution for the given data sets. K-Way-Join outperforms the other approaches for larger numbers of candidates.

5.2.2 *Set Containment Join*

In addition to the experiments comparing different query execution strategies, we have made initial performance tests using a set containment join operator instead of a set containment division, illustrated in Figure 9. We used the adaptive pick-and-sweep join algorithm. The query execution plan that we implemented using XXL is the same as in Figure 6(c), but a SCJ operator instead of the hash-based SCD. In the nested layout, the sets of items were represented by a Java vector of XXL tuples having a single item attribute. The test platform was a single-CPU Intel Pentium-4 PC with 1.9 GHz, 512 MB main memory, Microsoft Windows XP Professional SP 1, and JRE 1.4.1. The data resided on a single local hard disk. The experiments are based on the synthetic data set T5.I5.D10k and a subset of size 10,000 transactions of the real-life data set BMS-WebView-2. The candidate itemsets are similar to those used for the experiments before. For these small data sets, the operator processed the data with a linear performance for a growing number of candidate 4-itemsets.

6. CONCLUSIONS AND FUTURE WORK

We have shown that frequent itemset discovery is a prominent example for queries involving set containment tests. These tests can be realized by efficient algorithms for set containment join when the input data have set-valued attributes or by set containment division when the data are in 1NF. A DBMS has more options to solve the frequent itemset discovery problem optimally if it could choose to employ such operators inside the execution plans for some given data characteristics. No commercial DBMS offers an implementation of these operators to date. We believe that such a support would make data mining with SQL more attractive.

We currently realize query execution plans using algorithms for set containment joins and we compare them to plans involving set containment division. In future work, we will investigate also algorithms based on index structures supporting efficient containment tests, in particular the algorithms discussed in [8] and [11]. Furthermore, we will continue to study query processing techniques for switching between a vertical and a horizontal layout transparently (without changing the SQL statements) for data mining problems. Note that the investigation of the trade-offs of a horizontal and vertical data representation is not new. For example, the benefit of employing a vertical layout for querying e-commerce data has been observed in [1]. The techniques described there are generally applicable and we will study them in the context of the frequent itemset discovery problem.

7. ACKNOWLEDGMENTS

Daniel Böck's support of the performance experiments was greatly appreciated.

8. REFERENCES

[1] R. Agrawal, A. Somani, and Y. Xu. Storage and Querying of E-Commerce Data. In *Proceedings VLDB, Rome, Italy*, pages 149–158, September 2001.

[2] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proceedings VLDB, Santiago, Chile*, pages 487–499, September 1994.

[3] J. V. d. Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and M. Seeger. XXL – A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *Proceedings VLDB, Rome, Italy*, pages 39–48, September 2001.

[4] J. Chen and D. DeWitt. Dynamic Re-grouping of Continuous Queries. In *Proceedings VLDB, Hong Kong, China*, pages 430–441, August 2002.

[5] M. Gimbel, M. Klein, and P. Lockemann. Interactivity, Scalability and Resource Control for Efficient KDD Support in DBMS. In *Proceedings DTDM, Prague, Czech Republic*, pages 37–50, March 2002.

[6] G. Graefe and R. Cole. Fast Algorithms for Universal Quantification in Large Databases. *TODS*, 20(2):187–236, 1995.

[7] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2001.

[8] S. Helmer. *Performance Enhancements for Advanced Database Management Systems*. PhD thesis, University of Mannheim, Germany, December 2000.

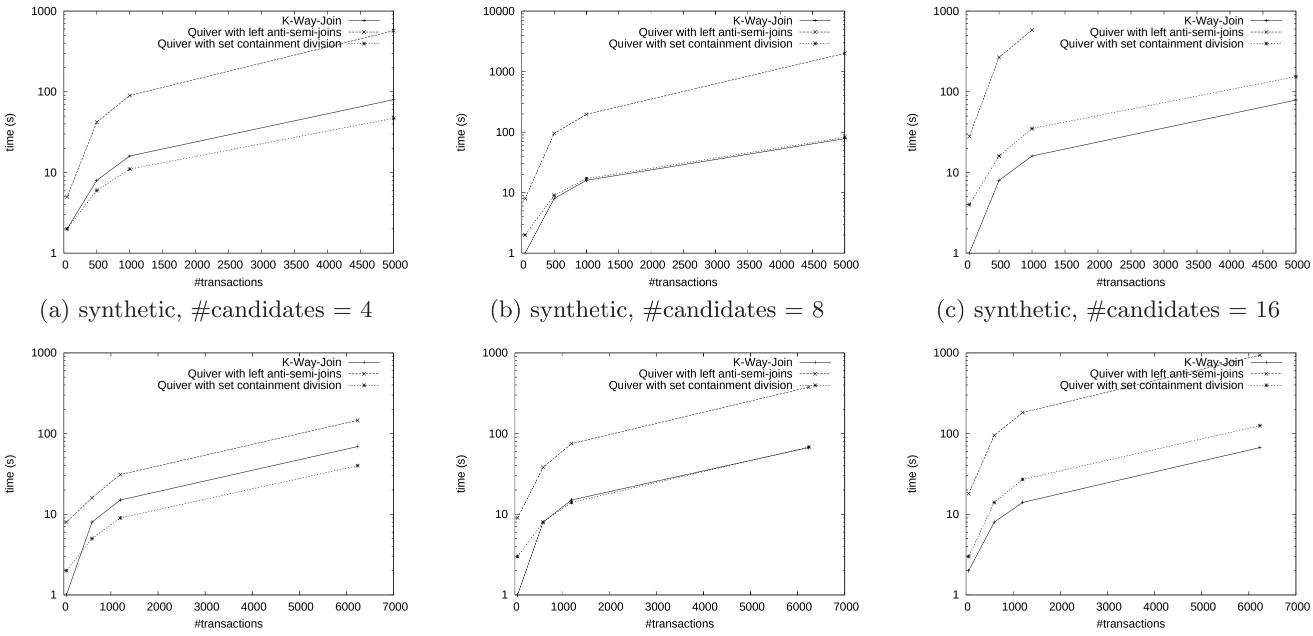
[9] S. Helmer and G. Moerkotte. Compiling Away Set Containment and Intersection Joins. Technical Report, University of Mannheim, Germany.

[10] M. Houtsma and A. Swami. Set-oriented Data Mining in Relational Databases. *DKE*, 17(3):245–262, December 1995.

[11] N. Mamoulis. Efficient Processing of Joins on Set-valued Attributes. In *Proceedings SIGMOD, San Diego, California, USA*, June 2003.

[12] W. Maniatty and M. Zaki. A Requirements Analysis for Parallel KDD Systems. In *Proceedings HIPS, Cancun, Mexico*, pages 358–365, May 2000.

[13] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient Algorithms for Discovering Association Rules. In *AAAI Workshop on Knowledge and Discovery in Databases, Seattle, Washington, USA*, pages 181–192, July 1994.



(a) synthetic, #candidates = 4
(b) synthetic, #candidates = 8
(c) synthetic, #candidates = 16
(d) real-life, #candidates = 4
(e) real-life, #candidates = 8
(f) real-life, #candidates = 16

Figure 8: Execution times of plans in Figure 6 for different numbers of candidate 4-itemsets and for two different datasets. Note the logarithmic scale of both axes.

[14] S. Melnik and H. Garcia-Molina. Divide-and-Conquer Algorithm for Computing Set Containment Joins. In *Proceedings EDBT, Prague, Czech Republic*, pages 427–444, March 2002.

[15] S. Melnik and H. Garcia-Molina. Adaptive Algorithms for Set Containment Joins. *TODS*, 28(1):56–99, March 2003.

[16] I. Pramudiono, T. Shintani, T. Tamura, and M. Kitsuregawa. Parallel SQL Based Association Rule Mining on Large Scale PC Cluster: Performance Comparison with Directly Coded C Implementation. In *Proceedings PAKDD, Beijing, China*, pages 94–98, April 1999.

[17] K. Ramasamy. *Efficient Storage and Query Processing of Set-valued Attributes*. PhD thesis, University of Wisconsin, Madison, Wisconsin, USA, 2002. 144 pages.

[18] K. Ramasamy, J. Patel, J. Naughton, and R. Kaushik. Set Containment Joins: The Good, The Bad and The Ugly. In *Proceedings VLDB, Cairo, Egypt*, pages 351–362, September 2000.

[19] R. Rantzau. Frequent Itemset Discovery with SQL Using Universal Quantification. In P. Lanzi and R. Meo, editors, *Database Support for Data Mining Applications*, volume 2682 of *LNCS*. Springer, 2003. To appear.

[20] R. Rantzau, L. Shapiro, B. Mitschang, and Q. Wang. Algorithms and Applications for Universal Quantification in Relational Databases. *Information Systems Journal, Elsevier*, 28(1):3–32, January 2003.

[21] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications. In *Proceedings SIGMOD, Seattle, Washington, USA*, pages 343–354, June 1998.

[22] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications. Research Report RJ 10107 (91923), IBM Almaden Research Center, San Jose, California, USA, March 1998.

[23] S. Thomas and S. Chakravarthy. Performance Evaluation and Optimization of Join Queries for Association Rule Mining. In *Proceedings DaWaK, Florence, Italy*, pages 241–250, August–September 1999.

[24] T. Yoshizawa, I. Pramudiono, and M. Kitsuregawa. SQL Based Association Rule Mining Using Commercial RDBMS (IBM DB2 UDB EEE). In *Proceedings DaWaK, London, UK*, pages 301–306, September 2000.

[25] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings SIGMOD, Santa Barbara, California, USA*, May 2001.

[26] Z. Zheng, R. Kohavi, and L. Mason. Real World Performance of Association Rule Algorithms. In *Proceedings SIGKDD, San Francisco, California, USA*, pages 401–406, August 2001.