TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

Extending Database Functionality to Support Frequent Itemset Processing

Clara Nippl, Angelika Reiser, Bernhard Mitschang

TUM-I0011
SFB-Bericht Nr. 342/07/00 A
August 00

TUM–INFO–08-I0011-0/1.–FI

Anforderungen an: Prof. Dr. A. Bode
Sprecher SFB 342
Institut für Informatik
Technische Universität München
D-80290 München, Germany

Druck: Fakultät für Informatik der
Technischen Universität München

Extending Database Functionality to Support Frequent Itemset Processing

Clara Nippl[1], Angelika Reiser[1], Bernhard Mitschang[2]

[1]Computer Science Department, Technische Universität München
D - 80290 Munich, Germany
e-mail: nippl@in.tum.de

[2]Institut für Parallele und Verteilte Höchstleistungsrechner, Universität Stuttgart
D - 70565 Stuttgart, Germany

Abstract *Given the highly complementary nature of data mining and data warehousing it seems obvious that data mining should be performed as an integral part of the analysis process directly on the data already in the warehouse. In this paper we focus on frequent itemset processing and a tight integration approach. We introduce a novel concept to calculate candidate supports, called StreamJoin, as well as the corresponding pruning strategy to effectively reduce search complexity. We show how this approach can be efficiently embedded within a database engine, thus being able to exploit query optimization as well as parallel execution. Our approach avoids costly database scan operations, additional disk spoolings, intermediate blocking or preparatory phases. In contrast to other strategies, it yields a uniform processing within a single query execution plan and can be easily expressed and referenced via SQL-like interfaces.*

1 Introduction

Data mining can provide new insights into the relationships between data elements and provide analysts and decision-makers with new discoveries. Integrating the data warehouse DBMS with data mining makes sense for many reasons. First, obtaining clean, consistent data to mine is a primary challenge, implicitly provided by data warehouses. Second, transferring and reorganizing the data for mining is prohibitively costly in the case of large-scale warehouse applications. Third, it is advantageous to mine data from multiple sources to discover as many interrelationships as possible. This requirement is also fulfilled by warehouses that contain data from a number of sources. Finally, the continuously extended functionality of the database engines, including parallelization, can also only be used in an integrated environment.

One of the basic operations in data mining is the discovery of frequent sets. Given a set of transactions, where each transaction refers to a set of items, this operation consists of finding itemsets that occur in the database with certain user-specified frequency, called *minimum support*. The derived itemsets can be used for further processing, such as association rule mining [AY97], time series analysis, cluster analysis etc.

However, special requirements have to be fulfilled when integrating frequent itemset generation with large-scale databases, such as data warehouses. First, due to the large amounts of data stored, multiple database scans cause prohibitive overhead in terms of I/O costs. Second, it is essential to provide adequate pruning techniques to reduce the exponential complexity of search space exploration. Finally, the mining algorithm itself has to prove high efficiency.

Itemsets that have no superset that is frequent are called *maximal frequent itemsets (MFI)*. The set of all maximal frequent itemsets is called the *maximal frequent set (MFS)*. The sum of the lengths of all MFIs defines the *MFS volume*. Since this measure accounts for both the number of MFIs, as well as for their length, we consider that it reflects best the inherent complexity of the frequent itemset evaluation problem. The MFS implicitly defines the set of all frequent itemsets as well. Based on this observation, we propose a novel methodology to efficiently evaluate the maximal frequent set only, called *MFSSearch*. This strategy is based on a new operator, called *StreamJoin,* that efficiently calculates the support of a candidate itemset, as well as of all of its prefixes. A dynamic pruning technique that combines both top-down as well as bottom-up techniques is used throughout search space exploration. As a result, the complexity of the algorithm is only proportional to the MFS volume.

In contrast to other strategies, *MFSSearch* can efficiently be embedded into the database engine by using a single query execution plan. We show how the approach can easily be expressed in augmented SQL using user-defined table operators [JM99] and common table expressions [SQL99]. The processing scheme is non-blocking, i.e. first results (MFIs) can be delivered fast before the whole MFS is derived. Moreover, only selective disk accesses are necessary, depending on the current search space status. Intermediate result materializations or preparatory phases are not necessary, either. Furthermore, we point out how this approach can make full profit of the parallelization possibilities of the database engine.

2 Related work

Many approaches on integrating data mining with DBMSs are based on the bottom-up *Apriori* algorithm. Hence, they bear also the main drawbacks of this strategy, namely exponential complexity and multiple database scan operations [AS96, AY98]. Additionally, the proposed approaches also involve some kind of intermediate result materializations or preprocessing of data. On the other hand, some variants of the algorithm are not suitable for integration with data warehouses, as they perform modifications on the stored data [PCY97].

When comparing different possibilities of integrating the *Apriori* algorithm within a DBMS, in [STA98] the most promising scenario was found to be one based on a so-called *Vertical* format of the database. We discuss this work in more detail later in the paper.

Solutions using top-down or hybrid search strategies were proposed as well, such as the *MaxMiner* algorithm [Ba98], *PincerSearch* [LK98] or *MaxClique* and *MaxEclat* [ZP+97]. Although the pruning strategies of these algorithms reduce the search complexity considerably, they still involve multiple database passes or even preparatory phases. This has a negative effect on performance, as demonstrated later on in this paper.

Recent work [AAP00, HPY00] proposes the construction of highly condensed data structures in order to avoid database scans. However, it is not clear how these approaches perform e.g. in an ad-hoc data warehouse environment, where multiple users impose memory limitations and thus also disk spoolings. Other approaches concentrate on language extensions [MPC96, MPC98, HF+96] that are based on special operators to generate association rules. However, for the easy development of data mining applications it is important that the constituting operations are unbundled so that they may be shared [Ch98]. Thus, a better alternative is to provide a primitive that can be exploited more generally for different data mining applications. The strategy for MFS calculation proposed in this paper follows exactly this recommendation.

3 Data Mining Scenario

The decision on whether a given candidate itemset is frequent is the performance-critical operation in the MFS calculation. In our approach we want to directly map this primitive operation to a single database operator. In order to model this scenario we assume the following two tables: TRANS(*tid, item*) giving report of which transactions contain which items and CAND(*itemset, item*) telling which itemsets contain which items, i.e. the potential frequent itemsets. An example of two itemsets (*100* and *200*), three transactions (*1, 2* and *3*) and four items (*10, 22, 35, 43*) is shown in Fig. 1. This relational modeling of the data mining scenario supports that both the number of items per *tid*, as well as the number of items per *itemset* is variable and unknown during table creation time. In contrast, other representation alternatives, as e.g. all items of a *tid* appearing as different columns of a single tuple are not useful in practice [STA98]. Given this scenario and a parameter *minsup* defining the minimal support set by the user, the problem of deciding for a potential frequent itemset *IS* in the CAND table whether *IS* is frequent can be formulated as follows:

"Find (through the TRANS table) those transactions containing **all** items of *IS*. If the sum of the qualifying transactions exceeds *minsup*, then return *IS* as being a frequent itemset."

Evaluating this query involves a join on *item* between the two tables TRANS(*tid, item*) and CAND(*itemset, item*) for itemset = *IS*. This yields a set of tuples *(IS, item, tid)*. We will call the subset of those tuples for a given itemset which contain one specific item *I* a *stream*, denoted by $S_{IS,I}$. The streams for one specific itemset form a *group*. This means in general, every tuple from the CAND table defines one stream.

For the task to find frequent itemsets, the streams only form an intermediate result (*IR* in Fig. 1). E.g., for our potential frequent itemset *IS* we must find those transactions which contain **all** the items of IS. This means that for the final result (*FR* in Fig. 1) we have to join the different streams of the itemset *IS* on the *tid* attribute, i.e. we have to join all the streams within a group. In Fig. 1 we have an example for two itemsets, *100* and *200*, which contain two resp. three items. Therefore two streams resp. three streams are built for the itemsets. Joining the streams yields for this example into a two-way resp. three-way join. However, in the general case we do not have knowledge on the number of items per itemset, hence the number of joins to be performed on a group is variable as well.

This task is a kind of all-quantification. At the same time, it is a very primitive operation within

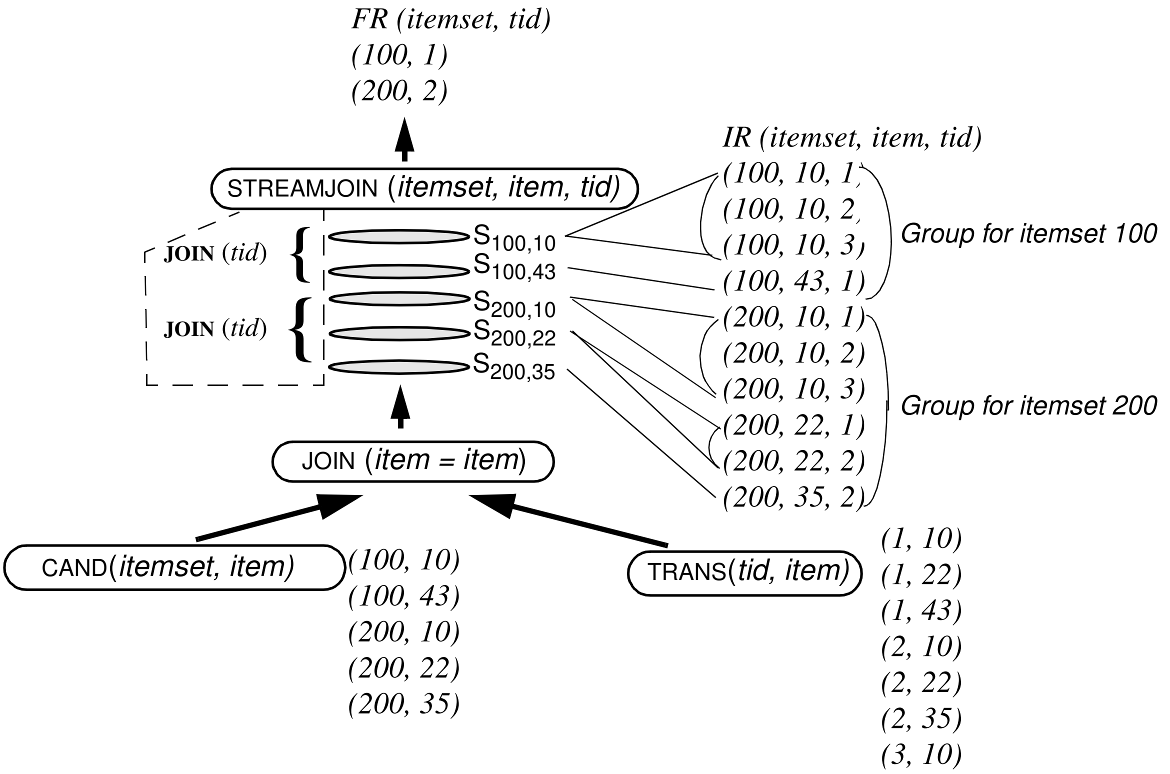


Fig. 1: Processing scenario for the evaluation of frequent itemsets

the processing of frequent itemsets. Hence, any efficient evaluation of this all-quantification directly supports the performance of the frequent itemset processing. Since all-quantification is not yet a frequently occurring operator in database processing, there are rarely implementations of it available [CK+97]. Here we designed our own solution, called *StreamJoin* operator, which perfectly fits into our algorithms. The operator will be introduced in the next section. Thereafter we will describe the particularities of our algorithm for efficient MFI candidate generation.

4 The *StreamJoin* Operator

We first describe the functionality of the operator. *StreamJoin* basically memorizes the incoming tuples as long as they belong to the same stream. Then, these tuples are joined with the next stream. This procedure continues for all streams of a group (i.e. for all items within a candidate itemset), such that at the end, only those tuples survive that support all streams within a group, i.e., all items within the given itemset. The *StreamJoin* operator has the following signature:

StreamJoin (Group-ID, Stream-ID, pred(Join-ID1), pred(Join-ID2), ...)

Two parameters specify the columns that define a group and the streams within a group; here, these parameters are *itemset* and *item*. The subsequent parameter(s) define(s) the join predicate(s). In the example from Fig. 1 it is a simple equi-join on the *tid* attribute. Thus the operator joins subsequent streams of the same group, as presented in Section 3. However, more complex predicates can be used as well to support e.g. pattern matching or sequence analysis [Ni99].

In order to perform well, the input of *StreamJoin* has to be grouped on the *Group-ID* and *Stream-ID* attributes. Obviously, this requirement can always be fulfilled by adequate sorting techniques. However, explicit sorting can mostly be avoided by adequate pre-processing of the data in the very same query execution plan.

For instance, assume that the CAND table in Fig. 1 is sorted on *itemset*. Consider the following evaluation alternatives w.r.t. the join between TRANS and CAND:

- an index-nested-loops join, using an index of the TRANS table on the *item* attribute; this is possible in almost all cases, since in most data warehouse schemas the central table has several indexes on the dimension attributes.
- a hash join, the CAND table being used as the probing table; please note that for an item domain containing l items the number of possible candidates is 2^l and thus the size of the CAND table might even exceed the size of the TRANS table.

In these cases, the join result is constituted as follows: for each tuple *(itemset, item)* of the CAND table a set of tuples *(itemset, item, tid)* is generated, yielding exactly a stream, i.e. the transac-

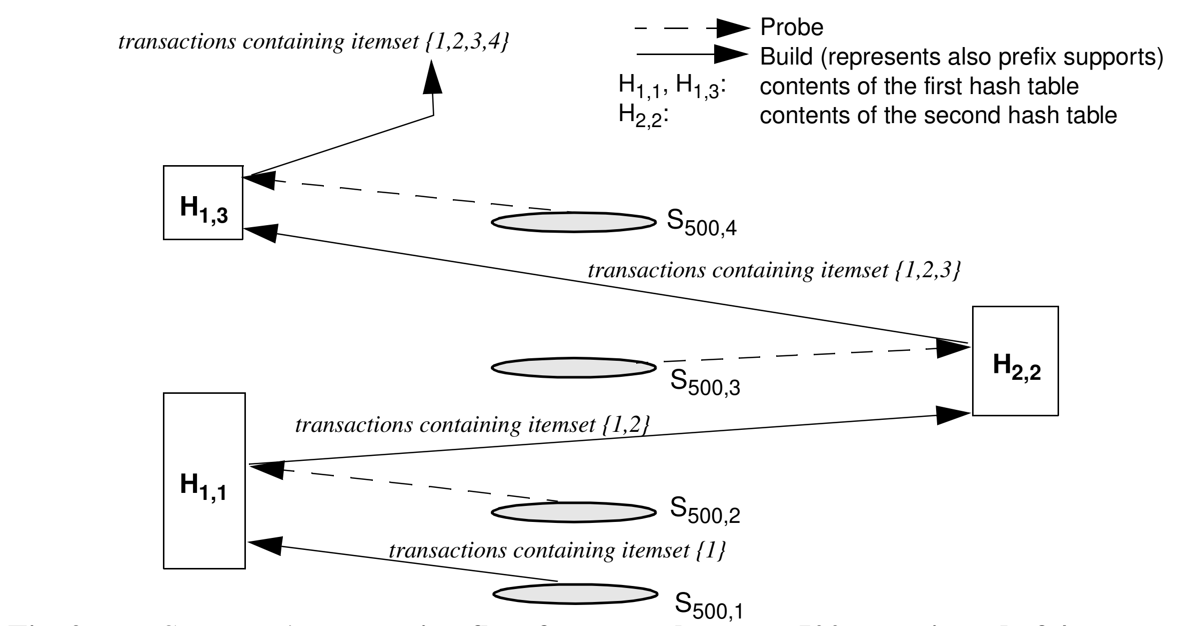


Fig. 2: ***StreamJoin*** **processing flow for example group 500, constituted of 4 streams**

tions that contain that specific item. For instance, the tuple *(100, 10)* has generated the stream $S_{100,10}$ consisting of the tuples *(100, 10, 1), (100, 10, 2)* and *(100, 10, 3)*. Hence, the necessary grouping of the intermediate result *IR* for the *StreamJoin* processing is already satisfied and no additional sort operations are necessary.

Based on this, our preliminary implementation for *StreamJoin* uses a dynamic hash-based approach with two hash tables. The strategy is illustrated in Fig. 2 for an example group, representing group *500*, constituted of four streams, derived from four items *1, 2, 3* and *4*. The first stream of each group is used to build the first hash table. The next stream is probed against this hash table, the matching tuples being inserted into the second hash table. At the beginning of the next iteration the first hash table is deleted. Similar to the previous iteration, the matching tuples are used to build up the new contents of the first hash table. This process continues until the next group is reached, or either a result of a probing phase or a constituting stream is empty. At the same time, the intermediate result (i.e. hash table) sizes decrease with each iteration, as the tuples which don't match the join condition are eliminated.

As indicated in Fig. 2 the continuous arrows also represent the transactions that contain the prefixes of the example itemset *{1,2,3,4}*. Hence, the corresponding supports can be easily evaluated by a simple subsequent *count(tid)* operation. Thereby, the frequent itemsets are those for which the calculated support exceeds *minsup*. In general, the following observation is valid:

Observation 1: Given an itemset *X = {1,2,..., N-1,N}*, by processing this itemset via the *StreamJoin* operator, we also obtain the supports of all prefixes *{1}, {1,2}... {1,2,...N-1}*. ❐

Hence, given a transaction table TRANS and a table CAND with candidate itemsets, the support of the candidates as well as of their prefixes can be efficiently evaluated within the database, performing a join on the two tables and pipelining the intermediate result *IR* into the *StreamJoin* operator, as already shown in Fig. 1.

5 The *MFSSearch* algorithm

An open question is how to guide the search space exploration in order to reduce search complexity and expensive database scans. In the hypothetical scenario from Fig. 1, we supposed that all candidate itemsets are stored in the CAND table. However, given the exponential complexity of the search space, this is impracticable for real-life applications and item domains. Hence, it is desirable to fill the CAND as much as possible with MFI candidates only. In our solution this sophisticated task of generating suitable candidates for the CAND table is performed by the *MFSSearch* algorithm that is in detail explained and analyzed in a separate paper that concentrates especially on pruning effects [NRM00]. The strategy expands the search space gradually, starting with the itemset containing all items. This maximizes the effect of prefix calculations that in turn reduce subsequent search efforts by means of pruning.

The *StreamJoin* processing is employed for the calculation of individual candidate supports. The produced results are used for the generation of further candidates. Thus, *MFSSearch* guides the search space exploration dynamically by already derived intermediate results. Please note that although *MFSSearch* starts with the itemset containing all items, it follows a *hybrid*, rather than a strict top-down search strategy. In the following, we briefly present this strategy as much as it is necessary to understand its combination with the *StreamJoin* operator.

At a given time, the search space is constituted of a limited number of expanded itemsets. These are organized in a stack. The algorithm basically extracts the topmost itemset of the stack and calculates its support (and prefix supports) by means of *StreamJoin*. If the itemset is found to be frequent, it is returned as such. If it is infrequent, *MFSSearch* expands the subsets of the current itemset, pushes them on the stack, and the evaluation starts again with the topmost itemset.

In the following, we would like to detail on the way subset itemsets are expanded.

Observation 2: Given an infrequent itemset $X = \{1,2,...,N-1,N\}$, in a top-down search it is necessary to test all of its subsets of level $N-1$. This can be done by successively eliminating the items $N-1, N-2,...1$ from X. It is not necessary to do this with item N, since $X - \{N\}$ is a prefix whose support is implicitly evaluated together with the support of X (see *Observation 1*). ❒

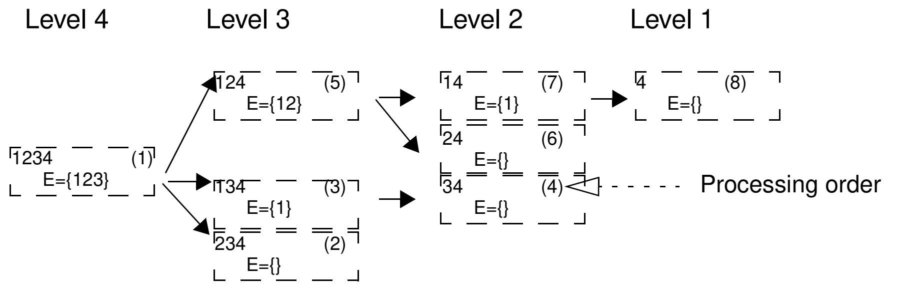


Fig. 3: Search space for frequent itemset evaluation over the finite item domain: 1,2,3,4.

However, if this procedure of generating subsets by eliminating the first *N-1* elements is applied recursively, duplicates are generated. For instance, both itemsets, *{1,3,4}* and *{2,3,4},* generate the subset *{3,4}*. In order to avoid this, we have assigned a so-called *elimination list (ElimList)* to each expanded itemset in the search space. The *ElimList* of an itemset specifies which of its constituting items are eligible to be used for subset expansion. Assume that the siblings X_1, X_2 ,..., X_{N-1} are expanded successively from *X*. The *Elimlists* of these siblings are then calculated by successively eliminating one element from the original *ElimList* of *X*. Thus for $X = \{1,2,..., N-1, N\}$ and $E_X = \{1,2,...,N-1\}$ the following sibling subsets and *ElimLists* are generated[1]:

$X_1 = \{1,2,...,N-2,N\}$, $E_1 = \{1,2,...,N-2\}$,

....

$X_{N-2} = \{1,3,...,N-1, N\}$, $E_{N-2} = \{1\}$,

$X_{N-1} = \{2,...,N-1, N\}$, $E_{N-1} = \varnothing$.

Fig. 3 shows a sample search space obtained in this way. As shown in [NRM00] the *ElimList* method guarantees a full expansion of the search space without duplicate generation and reduces in combination with the *StreamJoin* prefix processing the search space already by an order of magnitude.

The numbers in brackets show the order in which the candidate itemsets come to evaluation throughout search space evaluation, if no pruning methods are applied, i.e. the entire search space is spawn. This shows once again that *MFSSearch* doesn’t adopt a strict top-down exploration, but a mixed one. Thus, all direct subsets (corresponding to the *ElimList* method) of a current itemset *X* are first taken into consideration, before *MFSSearch* continues with the next sibling of *X*.

Based on this, we have the following *MFSSearch* algorithm:

1. Push the itemset containing all frequent items on the stack

1. For simplification, we use the notation E_i for E_{X_i}

```
2. while stack not empty
3.     Extract topmost itemset X from stack;
4.     Calculate support and prefix supports of X by means of StreamJoin;
5.     if X infrequent {
6.         Use X and infrequent prefixes of X for Bottom-up Pruning;
7.         Generate all subset of X according to the ElimList method;
8.         Push generated subsets of X on the stack}
9.     else{
10.        Use X for Top-Down Pruning;
11.        Add X to the output stream}          // X is a MFI
```

As can be seen in the pseudo code, *MFSSearch* employs both top-down as well as bottom-up[1] pruning techniques that are based on the results of previous support calculations (cf. Line 6. and 10. of the *MFSSearch* algorithm). Thereby, due to the *StreamJoin* processing scheme, both the support of an entire candidate itemset as well as of all of its prefixes can be used for pruning. Basically, the pruning strategy is founded on the following observation:

Observation 3: Given two itemsets X_1 and X_2, s.t. $X_1 \subseteq X_2$. In the *MFSSearch* exploration X_1 will be processed *after* X_2. ❐

Intuitively[2], this observation results also from Fig. 3, showing that despite of the hybrid search space exploration, any subset is explored only after all supersets have been explored as well. An important consequence is the fact that once an itemset is found to be frequent, it can immediately be returned, as *Observation 3* also guarantees that it is maximal. Hence, *MFSSearch* yields a *non-blocking* processing. This property is used in line 11. of the algorithm.

Both pruning techniques affect the expanded itemsets on the stack, by either deleting them and thus eliminating them from evaluation, or by reducing the number of the subsets to be expanded when these itemsets come to evaluation, i.e. when they are on the top of the stack. It can be shown that the number of items on the stack at a given time is *O(n)*, where *n* is the size of the considered item domain. This yields low memory requirements and high efficiency for the *MFSSearch* algorithm.

6 Integration with the DBMS

In Fig. 1 we visualized our approach to evaluate the supports of itemsets and prefixes within the database by using the *StreamJoin* operator. In this scenario, the candidates are given by the

---

1. *Top-down pruning* reduces the search space based on the fact that the subsets of a frequent itemset are also frequent. *Bottom-up pruning* eliminates supersets of known infrequent itemsets from evaluation.
2. The detailed proof of this observation as well as of the employed pruning techniques can be found in [NRM00].

(static) CAND table. Hence, in order to find the MFS, this table must contain all possible candidate itemsets, determined e.g. during a preprocessing step. However, as already mentioned, this approach is prohibitively costly in terms of time and disk space for real-life item domains.

Hence, in our solution the input for the *StreamJoin* operator is provided by the *MFSSearch* algorithm. In order to obtain an efficient and comprehensive integration of data mining with the data warehouse DBMS this task has to be performed in the database engine as well. In this section we present a strategy to efficiently map *MFSSearch* to database operators. The necessary flexibility will be provided by user-defined functions [SQL99] and user-defined table operators (UDTOs) [JM99]. UDTOs permit the definition of set-oriented operations within the database engine. They operate on one or more tables and possibly some additional scalar parameters and return a table or a tuple. The arguments (i.e. input tables) can be intermediate results of a query, i.e. they are not restricted to base tables only. Thus the *StreamJoin* operator itself can be implemented as a UDTO as well.

In addition, we assume that the candidate generation algorithm is realized as a UDTO as well, called *GenerateCand*. As already mentioned, this algorithm starts with the itemset holding all items. Thus, this is the first itemset produced by the *GenerateCand* operator. Later on, since dynamic pruning is employed, the generation of further candidates depends on the results of processing the current candidate in the search space via the *StreamJoin* operator. This approach obviously forms a cycle in the overall MFS generation scheme, as depicted in Fig. 4.

As the *GenerateCand* UDTO is incorporated within a cycle, candidate and result generation must be split up. In Fig. 4, the functionality of the *StreamJoin* operator has already been expanded to calculate also the aggregation on *itemset* as explained in Section 4, thus returning the support of each itemset as well as of all its prefixes. The resulting output stream is called

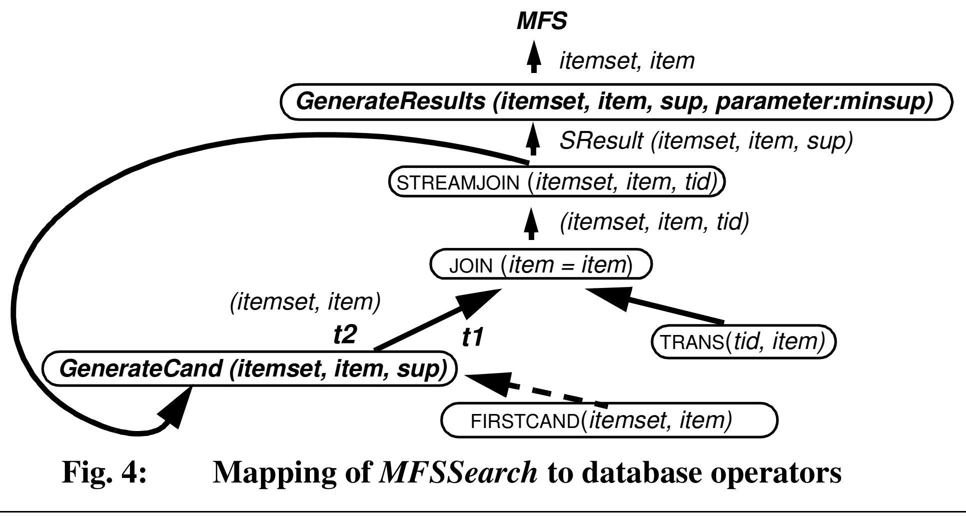


Fig. 4: Mapping of *MFSSearch* to database operators

SResult(itemset, item, sup).

This output stream is consumed by two operators: the *GenerateCand* UDTO and the *GenerateResults* UDF. The *GenerateCand* UDTO is only responsible for candidate generation. For initialization purposes, the first candidate, namely the itemset incorporating all items, is initially read from the table FIRSTCAND(*itemset, item*) and transmitted unchanged to the subsequent operators *Join*, respectively *StreamJoin*. These calculate the corresponding (prefix) supports as already presented before.

In all subsequent iterations, the input of the *GenerateCand* UDTO is provided by the output of *StreamJoin*, i.e. the *SResult* data stream. This intermediate result is used by *GenerateCand* to perform pruning as presented in the previous section and to further explore the search space. The resulting subsequent candidate itemsets are added to the output stream, thus starting new iterations. The process continues until no further candidate itemsets are available, i.e. the entire MFS is calculated. The required grouping for the *StreamJoin* operator can be accomplished by an index nested-loops join, TRANS being the inner table. Using an (usually already existing) index on the *item* attribute we can also circumvent repeated scans of the TRANS table.

The functionality for generating the final result is taken over by the *GenerateResults* UDF. This gets as input the result of the *StreamJoin* operator in the form *(itemset, item, sup)*. The minimal support defined by the user is provided by means of a scalar parameter *minsup*. Thus, *GenerateResults* selects frequent itemsets performing a filtering functionality. As already explained, by employing the *MFSSearch* candidate generation, a frequent itemset found is also a MFI. Thus, *GenerateResults* can immediately add it to the output stream. This results in fast response times and continuous input for further processing by e.g. association rule generation.

In the following, we focus on how the QEP from Fig. 4 can be expressed in (augmented) SQL. As already mentioned, *GenerateResults* can be realized by a UDF, as currently supported by most database vendors. However, UDFs can not be used for the *StreamJoin* and *GenerateCand* approaches, since they both deliver sets of tuples. This problem of expressing set-orientation can be solved by UDTOs [JM99], as presented in the following. As for the cycle within the QEP, this can be resolved in a similar way as recursion [SQL99], using e.g. common table expressions [Ch96].

By adequately using the above mentioned concepts, we obtain a single statement, as depicted in Fig. 5. Hereby, we have used a simplified syntax for a better understanding. The common table expression corresponds to the *SResult* stream. The first “SELECT *StreamJoin*” clause corre-

```
SET minsup = myminsup;

WITH SResult(itemset, item, sup) AS
((SELECT StreamJoin(itemset, item, tid)
  FROM TRANS,
     (GenerateCand(SELECT itemset, item, -1 FROM FIRSTCAND)) AS t1
  WHERE TRANS.item = t1.item)
UNION ALL
(SELECT StreamJoin(itemset, item, tid)
 FROM TRANS,
     (GenerateCand(SELECT itemset, item, sup FROM SResult)) AS t2
 WHERE TRANS.item = t2.item))

SELECT GenerateResults FROM SResult
```

Fig. 5: SQL representation using common table expressions, UDFs and UDTOs

sponds to the first iteration, where *GenerateCand* receives its input from the FIRSTCAND table, i.e. the itemset containing all items. This first input is used to initialize the search space, hence the value of the *sup* parameter is irrelevant (e.g. *-1* in Fig. 5). After initializing the search space, the *GenerateCand* operator transmits this first candidate unchanged to the *StreamJoin* operator. The corresponding output stream is called *t1* in Fig. 4 and Fig. 5. The subsequent iterations are expressed by the second input of the UNION operator. In this "SELECT *StreamJoin*" clause, the input of *GenerateCand* is already provided by the result of the *StreamJoin* operator, i.e.the *SResult* stream. This input is used to further explore the search space. The subsequent candidate itemset is added to the output stream *t2*, thus starting a new iteration.

In this way, the MFS calculation can be comprehensively expressed in (augmented) SQL. Thus the entire processing scheme or constituting parts of it can be referenced for other mining tasks as well. Please note that in contrast to other approaches [STA98], this strategy avoids intermediate table constructions as well as the formulation of separate SQL statements for each processing phase. Instead, as presented in Fig. 5, the entire MFS calculation can be expressed in a compact way by a single statement, thus query optimization and parallelization can be applied as usual for sake of increasing efficiency.

7 Parallelization Potential

The performance of database operations can be considerably improved by using parallelization techniques [NM98]. In this section we concentrate on the parallelization possibilities of *MFS-Search*. A critical aspect of parallelization is that the strategies work well with the existing physical data partitioning. This is especially important for an efficient integration of data mining with data warehouses. In contrast, most related work [HKK97, AS96] propose solutions that are based on proprietary partitioning strategies or even data replications, rendering these approaches inadequate for large-scale operating databases. In addition, for such applications communication and I/O overhead should be avoided as much as possible. However, most previous parallelization strategies [SK98, SK96, AS96] make repeated passes over the disk-resident database partition, thus incurring high I/O overhead. Moreover, in many cases they exchange their counts of candidates or remote database partitions during each iteration, resulting also in high communication overhead. Additionally, some of the approaches replicate complicated memory structures, thus inefficiently utilizing main memory.

In the following, we present parallelization approaches that maximize data locality, thus reducing communication as well as I/O overhead. Moreover, different kinds of physical disk partitionings of the data warehouse are taken into account.

Assume that the TRANS table in Fig. 4 is the central FACTS table in a data warehouse star schema, holding also other attributes like *customer, time* etc. The CAND table is generated on the fly, according to the current search space status, as presented in the previous section. We differentiate two scenarios w.r.t. possible physical partitionings on disk, as discussed in the following.

7.1 Collocated Transaction Items

According to different application scenarios, the FACTS table can be partitioned in multiple ways [Schn97]. In Fig. 6 we propose a solution which is compatible with a partitioning strategy of the central FACTS table so that all tuples belonging to a single transaction are on the same partition. This is the case for instance if the partitioning is done on *time, tid,* or *customer* attributes.

In this case, each partition can calculate the local supports of the candidate itemsets by using the *StreamJoin* processing scheme. In Fig. 6, the first candidate itemset is *{A, B, C}*. Only the supports of the prefixes need to be communicated to a central merge operator, called *CumulateSupports*, that evaluates the final supports by adding up the local supports of the prefixes. This cumulated result is the input of both the central *GenerateCand* as well as *GenerateResults* operators. As described in Section 6, the *GenerateCand* operator decides on the next candidate itemset. This itemset, e.g. *{A, C}* in Fig. 6, is broadcasted to all participating nodes, thus starting a new iteration. The *GenerateResults* operator produces the final results holding maximal frequent itemsets as already shown in Section 6.

Hence, only candidate itemsets and computed supports need to be communicated over the net-

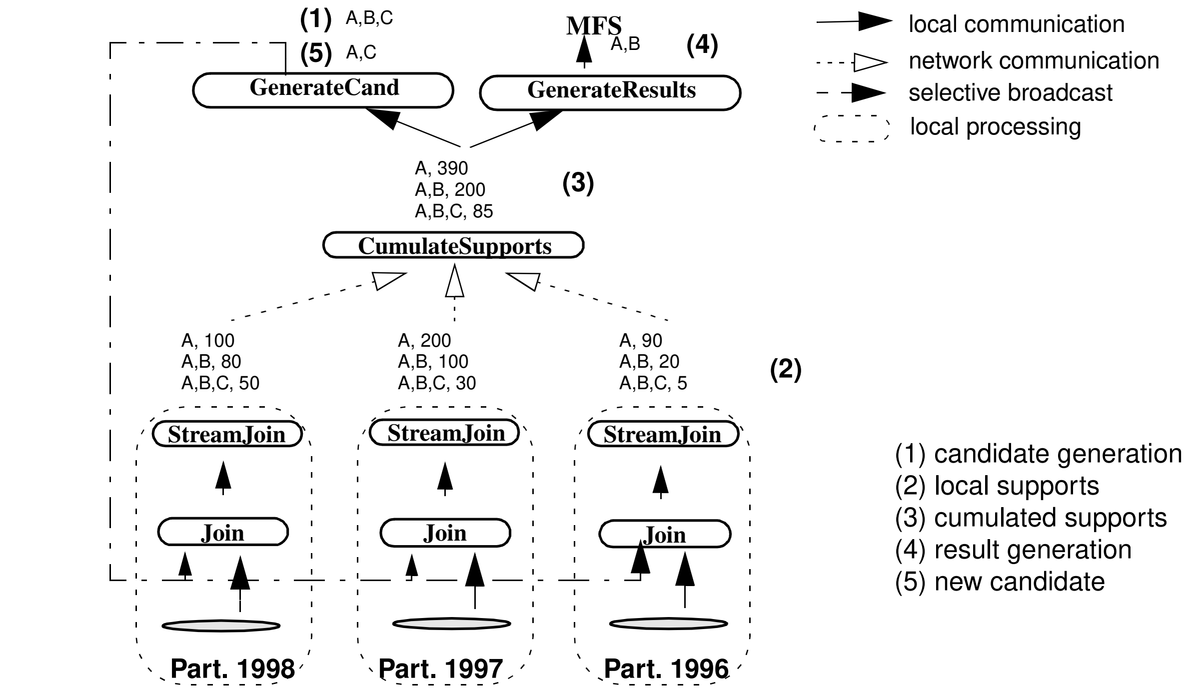


Fig. 6: Parallel processing for the evaluation of maximal frequent itemsets; items belonging to a transaction reside on the same partition

work, producing only minimal communication overhead. In contrast to similar strategies [AS96], performance is improved by avoiding multiple database passes and replicated memory structures.

7.2 Distributed Transaction Items

In the second possible scenario the FACTS table is partitioned in a way that doesn't guarantee that all items belonging to a transaction reside on the same partition. This is the case if the partitioning is done for instance on the *item* attribute.

A small modification of the *StreamJoin* operator allowing it to read streams from different inputs can also prevent from repartitioning. This is shown in Fig. 7, where the data warehouse is partitioned on the *item* attribute. When computing the support for candidate itemset *{A, B, C}*, the *StreamJoin* operator receives its input streams from different nodes, corresponding to the constituting items. Thereby, the *StreamJoin* operator can reside on any of the processing nodes. Please note that the communication overhead is increased by the fact that the *tids* belonging to each item need to be communicated over the network. This can be reduced by executing the *StreamJoin* processing on the node corresponding to the item with the highest support. In this case, the most voluminous *tid*-lists don't have to be communicated over the network.

It is not necessary to broadcast the candidate itemsets to all partitions, either. If the partitioning

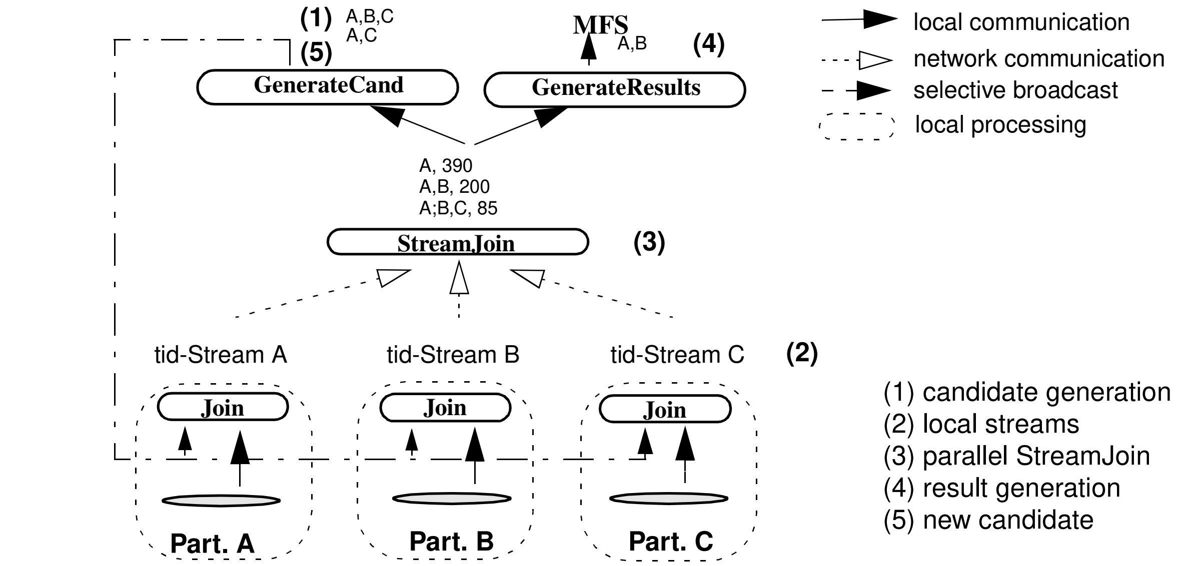


Fig. 7: Parallel processing for the evaluation of maximal frequent itemsets; items belonging to a transaction reside on different partition

function on the *item* attribute is known, a candidate itemset only has to be sent to the partitions that contain that item. For instance, in step (5) from Fig. 7, the new candidate *{A, C}* only has to be sent to partitions *A* and *C*. We do not know of any other parallelization strategy in the data mining area which would incur less communication overhead for this scenario without repartitioning or (selectively) replicating the database [HKK97].

8 Performance Evaluation

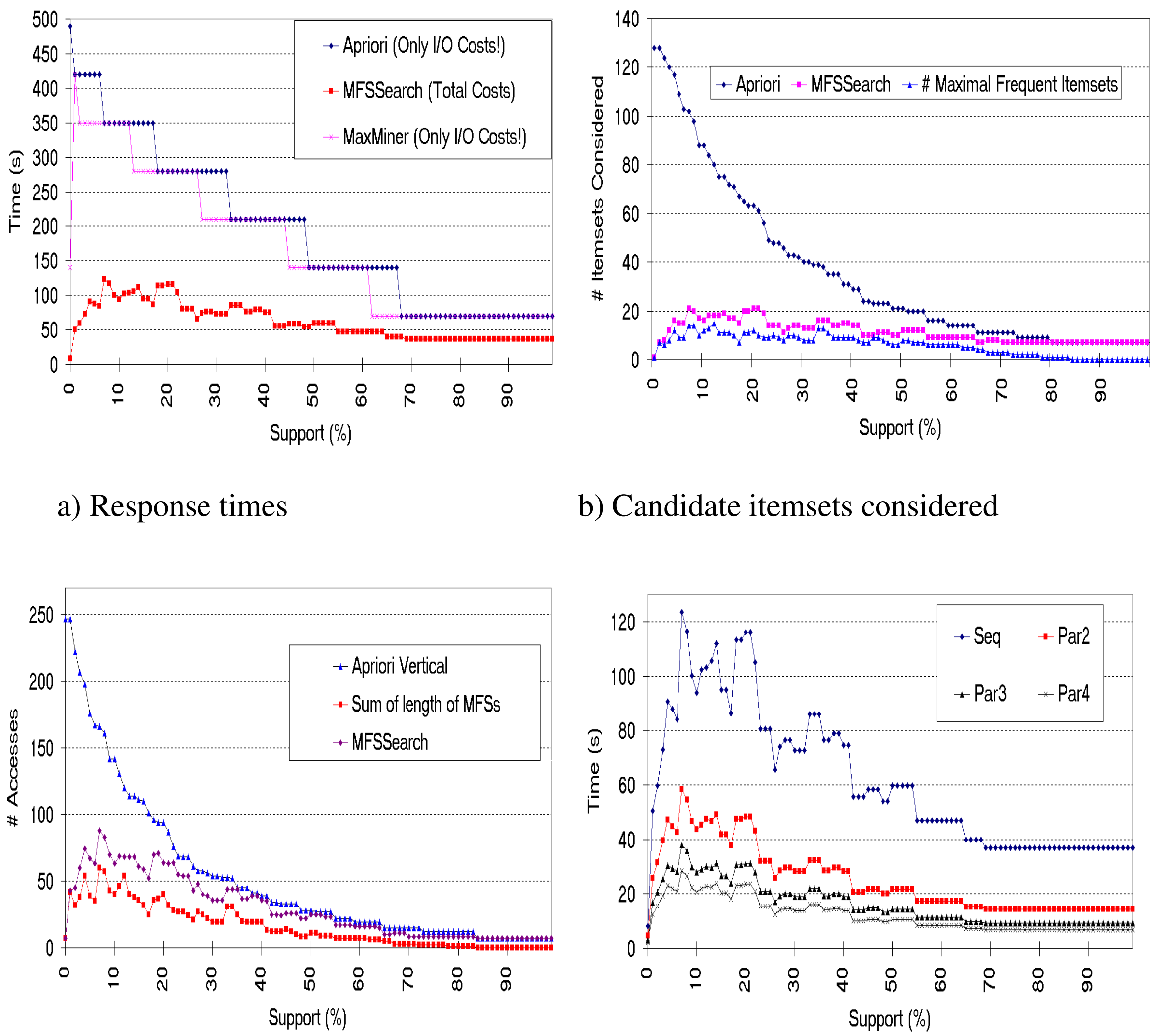
In order to evaluate the performance of our processing scheme *MFSSearch* for maximal frequent itemsets via the *StreamJoin* operator, we have integrated this operator into the MIDAS system. MIDAS [BJ+96] is a prototype of a parallel database system running on a hybrid architecture comprising several SMP nodes combined in a shared-disk manner. We have validated our approach using a 100 MB TPC-D database [TPC95], running on a SUN-ULTRA1 workstation with a 143 MHz Ultra Sparc processor. For the parallel scenarios, we used a cluster of up to 4 workstations. The database contains 150.000 transactions comprising orders on 20.000 different parts. For a detailed evaluation, it was important to consider a column having a limited value domain. Thus, we have performed our measurements on the LINEITEM table, where the place of the *item* column is taken over by *l_linenumber* and the pair *l_partkey, l_suppkey* is considered as being the *tid* attribute. The domain of the *l_linenumber* column is from 1 to 7, and the attributes *l_partkey, l_suppkey* define 67.806 transactions. Due to the uniform data distribution, the length of the MFIs decreases monotonically with increasing supports.

In the following we would like to point out the difference of this modeling to traditional market basket analysis. In a traditional market basket analysis, if an itemset *{A, B}* is found frequent, a possible resulting rule might be: *"If a customer buys item A at a given time it is likely that he/she buys also item B"*. For simplification purposes, assume that in our modeling the *l_linenumber* attribute represents some kind of timestamp: weekdays, months etc. Hence, a possible interpretation of a frequent itemset *{1, 2}* is the following: *"If a part is sold at timestamp 1, it is likely that the same part will be sold at timestamp 2 as well"*. Hence, this kind of modeling is particularly suitable for e.g. event analysis.

In order to compare the performance of *MFSSearch* with the *Apriori* algorithm that is the basis of most bottom-up approaches, we have presented in Fig. 8a the time that is necessary to perform the multiple database scans specific to this algorithm. Please note that this curve doesn't comprise any CPU costs that are also inherent to the *Apriori* algorithm. As can be seen in Fig. 8a, *MFSSearch* shows a performance that is orders of magnitude better than the *Apriori* algorithm. At the same time, we have listed the I/O costs that would result from processing the items using *MaxMiner* [Ba98]. This roughly corresponds also to the I/O necessary for the *PincerSearch* algorithm [LK98]. As can be seen, although both approaches have proven to be more efficient than the *Apriori* algorithm in terms of CPU costs, the repetitive database scans still cause significant I/O costs. The results in [Ba98] also show that the increased efficiency of *MaxMiner* does not result primarily from the reduction of database passes, but from the consideration of less candidates. However, as can be seen in Fig. 8a, the resulting I/O costs of these algorithms already exceed the total costs of *MFSSearch.*

The most striking difference between *Apriori* and *MFSSearch* is in the number of candidate itemsets considered to produce the set of maximal frequent itemsets. As can be seen in Fig. 8b, while for the *MFSSearch* processing scheme this number is proportional to the actual number of maximal frequent itemsets, the itemsets considered by the *Apriori* algorithm increase exponentially with decreasing supports. As shown in [LK98], the number of maximum frequent itemsets is a non-monotone function w.r.t. the minimal support. This result shows that, unlike most algorithms, *MFSSearch* can fully benefit from this property.

As mentioned in Section 2, in [STA98] the most promising improvement of the *Apriori* algorithm that is also suitable for database integration was found to be one based on a *Vertical* format. This format requires a preparatory phase that determines for each item a list comprising all *tids* that contain that item. Because of the variable length of these lists, in [STA98] they are stored in BLOBs. To compare this optimized variant of the *Apriori* algorithm with our approach,



a) Response times

b) Candidate itemsets considered

c) Data accesses

d) Parallelization performance

Fig. 8: Performance evaluation for the evaluation of maximal frequent itemsets

we have kept track of the data accesses necessary in each scenario. In this case, by a single data access we mean the access to all *tids* corresponding to a single item, i.e. in the implementation proposed by [STA98] reading in a single BLOB. In Fig. 8c, we have compared these numbers with the MFS volume[1]. As can be seen, the data accesses related to *MFSSearch* are proportional to this measure. In contrast, the data accesses needed for the *Vertical* approach increased exponentially with decreasing supports. Comparing Fig. 8a and Fig. 8c, we can see that the complexity of *MFSSearch* scales linearly with the MFS volume as well.

As for parallelization, we have partitioned the LINEITEM table on *l_partkey, l_suppkey.* This corresponds to a partitioning in which all tuples belonging to a transaction reside on the same par-

1. Fig. 8c demonstrates once again the non-monotone property of the MFS volume [LK98] w.r.t. the minimum support.

tition (cf. Section 7.1). We have used a degree of parallelism varying from 1 (sequential) to 4. The results presented in Fig. 8d show a linear speedup. Hence *MFSSearch* shows a good parallelization potential resulting in additional performance improvements [NM98].

Finally, we would like to make some general comments on these first experimental results. We are fully aware of the fact that frequent itemset evaluation on a database of this size and item domain can be very efficiently performed using optimized methods that exploit the small size of the database to construct compact data structures [AAP00, HPY00] and thus avoid multiple scans. However, our aim was to test the feasibility of *MFSSearch* as a generic approach integrated within the database engine and compare this with other strategies for the *general* case. Therefore, we haven't made use either of any optimizations, like e.g. caching. Instead, we have only used an index for efficient table access, as well as to achieve the necessary grouping for the *StreamJoin* operator, as suggested in Section 6. Since the measurements show that in this configuration *MFSSearch* still scales linearly with the MFS volume, we are confident that it will show comparable efficiency also for larger databases and item domains.

As already mentioned, in our test database the length of the MFIs decreases monotonically with increasing supports. The performance measurements from Fig. 8 show that *MFSSearch* provides a good performance w.r.t. to all significant measures for both low and high supports, i.e. for both short as well as long MFIs. This demonstrates the effectiveness of our hybrid pruning technique. Thus top-down pruning performs best for long MFIs, where (large) frequent itemsets eliminate several subsequent subsets from exploration. At the same time, bottom-up pruning comes mostly to application in the case of short MFIs, where infrequent subsets reduce the search space by eliminating their corresponding supersets from exploration. The effectiveness of this bottom-up pruning in *MFSSearch* is reinforced by the fact that it employs the *StreamJoin* technique that additionally calculates all prefixes. Hence infrequent prefixes can also very effectively contribute to bottom-up pruning.

As compared to [AAP00, HPY00] that also avoid multiple database scans, we believe that the main advantage of *MFSSearch* is the straightforward way it can be integrated and referenced on the database execution level. In addition, since *MFSSearch* explores the search space gradually, it yields a low memory consumption that avoids materialization even for large databases and item domains, as well as in multi-user environments. This feature cannot be guaranteed by strategies that operate on a condensed representation of the whole search space.

9 Conclusions

In this paper we have presented *MFSSearch*, a processing scheme for the generation of maximal frequent itemsets. *MFSSearch* employs a new operator, called *StreamJoin* as a highly effective strategy for (prefix) itemset support calculations.

The response time and data accesses of the *MFSSearch* algorithm are only proportional to the MFS volume, a measure that reflects the inherent complexity of the frequent itemset calculation problem. In addition, *MFSSearch* is characterized by a low memory consumption. Thus, it is applicable also for (ad-hoc) mining in multi-user DBMSs, such as data warehouses. The algorithm can efficiently be integrated with a database engine yielding a single query execution plan. Thus it is able to make profit of all forms of query execution optimizations, including parallelization. Furthermore, the suitability of *MFSSearch* for database integration is augmented by its non-blocking feature, making it attractive for pipelining, and by the fact that it uses only available index structures of the database.

MFSSearch represents a processing primitive for the calculation of the maximal frequent set, thus facilitating the modularization of similar problems, e.g. pattern recognition, and the reuse of this primitive. Our implementation concept based on user-defined table operators and user-defined functions yields a compact representation on the (SQL) language level. With this, *MFSSearch* technology is made available as a kind of primitive database operation that considerably facilitates its potential for reuse.

As for future work, we are in the process to experimentally validate the performance of *MFSSearch* for large-scale databases and item domains. We further plan to investigate the suitability of the approach for hybrid solutions as well, where the considered item domain is restricted by means of e.g. sampling [FS+98].

Literature

AAP00 R. Agarwal,C. Aggarwal, V. Prasad: A tree projection algorithm for generation of frequent sets, In: Journal of Parallel and Distributed Computing (to appear) 2000.

AM+95 R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. I. Verkamo: Fast Discovery of Association Rules, Advances in Knowledge Discovery and Data Mining, Chapter 12, AAAI/MIT Press, 1995.

AS96 R. Agrawal, J. C. Shafer: Parallel Mining of Association Rules, In: TKDE 8(6): 962-969, 1996.

AY97 C. C. Aggarwal, P. S. Yu: Mining Large Itemsets for Association Rules, TCDE Bull., 21(1), March 1998.

AY98 C. C. Aggarwal, P. S. Yu: Online Generation of Association Rules, In: DE Conf., Orlando, Florida, 1998.

BJ+96 G. Bozas, M. Jaedicke et al.: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project, In: Proceedings of the EUROPAR Conf., 1996.

Ba98 R. Bayardo: Efficiently Mining Long Patterns from Databases, In: Proc. SIGMOD Conf., Seattle, 1998.

BM+97 S. Brin, R. Motwani, J. Ullmann, S. Tsur: Dynamic Itemset Counting and Implication Rules for Market Basket Data, In: Proc. ACM SIGMOD Conf., 1997.

Ch96 D. Chamberlin: Using the New DB2, Morgan Kaufman Publishers, San Francisco, 1996.

Ch98 S. Chaudhuri: Data Mining and Database Systems: Where is the Intersection?, In: Bulletin of the TCDE, 21(1), March 1998.

CK+97 J. Claussen, A. Kemper, G. Moerkotte, K. Peithner: Optimizing Queries with Universal Quantification in Object-Oriented and Object-Relational Databases, In: Proc. VLDB Conf, Athens, Greece, 1997.

FS+98 M. Fang, N. Shivakumar et al: Computing Iceberg Queries Efficiently, Proc. VLDB Conf., New York, 1998.

HF+ J. Han, Y. Fu et al: A Data Mining Query Language for Relational Databases, Proc. SIGMOD Conf, 1996.

HGY98 J. Han, W. Gong, Y. Yin: Mining Segment-Wise Periodic Patterns in Time Related Databases, In: Proc. Intl. Conf. on Knowledge Discovery and Data Mining, New York City, NY, August 1998

HKK97 E.-H. Han, G. Karypis, V. Kumar: Scalable Parallel Data Mining for Association Rules, In: SIGMOD Conference, Tucson, Arizona, 1997.

HPY00 J. Han, J. Pei, Y. Yin: Mining Frequent Patterns without Candidate Generation, In: Proc. SIGMOD Conf., Dallas, 2000 (to appear).

JM99 M. Jaedicke, B. Mitschang: User-Defined Table Operators: Enhancing Extensibility for ORDBMS, Proc. VLDB Conference, Edinburgh, 1999.

LK98 D. Lin, Z. M. Kedem: Pincer-Search: a New Algorithm for Discovering the Maximum Frequent Set, In: Proc EDBT Conf., Valencia, Spain.

MPC96 R. Meo, G. Psaila, S. Ceri: A New SQL-like Operator for Mining Association Rules, In: Proc. VLDB Conf, Mumbai, India, 1996.

MPC98 R. Meo, G. Psaila, S. Ceri: A Tightly-Coupled Architecture for Data Mining, In: DE Conf., Orlando, 1998.

Ni99 C. Nippl: Providing efficient, extensible and adaptive intra-query parallelism for advanced applications, PhD Thesis in preparation, TU München, 1999.

NM98 C. Nippl, B. Mitschang: TOPAZ: a Cost-Based, Rule-Driven, Multi-Phase Parallelizer, Proc. VLDB Conf., New York City,1998.

NRM00 C. Nippl, A. Reiser, B. Mitschang: Conquering the Search Space for the Calculation of the Maximal Frequent Set, TR TU München, 2000, http://www3.informatik.tu-muenchen.de/public/mitarbeiter/nippl.html.

PCY97 J. S. Park, M.S. Chane, P.S. Yu: Using a Hash-Based Method with Transaction Trimming for Mining Association Rules, In: IEEE Trans. on TKDE, 9(5), Sept. 1997.

RBG96 S. Rao, A. Badia, D. v. Gucht: Providing Better Support for a Class of Decision Support Queries, In: Proc. SIGMOD Conf., Montreal, 1996.

SON95 A. Savasare, E. Omiecinski, S. Navathe: An Efficient Algorithm for Mining Association Rules in Large Databases, In: Proc. VLDB Conf., Zurich, 1995.

STA98 S. Sarawagi, S. Thomas, R. Agrawal: Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications, In: Proc. ACM SIGMOD Conf, Seattle, 1998.

Sch97 D. Schneider: The Ins and Outs of Data Warehousing, In: Tutorial on the VLDB Conference, Athens, 1997.

SK98 T. Shintani, M. Kitsuregawa: Parallel Mining Algorithms for Generalized Association Rules with Classification Hierarchy. SIGMOD Conference, Seattle, 1998.

SK96 T. Shintani, M. Kitsuregawa: Hash Based Parallel Algorithms for Mining Association Rules. PDIS 1996.

SQL99 ISO/IEC 9075:1999, Information Technology-Database languages-SQL-Part2, 1999.

TPC97 Transaction Processing Council. TPC Benchmark D, Standard Spec., Rev 1.3, 1997.

ZP+97 M. J. Zaki, S. Parthasarathy, M. Ogihara, W. Li: New Algorithms for Fast Discovery of Association Rules, In: Proc. Intl. Conf. on Knowledge Discovery and Data Mining, Newport Beach, California, 1997.

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

bisher erschienen :

Reihe A

Liste aller erschienenen Berichte von 1990-1994
auf besondere Anforderung

342/01/95 A Hans-Joachim Bungartz: Higher Order Finite Elements on Sparse Grids
342/02/95 A Tao Zhang, Seonglim Kang, Lester R. Lipsky: The Performance of Parallel Computers: Order Statistics and Amdahl's Law
342/03/95 A Lester R. Lipsky, Appie van de Liefvoort: Transformation of the Kronecker Product of Identical Servers to a Reduced Product Space
342/04/95 A Pierre Fiorini, Lester R. Lipsky, Wen-Jung Hsin, Appie van de Liefvoort: Auto-Correlation of Lag-k For Customers Departing From Semi-Markov Processes
342/05/95 A Sascha Hilgenfeldt, Robert Balder, Christoph Zenger: Sparse Grids: Applications to Multi-dimensional Schrödinger Problems
342/06/95 A Maximilian Fuchs: Formal Design of a Model-N Counter
342/07/95 A Hans-Joachim Bungartz, Stefan Schulte: Coupled Problems in Microsystem Technology
342/08/95 A Alexander Pfaffinger: Parallel Communication on Workstation Networks with Complex Topologies
342/09/95 A Ketil Stølen: Assumption/Commitment Rules for Data-flow Networks - with an Emphasis on Completeness
342/10/95 A Ketil Stølen, Max Fuchs: A Formal Method for Hardware/Software Co-Design
342/11/95 A Thomas Schnekenburger: The ALDY Load Distribution System
342/12/95 A Javier Esparza, Stefan Römer, Walter Vogler: An Improvement of McMillan's Unfolding Algorithm
342/13/95 A Stephan Melzer, Javier Esparza: Checking System Properties via Integer Programming
342/14/95 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Point-to-Point Dataflow Networks
342/15/95 A Andrei Kovalyov, Javier Esparza: A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs
342/16/95 A Bernhard Schätz, Katharina Spies: Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik
342/17/95 A Georg Stellner: Using CoCheck on a Network of Workstations
342/18/95 A Arndt Bode, Thomas Ludwig, Vaidy Sunderam, Roland Wismüller: Workshop on PVM, MPI, Tools and Applications
342/19/95 A Thomas Schnekenburger: Integration of Load Distribution into ParMod-C
342/20/95 A Ketil Stølen: Refinement Principles Supporting the Transition from Asynchronous to Synchronous Communication

Reihe A

342/21/95 A	Andreas Listl, Giannis Bozas: Performance Gains Using Subpages for Cache Coherency Control
342/22/95 A	Volker Heun, Ernst W. Mayr: Embedding Graphs with Bounded Treewidth into Optimal Hypercubes
342/23/95 A	Petr Jančar, Javier Esparza: Deciding Finiteness of Petri Nets up to Bisimulation
342/24/95 A	M. Jung, U. Rüde: Implicit Extrapolation Methods for Variable Coefficient Problems
342/01/96 A	Michael Griebel, Tilman Neunhoeffer, Hans Regler: Algebraic Multigrid Methods for the Solution of the Navier-Stokes Equations in Complicated Geometries
342/02/96 A	Thomas Grauschopf, Michael Griebel, Hans Regler: Additive Multilevel-Preconditioners based on Bilinear Interpolation, Matrix Dependent Geometric Coarsening and Algebraic-Multigrid Coarsening for Second Order Elliptic PDEs
342/03/96 A	Volker Heun, Ernst W. Mayr: Optimal Dynamic Edge-Disjoint Embeddings of Complete Binary Trees into Hypercubes
342/04/96 A	Thomas Huckle: Efficient Computation of Sparse Approximate Inverses
342/05/96 A	Thomas Ludwig, Roland Wismüller, Vaidy Sunderam, Arndt Bode: OMIS — On-line Monitoring Interface Specification
342/06/96 A	Ekkart Kindler: A Compositional Partial Order Semantics for Petri Net Components
342/07/96 A	Richard Mayr: Some Results on Basic Parallel Processes
342/08/96 A	Ralph Radermacher, Frank Weimer: INSEL Syntax-Bericht
342/09/96 A	P.P. Spies, C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer, H.-M. Windisch: Sprachkonzepte zur Konstruktion verteilter Systeme
342/10/96 A	Stefan Lamberts, Thomas Ludwig, Christian Röder, Arndt Bode: PFS-Lib – A File System for Parallel Programming Environments
342/11/96 A	Manfred Broy, Gheorghe Ştefănescu: The Algebra of Stream Processing Functions
342/12/96 A	Javier Esparza: Reachability in Live and Safe Free-Choice Petri Nets is NP-complete
342/13/96 A	Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Many-to-Many Data-flow Networks
342/14/96 A	Giannis Bozas, Michael Jaedicke, Andreas Listl, Bernhard Mitschang, Angelika Reiser, Stephan Zimmermann: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project
342/15/96 A	Richard Mayr: A Tableau System for Model Checking Petri Nets with a Fragment of the Linear Time μ -Calculus
342/16/96 A	Ursula Hinkel, Katharina Spies: Anleitung zur Spezifikation von mobilen, dynamischen Focus-Netzen
342/17/96 A	Richard Mayr: Model Checking PA-Processes
342/18/96 A	Michaela Huhn, Peter Niebert, Frank Wallner: Put your Model Checker on Diet: Verification on Local States
342/01/97 A	Tobias Müller, Stefan Lamberts, Ursula Maier, Georg Stellner: Evaluierung der Leistungsfähigkeit eines ATM-Netzes mit parallelen Programmierbibliotheken

Reihe A

342/02/97 A Hans-Joachim Bungartz and Thomas Dornseifer: Sparse Grids: Recent Developments for Elliptic Partial Differential Equations

342/03/97 A Bernhard Mitschang: Technologie für Parallele Datenbanken - Bericht zum Workshop

342/04/97 A nicht erschienen

342/05/97 A Hans-Joachim Bungartz, Ralf Ebner, Stefan Schulte: Hierarchische Basen zur effizienten Kopplung substrukturierter Probleme der Strukturmechanik

342/06/97 A Hans-Joachim Bungartz, Anton Frank, Florian Meier, Tilman Neunhoeffer, Stefan Schulte: Fluid Structure Interaction: 3D Numerical Simulation and Visualization of a Micropump

342/07/97 A Javier Esparza, Stephan Melzer: Model Checking LTL using Constraint Programming

342/08/97 A Niels Reimer: Untersuchung von Strategien für verteiltes Last- und Ressourcenmanagement

342/09/97 A Markus Pizka: Design and Implementation of the GNU INSEL-Compiler gic

342/10/97 A Manfred Broy, Franz Regensburger, Bernhard Schätz, Katharina Spies: The Steamboiler Specification - A Case Study in Focus

342/11/97 A Christine Röckl: How to Make Substitution Preserve Strong Bisimilarity

342/12/97 A Christian B. Czech: Architektur und Konzept des Dycos-Kerns

342/13/97 A Jan Philipps, Alexander Schmidt: Traffic Flow by Data Flow

342/14/97 A Norbert Fröhlich, Rolf Schlagenhaft, Josef Fleischmann: Partitioning VLSI-Circuits for Parallel Simulation on Transistor Level

342/15/97 A Frank Weimer: DaViT: Ein System zur interaktiven Ausführung und zur Visualisierung von INSEL-Programmen

342/16/97 A Niels Reimer, Jürgen Rudolph, Katharina Spies: Von FOCUS nach INSEL - Eine Aufzugssteuerung

342/17/97 A Radu Grosu, Ketil Stølen, Manfred Broy: A Denotational Model for Mobile Point-to-Point Data-flow Networks with Channel Sharing

342/18/97 A Christian Röder, Georg Stellner: Design of Load Management for Parallel Applications in Networks of Heterogenous Workstations

342/19/97 A Frank Wallner: Model Checking LTL Using Net Unfoldings

342/20/97 A Andreas Wolf, Andreas Kmoch: Einsatz eines automatischen Theorembeweisers in einer taktikgesteuerten Beweisumgebung zur Lösung eines Beispiels aus der Hardware-Verifikation – Fallstudie –

342/21/97 A Andreas Wolf, Marc Fuchs: Cooperative Parallel Automated Theorem Proving

342/22/97 A T. Ludwig, R. Wismüller, V. Sunderam, A. Bode: OMIS - On-line Monitoring Interface Specification (Version 2.0)

342/23/97 A Stephan Merkel: Verification of Fault Tolerant Algorithms Using PEP

342/24/97 A Manfred Broy, Max Breitling, Bernhard Schätz, Katharina Spies: Summary of Case Studies in Focus - Part II

342/25/97 A Michael Jaedicke, Bernhard Mitschang: A Framework for Parallel Processing of Aggregat and Scalar Functions in Object-Relational DBMS

342/26/97 A Marc Fuchs: Similarity-Based Lemma Generation with Lemma-Delaying Tableau Enumeration

Reihe A

342/27/97 A	Max Breitling: Formalizing and Verifying TimeWarp with FOCUS
342/28/97 A	Peter Jakobi, Andreas Wolf: DBFW: A Simple DataBase FrameWork for the Evaluation and Maintenance of Automated Theorem Prover Data (incl. Documentation)
342/29/97 A	Radu Grosu, Ketil Stølen: Compositional Specification of Mobile Systems
342/01/98 A	A. Bode, A. Ganz, C. Gold, S. Petri, N. Reimer, B. Schiemann, T. Schnekenburger (Herausgeber): "Anwendungsbezogene Lastverteilung", ALV'98
342/02/98 A	Ursula Hinkel: Home Shopping - Die Spezifikation einer Kommunikationsanwendung in Focus
342/03/98 A	Katharina Spies: Eine Methode zur formalen Modellierung von Betriebssystemkonzepten
342/04/98 A	Stefan Bischof, Ernst W. Mayr: On-Line Scheduling of Parallel Jobs with Runtime Restrictions
342/05/98 A	St. Bischof, R. Ebner, Th. Erlebach: Load Balancing for Problems with Good Bisectors and Applications in Finite Element Simulations: Worst-case Analysis and Practical Results
342/06/98 A	Giannis Bozas, Susanne Kober: Logging and Crash Recovery in Shared-Disk Database Systems
342/07/98 A	Markus Pizka: Distributed Virtual Address Space Management in the MoDiS-OS
342/08/98 A	Niels Reimer: Strategien für ein verteiltes Last- und Ressourcenmanagement
342/09/98 A	Javier Esparza, Editor: Proceedings of INFINITY'98
342/10/98 A	Richard Mayr: Lossy Counter Machines
342/11/98 A	Thomas Huckle: Matrix Multilevel Methods and Preconditioning
342/12/98 A	Thomas Huckle: Approximate Sparsity Patterns for the Inverse of a Matrix and Preconditioning
342/13/98 A	Antonin Kucera, Richard Mayr: Weak Bisimilarity with Infinite-State Systems can be Decided in Polynomial Time
342/01/99 A	Antonin Kucera, Richard Mayr: Simulation Preorder on Simple Process Algebras
342/02/99 A	Johann Schumann, Max Breitling: Formalisierung und Beweis einer Verfeinerung aus FOCUS mit automatischen Theorembeweisern – Fallstudie –
342/03/99 A	M. Bader, M. Schimper, Chr. Zenger: Hierarchical Bases for the Indefinite Helmholtz Equation
342/04/99 A	Frank Strobl, Alexander Wisspeintner: Specification of an Elevator Control System
342/05/99 A	Ralf Ebner, Thomas Erlebach, Andreas Ganz, Claudia Gold, Clemens Harlfinger, Roland Wismüller: A Framework for Recording and Visualizing Event Traces in Parallel Systems with Load Balancing
342/06/99 A	Michael Jaedicke, Bernhard Mitschang: The Multi-Operator Method: Integrating Algorithms for the Efficient and Parallel Evaluation of User-Defined Predicates into ORDBMS

Reihe A

342/07/99 A	Max Breitling, Jan Philipps: Black Box Views of State Machines
342/08/99 A	Clara Nippl, Stephan Zimmermann, Bernhard Mitschang: Design, Implementation and Evaluation of Data Rivers for Efficient Intra-Query Parallelism
342/09/99 A	Robert Sandner, Michael Mauderer: Integrierte Beschreibung automatisierter Produktionsanlagen - eine Evaluierung praxisnaher Beschreibungstechniken
342/10/99 A	Alexander Sabbah, Robert Sandner: Evaluation of Petri Net and Automata Based Description Techniques: An Industrial Case Study
342/01/00 A	Javier Esparza, David Hansel, Peter Rossmanith, Stefan Schwoon: Efficient Algorithm for Model Checking Pushdown Systems
342/02/00 A	Barbara König: Hypergraph Construction and Its Application to the Compositional Modelling of Concurrency
342/03/00 A	Max Breitling and Jan Philipps: Verification Diagrams for Dataflow Properties
342/04/00 A	Günther Rackl: Monitoring Globus Components with MIMO
342/05/00 A	Barbara König: Analysing Input/Output Capabilities of Mobile Processes with a Generic Type System
342/06/00 A	Michael Bader, Christoph Zenger: A Parallel Solver for Convection Diffusion Equations based on Nested Dissection with Incomplete Elimination
342/07/00 A	Clara Nippl, Angelika Reiser, Bernhard Mitschang: Extending Database Functionality to Support Frequent Itemset Processing
342/08/00 A	Clara Nippl, Angelika Reiser, Bernhard Mitschang: Conquering the Search Space for the Calculation of the Maximal Frequent Set

SFB 342 : Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

Reihe B

342/1/90 B	Wolfgang Reisig: Petri Nets and Algebraic Specifications
342/2/90 B	Jörg Desel: On Abstraction of Nets
342/3/90 B	Jörg Desel: Reduction and Design of Well-behaved Free-choice Systems
342/4/90 B	Franz Abstreiter, Michael Friedrich, Hans-Jürgen Plewan: Das Werkzeug runtime zur Beobachtung verteilter und paralleler Programme
342/1/91 B	Barbara Paech: Concurrency as a Modality
342/2/91 B	Birgit Kandler, Markus Pawlowski: SAM: Eine Sortier-Toolbox – Anwenderbeschreibung
342/3/91 B	Erwin Loibl, Hans Obermaier, Markus Pawlowski: 2. Workshop über Parallelisierung von Datenbanksystemen
342/4/91 B	Werner Pohlmann: A Limitation of Distributed Simulation Methods
342/5/91 B	Dominik Gomm, Ekkart Kindler: A Weakly Coherent Virtually Shared Memory Scheme: Formal Specification and Analysis
342/6/91 B	Dominik Gomm, Ekkart Kindler: Causality Based Specification and Correctness Proof of a Virtually Shared Memory Scheme
342/7/91 B	W. Reisig: Concurrent Temporal Logic
342/1/92 B	Malte Grosse, Christian B. Suttner: A Parallel Algorithm for Set-of-Support Christian B. Suttner: Parallel Computation of Multiple Sets-of-Support
342/2/92 B	Arndt Bode, Hartmut Wedekind: Parallelrechner: Theorie, Hardware, Software, Anwendungen
342/1/93 B	Max Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung
342/2/93 B	Ekkart Kindler: Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick
342/1/94 B	Andreas Listl, Thomas Schnekenburger, Michael Friedrich: Zum Entwurf eines Prototypen für MIDAS