

# Data Mining with the SAP NetWeaver BI Accelerator

Thomas Legler  
Dresden University of  
Technology  
Database Technology Group  
01307 Dresden, Germany  
t.legler@sap.com

Wolfgang Lehner  
Dresden University of  
Technology  
Database Technology Group  
01307 Dresden, Germany  
lehner@inf.tu-dresden.de

Andrew Ross  
PTU NetWeaver AS TREX  
SAP AG  
Dietmar-Hopp-Allee 16  
69190 Walldorf, Germany  
a.ross@sap.com

## ABSTRACT

The new SAP NetWeaver Business Intelligence accelerator is an engine that supports online analytical processing. It performs aggregation in memory and in query runtime over large volumes of structured data. This paper first briefly describes the accelerator and its main architectural features, and cites test results that indicate its power. Then it describes in detail how the accelerator may be used for data mining. The accelerator can perform data mining in the same large repositories of data and using the same compact index structures that it uses for analytical processing. A first such implementation of data mining is described and the results of a performance evaluation are presented. Association rule mining in a distributed architecture was implemented with a variant of the BUC iceberg cubing algorithm. Test results suggest that useful online mining should be possible with wait times of less than 60 seconds on business data that has not been preprocessed.

## 1. INTRODUCTION

The SAP NetWeaver Business Intelligence (BI) accelerator is a new appliance for accessing structured data held in relational databases. Compared with most previous approaches, the appliance offers an order of magnitude improvement in speed and flexibility of access to the data. This improvement is significant in business contexts where existing approaches impose quantifiable costs. By leveraging the falling prices of hardware relative to other factors, the new appliance offers business benefits that more than balance its procurement cost.

The BI accelerator was developed for deployment in the IT landscape of any company that stores large and growing volumes of business data in a standard relational database. Currently, most approaches to accessing the data held in such databases confront IT staff with a maintenance challenge. Administrators are required to study user behavior to identify frequently asked queries, then build materialized

views and database indexes for those queries to boost their performance. This is skilled work and hence a major cost driver. At best, response times are improved for the identified queries, so good judgement is required to satisfy users. To keep this work within bounds, access to the data is often restricted by means of predefined reports for selected users. In some SAP customer scenarios, the space occupied by data for materialized views is up to three times that required for the original data from which the views are built.

The new accelerator is based on search engine technology and offers comparable performance for both frequently and infrequently asked queries. This enables companies to relax previous technically motivated restrictions on data access. For any query against data held in an SAP NetWeaver BI structure called an InfoCube, the results are computed using a single compact index structure for that cube. In consequence, users no longer find that some queries are fast because they hit a materialized view whereas others are slow. The accelerator makes use of this index structure to determine the relevant join paths and aggregate the result data for the query, and moreover to do so in query run time and in memory. In practical terms, the result is that with the help of the accelerator, users can select and display aggregated data, slice it and dice it, drill down anywhere for details, and expect exact responses in seconds, however unusual their request, even over cubes containing billions of records and occupying terabyte volumes.

From the standpoint of company decision making, the key benefits of SAP NetWeaver BI with the accelerator are speed, flexibility, and low cost. Average response times for the queries generated in typical business scenarios are ten times shorter than traditional approaches, and some times up to a thousand times shorter. Because the accelerator generally performs aggregation anew for each individual query, administrative overhead is greatly reduced and there is no technical need to restrict user freedom. As for cost, state-of-the-art hardware can be scaled exactly to customer requirements. Current trends in hardware prices suggest that the accelerator's total cost of ownership (TCO) advantage over manual tuning approaches will likely grow in future.

The rest of this paper is organized as follows. Sections 2 and 3 present the SAP NetWeaver BI accelerator in sufficient detail for the reader to understand its applicability for data mining: section 2 describes the basic architecture of the BI accelerator and outlines its main technical features,

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '06, September 12-15, 2006, Seoul, Korea.

Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09

and section 3 presents the results of a performance test comparing the BI accelerator with an Oracle database. Section 4 describes the data mining implementation in detail and presents results that indicate its promise. Section 5 concludes with an outlook on future developments.

## 2. BASIC ARCHITECTURE

The main architectural features of the BI accelerator are as follows:

- Distributed and parallel processing for scalability and robustness
- In-memory processing to eliminate the runtime cost of disk accesses
- Optimized data structures for reads and optimized structures for writes
- Data compression coding to minimize memory usage and I/O overhead

The following subsections describe these features more fully.

### 2.1 Scalable Multiserver Architecture

The BI accelerator runs on commodity blade servers that can be procured and installed as required to handle increasing volumes. For example, a moderately large instance of the accelerator may run on 16 blades, each with two processors and 8 gigabytes (GB) of main memory, so the accelerator has 128 GB of RAM to handle user requests. The accelerator index structures are compressed by a factor of well over ten relative to the database tables from which they are generated, so 128 GB is enough to handle most of the reports running in most commercial SAP NetWeaver BI systems. If more memory space is needed in runtime, the accelerator swaps out unused attributes or indexes via LRU for requested data. To increase space without swapping, either more RAM can be installed in 8 GB increments on additional blades or next-generation blades with more than 8 GB per blade can be procured.

The use of scalable and distributed search technology enables investment in hardware and network resources to be optimized on an ongoing basis to reflect changing availability requirements and load levels (see figure 1). In each landscape, a name server maintains a list of active services, pings them regularly and switches to backups where necessary, and balances load over active services. The first customers will implement the BI accelerator on preconfigured hardware that can be plugged into their existing landscape. It is expected that future customers will have the option to configure new hardware dynamically. If the capacity is available, the accelerator will then be able to scale rapidly to suit changing load. In an adaptive landscape, accelerator instances will be replicated as required, by cloning services, and they will form groups with master and backup servers and additional cohort name servers for scalability. The master index servers will handle both indexing and query load. Such groups can be optimized for both high availability and good load balancing, with the overall goal of requiring zero administration.

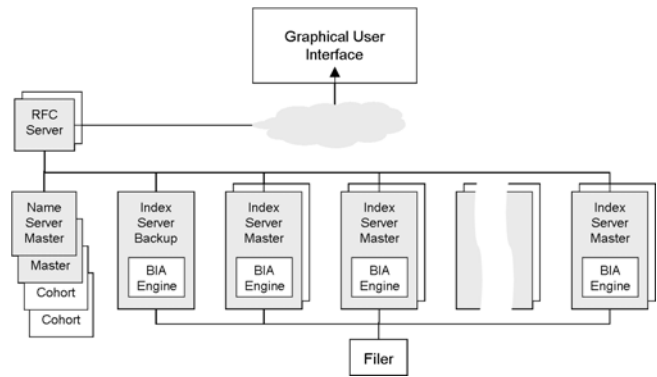


Figure 1: Multiserver architecture for scalability

The accelerator engine can partition large tables horizontally for parallel processing on multiple machines in distributed landscapes (see figure 2). This enables it to handle very large data volumes yet stay within the limits of installed memory. The engine splits such large volumes over multiple hosts, by a round-robin procedure to build up parts of equal size, so that they can be processed in parallel. A logical index server distributes join operations over partial indexes and merges the partial results returned from them. This scalability enables the accelerator to run on advanced computing infrastructures, such as blade servers over which load can be redistributed dynamically. The accelerator is designed to run on 64-bit platforms, which work within a 16 EB address space (instead of a 4 GB space for 32-bit platforms).

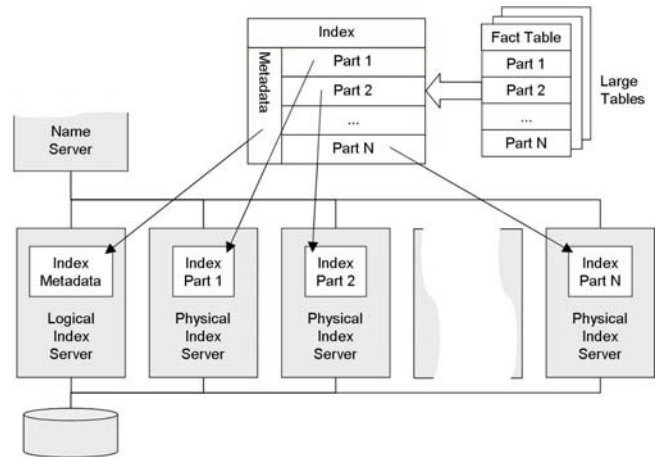


Figure 2: Horizontal partitioning of indexes

### 2.2 Processing in Memory

The BI accelerator works entirely in memory (see figure 3). Before a query is answered, all the data needed to answer it is copied to memory, where it stays until the space is needed for other data. To calculate result sets for analytical queries, the accelerator engine runs highly optimized algorithms that avoid disk accesses. The engine takes the query entered by the user, computes a plan for answering it, joins

the relevant column indexes to create a join path from each view attribute to the BI fact table, performs the required aggregations, and merges the results for return to the user. This ability to aggregate during runtime without additional precalculations is critical for realizing the accelerator's flexibility and TCO benefits. It frees IT staff from the need to prebuild aggregates or realign them regularly, and users benefit from predictable response times.

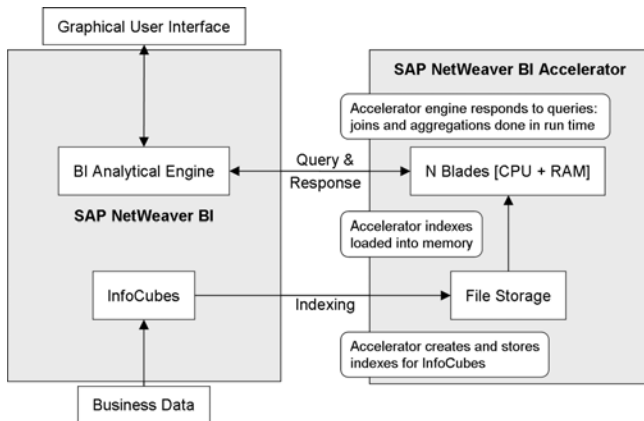


Figure 3: Query processing in memory

The accelerator engine decomposes table data vertically, into columns, that are stored separately (see figure 4). This columnwise storage is similar to the techniques used by C-Store [13] and MonetDB [4]. It makes more efficient use of memory space than row-based storage, since the engine needs to load only the data for relevant attributes or characteristics into memory. This is appropriate for analytics, where most users want to see only a selection of data and attributes. In a conventional database, all the data in the table is loaded together, in complete rows, whereas the new engine touches only relevant data columns. The engine can also sort the columns individually to bring specific entries to the top. The column indexes are written to memory and cached as flat files. This concept improves efficiency in two ways: both the memory footprint of the data and the I/O flows between CPUs, memory and filer are smaller.

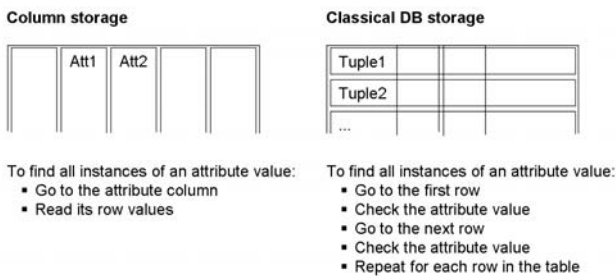


Figure 4: Columnwise decomposition of tables

## 2.3 Optimized Data Structures

The BI accelerator indexes tables of relational data to create ordered lists, using integer coding, with dictionaries for value lookup. These index structures are optimized not only for read access in very large datasets but also more specifically to work efficiently with proprietary SAP NetWeaver data structures called BI InfoCubes. A BI InfoCube is an extended star schema for representing structured data (see figure 5). A large fact table is surrounded by dimension tables (D) and sometimes also X and Y tables, where X tables store time-independent data and Y tables time-dependent data. Peripheral S tables spell out the values of the integer IDs used in the other tables.

The BI accelerator metamodel represents an InfoCube logically as a join graph, where joins between the tables forming the star schema are predefined in the metamodel and materialized at run time by the accelerator engine. In effect, the metamodel bridges the gap between the structured data cubes and search engine technology, which was originally developed to work with unstructured data. Currently, the accelerator answers queries without the use of special optimization strategies, or cached or precalculated data, although the attached BI system implements a caching strategy for responding to repeated queries.

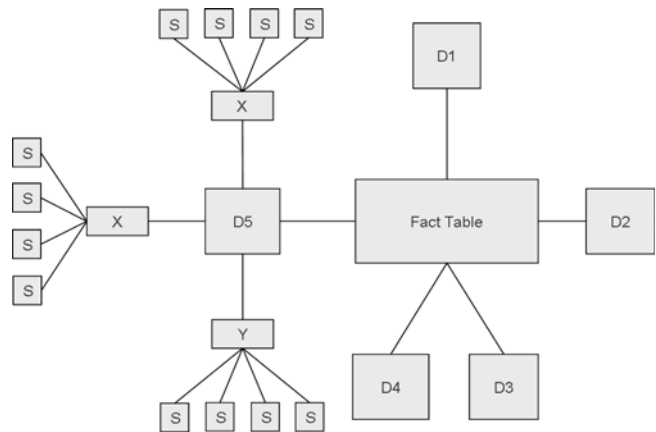


Figure 5: SAP NetWeaver BI InfoCube extended star schema

The SAP InfoCubes that store business data for reporting normally receive only infrequent updates and inserts. The accelerator index structures were designed to support such read-only or infrequent-write scenarios for BI reporting. But the accelerator has also been designed to handle updates efficiently in the following two respects:

- Updates to an InfoCube are visible with minimal delay in the accelerator for query processing, and
- Accumulated updates do not degrade response times significantly.

To handle updates to the accelerator index structures, the accelerator uses a delta index mechanism that stores changes to an index structure in a delta index, which resides next to

the main index in memory. The delta index structure is optimized for fast writes and is small enough for fast searches. The engine performs searches within an index structure in parallel over both the main index and the delta index, and merges the results. From time to time, to prevent the delta index from growing too large, it is merged with the main index. The merge is performed as a background job that does not block searches or inserts, so response times are not degraded.

In case of problems involving loss or corruption of data, the accelerator is able to detect correctness violations and invalidate the affected relations. To restore corrupted data it needs to recreate the relevant index structures from the original InfoCubes stored on the database.

## 2.4 Data Compression Using Integers

As a consequence of its search engine background, the BI accelerator uses some well known search engine techniques, including some powerful techniques for data compression, for example as described in [15]. These techniques enable the accelerator to perform fast read and search operations on mass data yet remain within the constraints imposed by installed memory.

Data for the accelerator is compressed using integer coding and dictionary lookup. Integers represent the text or other values in table cells, and the dictionaries are used to replace integers by their values during post-processing. In particular, each record in a table has a document ID, and each value of a characteristic or an attribute in a record has a value ID, so an index for an attribute is simply an ordered list of value IDs paired with sets of IDs for the records containing the corresponding values (see figure 6). To compress the data further, a variety of methods are employed, including difference and Golomb coding, front-coded blocks, and others. Such compression greatly reduces the average volumes of processed and cached data, which allows more efficient numerical processing and smart caching strategies. Altogether, data volumes and flows are reduced by an average factor of ten. The overall result of this reduction is to improve the utilization of memory space and to reduce I/O within the accelerator.

Attribute Table		Dictionary		Index	
DocId	ValueId	ValueId	Value	ValueId	DocIdList
1	24	1	IBM	1	...
2	3	2	Microsoft	2	...
3	7	...	...	3	2, 5
4	17	17	SAP	4	...
5	3	...	...	...	...
6	...	...	...	17	4

Figure 6: Data compression using integers

## 3. PERFORMANCE

To give a more tangible indication of the power of the BI accelerator, this section presents the results of a performance test comparing it with an Oracle 9 database. The accelerator

hardware consisted of six blades, each equipped with two 64-bit Intel Xeon 3.6 GHz processors and 8 GB of RAM, running under 64-bit Suse Linux Enterprise Server 9. The database hardware was a Sun Fire V440, with four CPUs and 16 GB of memory, running under SunSoft Solaris 8.

In terms of CPU power and memory size, this is an unequal comparison, but it is worth emphasizing that the two systems tested here are indeed comparable in economic terms, which is to say in terms of the sum of initial and ongoing costs. The accelerator runs on multiple commercial, off-the-shelf (COTS) blade servers mounted in a rack and paired with a commodity database as file server, where the combination is marketed as a potentially standalone appliance with a competitive price. By contrast, an SAP NetWeaver BI installation without the accelerator requires a high-performance big-box database server, with both a high initial procurement cost and high ongoing costs for administration and maintenance. For business purposes, the proper comparison to make is between alternatives with similar TCO, not with similar CPU power and memory size.

The data and queries for the performance test were copied (with permission) from real customers. The data consisted of nine SAP InfoCubes with a total of 850 million rows in the fact tables and 22 aggregates (materialized views) for use by the database. This added up to over 130 GB of raw InfoCube data plus 6 GB of data for aggregates in the case of the database and a total of approximately 30 GB for the data structures used in the accelerator.

The results do not support exact predictions about what potential users of the BI accelerator should expect. In SAP experience, companies with different data structures and different usage scenarios can realize radically different performance improvements when implementing new solutions, and the BI accelerator is likely to be no exception. Nevertheless, the query runtimes cited here may be interpreted to indicate the order of magnitude of the performance a prospective user of the accelerator can reasonably expect.

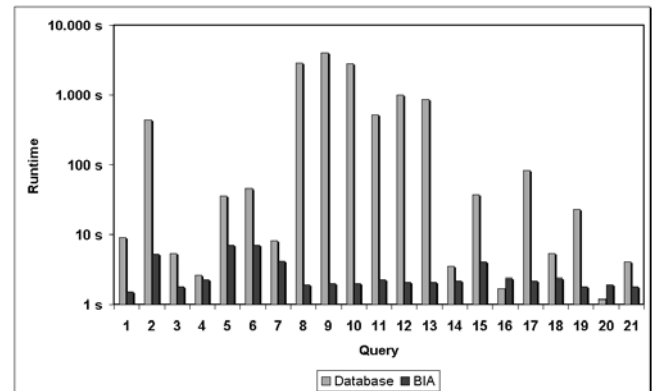


Figure 7: Performance comparison

Figure 7 shows the main results. It shows that the accelerator runtime was relatively steady for very different queries. Most of the queries tested ran in (much) less than five sec-

onds. This behavior has been confirmed in early experience with BI accelerators installed at customer sites. Together, these results may encourage an accelerator user to run any query with reasonable confidence that excessively long response times will not occur.

## 4. DATA MINING

As shown in such papers as [14], many business organizations regard the mining of existing repositories of mass data to be an important decision support tool. Given this fact and following the ideas in [11], the BI accelerator development team has implemented some common data mining technologies, such as association rule mining, time series and outlier analysis, and some statistical algorithms. Because the accelerator holds all relevant data in main memory for reporting, data mining can be enabled without further replication of data and without the need to build special new structures for mining. The implementation is intended to enable BI accelerator users to perform data mining without requiring any additional hardware resources, using the architecture described above (section 2) and with minimal additional memory consumption. To achieve this goal, there were two main challenges to overcome:

- To reuse the existing accelerator infrastructure and data structures for data mining, and
- To implement data mining efficiently enough to achieve acceptable wait times.

This section briefly describes our implementation of association rule mining. The algorithm is designed to mine for interesting rules inside the SAP InfoCube extended star schema, in particular its fact and dimension tables. The algorithm examines the attributes of the fact table and searches for rules like: “If a customer lives in Europe (dimension: location) and sells cars (dimension: product), the customer is likely to use SAP software (dimension: ERP software)”. This knowledge can be exploited, for example, to offer specific product bundles to targeted customers on the basis of their inferred interests. The principles of how to implement association rule mining are well known. To find such association rules, it is necessary first to discover frequent itemsets and then to generate rules from this information.

Several algorithms for association rule mining on single and multi-core machines are known (see, e.g., the papers [2, 7, 12]). These algorithms are able to handle arbitrary itemsets of unrestricted size. For an SAP InfoCube fact table, it is only necessary to handle a fixed number of dimensions or attributes, which is defined by the user and bounded by the number of columns of a relation. Given knowledge of the maximum size of an itemset, one can simplify the mining strategy to iceberg cubing. An overview of common association rule mining algorithms can be found in [16].

One of the basic strengths of the accelerator is that it runs in a distributed landscape. To handle association rule mining in a distributed architecture, we implemented a two-step algorithm:

1. A variant of BUC iceberg cubing introduced in [3] mines for frequent itemsets in every partition of a distributed relation.

2. A PARTITION-like consolidation of local results to a global one proposed in [12] responds to user requests.

In subsection 4.1, we explain our mining algorithm on unpartitioned relations. In subsection 4.2, we describe how we use local mining results on multiple partitions to resolve the global result on distributed relations.

We use the terminology introduced in [1] for association rule mining. A row of a relation  $R$  can be stated as set  $L = i_1, i_2, \dots, i_n$  where  $i$  is an item (a column entry) and  $n$  the number of columns in  $R$ .  $|R|$  represent the cardinality of  $R$ . The number of occurrences of any  $X \subset L$  is called  $support(X)$ . Every  $X$  with  $support(X) \geq minSupport$ , where  $minSupport$  is a user-defined threshold level, is called a frequent itemset. In distributed landscapes, in addition to the global  $minSupport$ , a local  $minSupport$  for  $X$  is defined.  $LocalminSupport$  for part  $n$  of relation  $R$  is defined as

$$localMinSup_n(R) = globalMinSup * \frac{|R_n|}{|R|}$$

For rule generation, we use *confidence* defined as

$$confidence(IP \subset I, I) = \frac{support(IP)}{support(I)}$$

A generated rule with high confidence is probably more interesting and significant for the user than rules with lower values. Similarly to  $minSupport$ , we use a threshold level for interesting rules, called minimum confidence  $minConfidence$ . The optimum value for reasonable rules depends on the data characteristics.

### 4.1 Frequent Itemset Mining with BUC

We start with the simple case of unpartitioned relations and describe the mining algorithm on undistributed data.

The SAP NetWeaver BI accelerator engine has many of the features proposed by [6] to implement efficient data mining. These features include:

1. A very low system level,
2. Simple and fast in-memory, array-like data structures,
3. Dictionaries to map strings to integers,
4. Distribution of data to handle large datasets in memory.

Because of these features, the accelerator is an appropriate infrastructure for association rule mining.

We adopted BUC as the frequent itemset mining algorithm because it suits the internal data structures used by the accelerator. The accelerator uses column-based relations and very simple data structures, so it can achieve fast direct access to arbitrary rows of an attribute, which is a significant part of our BUC implementation.

The BUC algorithm works as follows. As with apriori-based association rule mining techniques (described in [2, 9, 10]), an initial scan for itemsets with one element is needed. To resolve the counts, we perform one internal scan per attribute. After filtering the attribute values with minimum

support, we use the row numbers of these values to resolve support for all itemsets with size two. Their row IDs are used to find itemsets of size three, and so on.

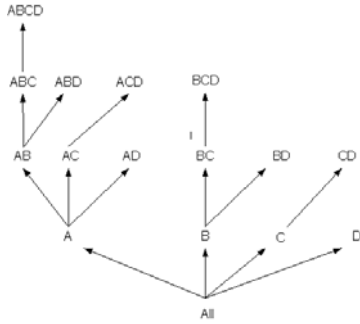


Figure 8: Processing tree for BUC

In contrast to apriori-based algorithms, BUC works depth-first. BUC generates an itemset  $(i_1, i_2, i_3, \dots, i_n)$  of size  $n$  out of itemset  $(i_1, i_2, i_3, \dots, i_{n-1})$ . Figure 8 shows this procedure. Itemsets are generated bottom-up, beginning on the left branch of the tree in figure 8. At first, the occurrences of distinct values in  $A, B, C$  and  $D$  are counted in parallel. With the results for attribute  $A$ , the algorithm determines supports for the left branch  $AB, ABC$  and  $ABCD$  in a recursive manner. Based on the row IDs for  $AB$ , a computation of counts for  $ABD$  is easy and fast because we only have to test the rows of  $AB$  for the values of attribute  $D$ . The recursion stops iff the minimum support for an itemset  $I = (i_1, i_2, i_3, \dots, i_n)$  is missed or no more itemsets  $(i_1, i_2, i_3, \dots, i_n, i_{n+1})$  can be generated out of  $I$ . In this way, the whole cube ( $support = 1$ ) or partial cube ( $support > 1$ ) can be computed efficiently inside the accelerator engine using low-level read access.

To optimize this procedure, the attributes are ordered by the number of distinct values. For example, in figure 8,  $A$  is the attribute with the maximum number of distinct values and attribute  $D$  that with the minimum. This heuristic is described in [3] and uses the assumption that for uniformly distributed values the average support counts for single values with minimum support decreases with increasing number of distinct values. In this case, the lower number of occurrences of value  $v_A$  found on  $A$  causes fewer checks on  $B, BC$  and  $BCD$  to count support for itemsets  $(v_A, v_B), (v_A, v_B, v_C)$  and  $(v_A, v_B, v_C, v_D)$ . In general, this reduces the overall number of rows to read. The worst-case scenario for this heuristic is an attribute with many distinct values having very low support but a few values having very high support. This causes a serious reduction of performance.

Other heuristics, such as skew-based ordering, are also possible. We implemented ordering by number of distinct values because this information can be extracted easily and quickly from statistics available in the accelerator engine. But as [3] states, for real datasets only slight differences between these heuristics are observed. Nevertheless, the conditions for an optimum ordering of attributes cannot be guaranteed for real business data, so for future implementations the best ordering strategy may be chosen dynamically.

Algorithm 1 gives an impression of our implementation.

---

**Algorithm 1** Algorithm BUC

---

**Require:**  $minSupport$  and  $A \leftarrow$  list of attributes

---

```

1: function BUC(tuple, rowids, attpos, results)
2: {
3:   for all value  $\in$  attributeattpos do
4:     newRows = [ ]
5:     for all rowid  $\in$  rowids do
6:       if value == attributeattpos[rowid] then
7:         newRows.add(rowid)
8:       end if
9:     end for
10:    support = size(newRows)
11:    if support  $\geq$  minSupport then
12:      newTuple = (tuple + value)
13:      results  $\leftarrow$  (newTuple, support)
14:      if attPos < size(A) then
15:        BUC(newTuple, newRows,
16:            attPos + 1, results)
17:      end if
18:    end if
19:  end for
20: }

21:
22: for all a  $\in$  A do
23:   values, rowids, counts = countSupports(a)
24:   filter(counts  $\geq$  minSupport)
25: end for
26:
27: results = [ ]
28: for all value  $\in$  values do
29:   results  $\leftarrow$  value
30:   BUC(value, rowids, 1, results)
31: end for
32:
33: return results

```

---

The algorithm starts with a list of attribute values with minimum support for every attribute. The FOR statement in line 28 represents the main loop. Line 29 appends the frequent attribute value as frequent itemset  $I$  with  $|I| = 1$  to the result set and line 30 starts the generation of frequent itemsets based on  $I$ .

The BUC function in lines 1 to 20 is used to build itemsets, count their occurrences, and run BUC recursively for the next attribute if support is given.

The function get a base itemset called *tuple*, a list of row IDs for *tuple*, the position of the attribute to check called *attpos*, and a result list to store itemsets with minimum support.

Line 3 is used to determine all values of attribute *attpos* to check if these values occur on rows in *rowids* on line 5. The hits are stored in *newRows*. As a consequence the number of entries in *newRows* is the support for the new itemset  $I = (tuple + value)$ .

If minimum support is given for  $I$  in line 11, it is stored in the result list *results* and, if possible, line 15 initiates the scans for the itemsets based on  $I$ .

## 4.2 Consolidation of Local Results

After the local mining step on all partitions, a consolidation of these local result sets is needed for our implementation. Here we have two possible cases to consider when merging all local itemsets to global itemsets:

1. An itemset was found in all partitions.
2. An itemset was missed in some partitions.

As shown in [12], it is not possible to miss a globally frequent itemset in every partition because all globally frequent itemsets must be locally frequent on at least one partition. So the distributed BUC algorithm delivers complete results. But if the counts for some partitions are missing, the itemset counts can be globally wrong. To ensure the correctness of the algorithm, it is necessary to sum up all local support counts to a global support. Any itemset with a least one missing count can be:

1. Frequent, but potentially with lower support,
2. Not frequent, but  $minSupport$  is reachable with missing partitions, or
3. Not frequent, and  $minSupport$  is not reachable with missing partitions.

In cases 1 and 2, additional searches are needed; in case 3, the itemsets can be dropped.

The next step is to resolve the missing counts. After summation of all available local support counts for an itemset  $I$ , we resolve all missing itemsets  $I_p$  for all partitions  $p \in P$  of a relation  $R$  and count their occurrences on all partitions in parallel. With these results, the final support for globally frequent itemsets can be computed.

Algorithm 2 describes this procedure. Lines 5 to 8 are used to run the local data mining steps with BUC. Lines 11 and 12 consolidate the local results to global results and filter all itemsets that still get or can get minimum global support. For these sets, lines 15 to 21 resolve missing counts for every partition. Line 23 computes the final result set.

Rescans for missing counts on partitions are done in parallel. We use a technique similar to BUC with  $support = 1$ , except that we only scan for values defined in at least one missing itemset.

All the itemsets to scan for on a partition are bundled to reduce scan effort. We do this by ordering all missing itemsets on that partition so that, if possible, itemsets following in order have an equal subset of items. This procedure prevents multiple scans for the same itemset because we can reuse the results for the shared subset to get the support for  $I_{n+1}$  based on an intermediate result for  $I_n$ .

## 4.3 Performance Evaluation

The performance evaluation was conducted as follows. The tests ran on up to 11 blade servers. Each blade was equipped with dual 32-bit Intel Xeon 3.2 GHz processors and 4 GB of main memory, and the blades were connected in a network with 1 gigabit/s bandwidth. The operating system was Microsoft Windows Server 2003 Enterprise Edition SP1. The SAP NetWeaver BI accelerator was compiled in Microsoft Visual C++ .NET.

---

### Algorithm 2 Algorithm Distributed BUC

---

**Require:**  $0\% \leq minSupport \leq 100\%$ .

```
1:  $P \leftarrow$  all partitions of a relation
2:  $results = [ ]$ 
3:
4: /* step 1 */
5: for all  $p \in P$  do
6:    $r_p = BUC(p)$ 
7:    $results \leftarrow r_p$ 
8: end for
9:
10: /* step 2 - 1 */
11:  $itemsets = consolidate(results)$ 
12:  $filterFrequentItemsets(itemsets, minSupport)$ 
13:
14: /* step 2 - 2 */
15: for all  $p \in P$  do
16:   for all  $i \in itemsets$  do
17:     if  $i \notin r_p$  then
18:        $results \leftarrow countMissingItemset(p, i)$ 
19:     end if
20:   end for
21: end for
22:
23:  $itemsets = consolidate(results)$ 
24:  $filterFrequentItemsets(itemsets, minSupport)$ 
25:
26: return  $itemsets$ 
```

---

### 4.3.1 Data

We used several different datasets to validate our implementation and chose two of them for this evaluation:

1. Three versions of a mushroom table (<http://www.aialab.si/orange/datasets.asp>) with original sizes of 8126, 812600, and 8126000 rows and 10 attributes
2. Real customer business data from an SAP InfoCube with about 180 million rows and 15 attributes relevant for association rule mining. The attributes have between 1 and 20000 distinct values.

The mushroom set is inflated in size by adding the original data 100 or 1000 times to new tables. This causes 100 or 1000 replications of every row of the original table. These rows are randomly distributed over all partitions. Association rule mining on these inflated mushroom tables results in the same frequent itemsets and rules as for the base version of the mushroom but with 100 or 1000 times the absolute support. The relative support ( $\frac{support}{|relation|}$ ) and confidence are constant. This dataset is used to show the effect of table size on execution time and the scaling of execution time with the number of utilized blades. Table 1 shows the dependency between support and the number of frequent itemsets, which grow exponentially but with shrinking support. The effect of this behavior is a serious increase of I/O and merge costs for low support values.

The real business dataset should close the gap to the practical utilization of the algorithm. The inspected table was the fact table of a real customer SAP InfoCube, the attributes were dimension IDs with 5 to 100000 distinct values. Table





