Query Processing Concepts and Techniques to Support Business Intelligence Applications

Ralf Rantzau

University of Stuttgart
Germany

Motivation & Goals

Current Situation

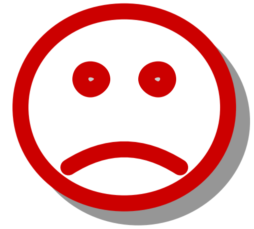
- **~~Database~~ Mining**
  Today, data mining tools do not analyze the data of the warehouse DBMS but they access flat files that have been extracted from the warehouse and that are adapted to the required input data structure of the mining method in use.
- **In-Memory Algorithms**
  SQL-based mining algorithms are considered inferior to highly tuned in-memory algorithms.
- **Powerful Technology Unused**
  Data sets to be analyzed typically reside in data warehouses, managed by powerful relational database systems.

Thesis Objectives

- **Identify Data Mining Primitives**
  Find basic operations that appear in data mining algorithms ("data mining primitives") and that require scalable and high-performance implementations.
  Example: *Frequent Itemset Discovery*
- **Design DB Operators Supporting the DM Primitives**
  Develop query processing strategies, like novel relational operators, to support SQL-based data mining algorithms. This includes investigating query optimization issues to enable a seamless integration of such operators into commercial database systems.
  Example: *Set Containment Division*

Pros and Cons of SQL-Based Data Mining

- **Data Currency**
  The latest updates applied to the data warehouse are reflected in the query result. No (replicated) data copies have to be maintained.
- **Scalability**
  If extremely large data sets are to be mined then it is much easier to design a scalable SQL-based algorithm than designing an algorithm that has to manage data in external files. The storage management is one of the key strengths of a database system.
- **Adaptability to Data**
  A database optimizer tries to find the best possible execution strategy based on the current data characteristics for a given query.



- **Less Portability**
  A data mining application that does not rely on a query language can be deployed more easily because no assumptions on the language's functionality have to be made.
- **Less Performance**
  A highly tuned black-box algorithm with in-memory data structures will always be able to outperform any query processor that employs a combination of generic algorithms.
- **Less Secrecy**
  A tool vendor does not want to reveal application logic. By employing SQL-based algorithms, the database administrator will be able to see these queries.

The *Quiver* Approach for Frequent Itemset Discovery

Universal Quantification

c = candidate with fixed *itemset* value
t = transaction with fixed *tid* value

$c \subseteq t \equiv$ **for all** values ***c.item*** there is a matching value ***t.item***

Vertical (1NF) Table Layout

Transaction (*tid*, *item*)
C_k (*itemset*, *pos*, *item*)
F_k (*itemset*, *pos*, *item*)

Quiver

(**Qu**antified **i**temset discovery using a **ver**tical table layout)

- SQL-based algorithm for computing frequent itemsets
- Both candidate generation phase and support counting phase can be expressed by universal quantifications over the items in itemsets and transactions
- Could make use of a new relational operator, called ***set containment division*** ($\div_{\supseteq}$), which is similar to the well-known set containment join ($\bowtie_{\subseteq}$) but assumes input tables in 1NF

Support Counting: K-Way-Join vs. Quiver Approach

Original K-Way-Join (Horizontal Layout)

```
INSERT
INTO     S3 (itemset, support)
SELECT   a1.itemset, COUNT(*)
FROM     C3 AS c, T AS t1, T AS t2, T AS t3
WHERE    c.item1 = t1.item AND
         c.item2 = t2.item AND
         c.item3 = t3.item AND
         t1.tid  = t2.tid  AND
         t1.tid  = t3.tid
GROUP BY c.itemset
HAVING   COUNT(*) >= @minimum_support;

INSERT
INTO   F3 (itemset, item1, item2, item3)
SELECT c.itemset, c.item1, c.item2, c.item3
FROM   C3 AS c, S3 AS s
WHERE  c.itemset = s.itemset;
```

Adapted K-Way-Join (Vertical Layout)

```
INSERT
INTO     S3 (itemset, support)
SELECT   a1.itemset, COUNT(*)
FROM     C3 AS c1, C3 AS c2, C3 AS c3,
         T  AS t1, T  AS t2, T  AS t3
WHERE    c1.itemset = c2.itemset AND
         c1.itemset = c3.itemset AND
         t1.tid     = t2.tid     AND
         t1.tid     = t3.tid     AND
         c1.item    = t1.item    AND
         c2.item    = t2.item    AND
         c3.item    = t3.item    AND
         c1.pos     = 1          AND
         c2.pos     = 2          AND
         c3.pos     = 3
GROUP BY c1.itemset
HAVING   COUNT(*) >= @minimum_support;

INSERT
INTO   F3 (itemset, pos, item)
SELECT c.itemset, c.pos, c.item
FROM   C3 AS c, S3 AS s
WHERE  c.itemset = s.itemset;
```

T (*tid*, *item*): transactions

S_k (*itemset*, *support*): support counts of candidate *k*-itemsets

Vertical Table Layout:

- C_k (*itemset*, *pos*, *item*): candidate *k*-itemsets
- F_k (*itemset*, *pos*, *item*): frequent *k*-itemsets

Horizontal Table Layout:

- C_k (*itemset*, $item_1$, …, $item_k$): candidate *k*-itemsets
- F_k (*itemset*, $item_1$, …, $item_k$): frequent *k*-itemsets

Quiver (Vertical Layout)

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

INSERT
INTO   Fk (itemset, pos, item)
SELECT c.itemset, c.pos, c.item
FROM   Ck AS c, Sk AS s
WHERE  c.itemset = s.itemset;
```

Frequent Itemset Discovery & Set Containment Division

Which *transactions* *contain* ALL items of a given *itemset*?

Transaction $\div_{\hat{E}}$ ***Itemsets*** = ***Contains***

tid	*item*
1001	diapers
1001	beer
1001	chips
1002	chips
1002	diapers
1003	beer
1003	avocados
1003	chips
1003	diapers

item	*itemset*
chips	1
beer	1
diapers	1
avocados	2
diapers	2

tid	*itemset*
1001	1
1003	1
1003	2

Find frequent itemsets by counting

Definition of set containment division operator:

$$R(a,b) \div_{b \supseteq c} S(c,d) = \bigcup_{x \in p_{d}(S)} ((R \div p_{c}(s_{d=x}(S))) \times (x)) = ?\,(a,d)$$

Expected Results & Future Work

- Demonstrate that SQL-based data mining algorithms are useful under certain conditions despite known problems
- Find a set of basic query processing operations that are shared by more sophisticated data mining algorithms

- Compare set containment join algorithms (set-valued attributes) with set containment division algorithms (based on 1NF tables)
- Develop optimization strategies for set containment division
- Investigate further data mining methods: classification & clustering