On Effectiveness of Database Accessing Methods for Subset Searching

Maciej Zakrzewicz
Institute of Computing Science
Poznan University of Technology
ul. Piotrowo 3a, 60-965 Poznan, Poland
Tel.: (+48) 61 878 23 78
Fax: (+48) 61 877 15 25
mzakrz@cs.put.poznan.pl

Abstract

The field of relational database research has developed many database accessing methods for effective data retrieval, e.g. B^+ tree indexing, Bitmap indexing, hash-based join, sort-merge join. These methods are oriented on finding or joining single items (represented by records) that satisfy point or range conditions. However, in the area of data mining research, there is often the need to effectively retrieve the multi-item sets that contain a given multi-item subset. We will refer to this type of retrieval as Subset Search Problem.

In this paper we formally define the Subset Search Problem. We analyze and experimentally verify the usefulness and effectiveness of the most common database accessing methods applied to the Subset Search Problem. The results show that Bitmap indexing gives the best improvement to the implementation of the Subset Search Problem.

Keywords: data mining, database indexing

1. Introduction

Data Mining is a new area of database research that aims at finding previously unknown and potentially useful patterns in large databases [[5]]. The patterns are called knowledge and are usually represented by means of *knowledge rules*. There are many types of knowledge rules: classification, characteristic, discriminant, association etc., which are employed by different data mining tasks [[14]]. The most commonly sought patterns are *association rules* [[1]]. Intuitively, an association rule identifies a frequently occurring pattern of information in a database. Formally, by an association rule we mean a formula of the form $X \rightarrow Y$, where X and Y are two sets of items. Association rules are discovered from database tables that store sets of items. Consider a supermarket database where the set of items purchased by a customer on a single visit to a store is recorded as a *transaction*. The supermarket managers might be interested in finding *associations* among the items purchased together in one transaction. An example of a supermarket database and the set of association rules derived from the database are presented in Figure 1. The example discovered rule: `bread ∧ butter ∧ milk → apples` states that a customer who purchases bread, butter and milk, probably also purchases apples. We refer to the left hand side of the rule as *body*, and to the right hand side as *head*. We also say, that the rule is *satisfied* by a given item set (item set *satisfies* the rule) if $X \cup Y$ is contained in the set. We say, that the rule is *violated* by a given item set (item set *violates* the rule) if the set contains X, but does not contain Y. Each rule has two measures of its statistical importance and strength: *support* and *confidence*. The support of the rule equals to the number of item sets that satisfy the rule divided by the number of all item sets. The rule confidence equals to the number of item sets that satisfy the rule divided by the number of item sets that contain X.

Transaction_id	items
1	bread, butter
2	bread, butter, milk, apples
3	bread, butter, milk, apples

bread → butter
bread ∧ butter ∧ milk → apples

Figure 1. Example of a database and discovered rules

The main application area of association rules discovery is *basket analysis*. The basket analysis consists in finding the dependencies between products bought by customers in a supermarket. The results of the basket analysis can help in marketing, pricing, inventory planning or shelf planning to increase transactions and profit. For example, the observation that when customers buy milk, they also buy corn flakes, may lead to the price promotion of milk to increase the sale of corn flakes. Another popular data mining application is deviation detection that identifies deviations from established statistical norms in order to find suspicious data that may be indicative of fraudulent activity. Through deviation detection, for example, a financial services company could easily detect possible fraudulent transactions by examining deviations in customer transaction patterns. Deviation detection consists mostly in finding the item sets that violate given set of rules.

The data mining process is interactive and iterative in nature. Users are interested in finding rules that satisfy given constraints rather than finding all possible rules in a database. Such process requires an efficient knowledge

discovery management system (*KDDMS*) [[7]] cooperating with a database management system (*DBMS*), and a special query language that allows users to express their specific rule discovery problems [[10]]. By means of a *rule query*, expressed in a query language, user specifies his requirements for rules to be discovered and KDDMS uses DBMS to explore the database tables and find rules that satisfy the user conditions. The rules are then returned to the user as the result of the data mining process. Such data mining process can produce hundreds or thousands of rules fulfilling user requirements. Usually, the rules are stored in a database for future retrieval by users or decision support systems. Having stored the rules, users may search and analyze them to find more specific rules e.g. rules that relate bread and milk. Later on, the users may constrain their search to rules that associate bread, milk plus apples. In general, the users may iteratively penetrate the set of rules discovered from the given database from many points of view. Moreover, users might be interested in finding customer transactions that e.g. violate the rule about bread and milk association. Generally, the users may look for data item sets that satisfy or violate given rule or set of rules.

In this paper we consider the problem of retrieval of item sets stored in a relational database, which we refer to as the Subset Search Problem. We propose a data structure for the item sets storage in a database and we give examples of practical applications of such structure in data mining area. Then, we analyze and experimentally compare the performance of the most popular database accessing methods applied to the Subset Search Problem.

1.1 Related Work

Database researchers have developed many database accessing methods for effective data retrieval. These methods can be classified into two groups: 1. without use of any additional permanent data structures (sort-merge join, hash-based join, nested-loops join) and 2. using additional permanent data structures (B^+ tree indexing, Bitmap indexing etc.).

The most popular methods of performing database joins on large database tables without use of any indexes are sort-merge join and hash-based join. Both methods are based on full table scanning. To improve the performance of database join processing, sort-merge join method first sorts the joined tables and then performs the join [[9]]. The hash-based join method first builds a hash table for one of the joined database tables, and then performs the join [[17]].

Database indexes provided today by most database systems are B^+ tree indexes to retrieve records of a table with specified values involving one or more columns [[4]]. This type of index is very effective in performing point and range queries for sparsely-populated attribute values. However, performance of B^+ tree index significantly decreases for attributes with relatively small number of key values compared to the number of records that are uniformly spread over the table. Another type of index that received recently significant attention and was implemented by most of commercial DBMSs is Bitmap index. Bitmap indexes were first developed for database use in the Model 204 product from Computer Corporation of America [[12]]. They use Bitmaps to represent record identifier lists. It was shown in [[13]] that Bitmaps are usually more CPU and space efficient than traditional B^+ tree indexes.

1.2 Outline

The paper is organized as follows. In Section 2 we introduce the Subset Search Problem, consider the data structures for item sets storage and explain the implementation of the Subset Search Problem in a relational, SQL-accessed database. In Section 3 we describe the application of the most popular database accessing methods to the Subset Search Problem. Section 4 contains our experimental results on synthetic data. Section 5 contains final conclusions.

2. The Subset Search Problem

2.1 Problem Formulation

In this Section we define and analyze a generic structure for item sets storage in relational databases. Next, we introduce the Subset Search Problem and explain its implementation in a relational database.

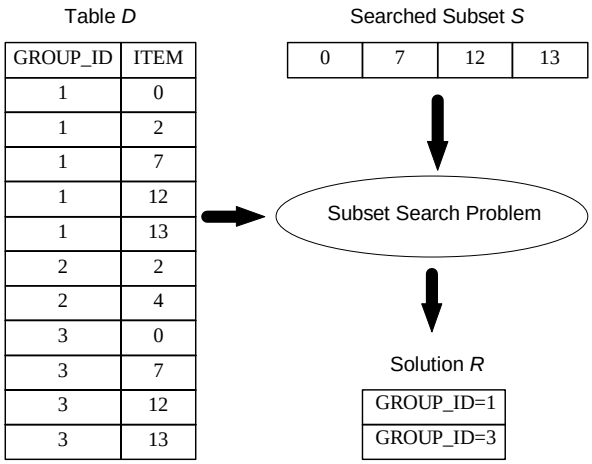


Figure 2: Example of the Subset Search Problem

We are given a large *database table of item sets D* in which each record *T* has two attributes: *GROUP_ID* and *ITEM*. The attribute *ITEM* represents an item and the attribute *GROUP_ID* represents a set that contains the item. For the sake of simplicity, we will alternatively refer to the attribute *GROUP_ID* as the *item set identifier*. In other words, each item set consists of records with the same value of *GROUP_ID* attribute. Let n_{trans} denote the *number of item sets* and n_{items} denote the *number of items*. We assume that both attributes are integer numbers, $GROUP\_ID \in <0, n_{trans}>$ and $ITEM \in <0, n_{items}>$. We are also given a *searched subset of items S* which is a collection of the form $\{item_1, item_2, ..., item_n\}$, where $item_i$ is an integer number, $item_i \in <0, n_{items}>$ and *n* is referred to as *searched subset size*.

Definition 1

The *Subset Search Problem* is to find in the database table *D* all item sets that contain all items from the subset *S*. The solution of the Problem consists of the set *R* of integer values *v* representing the identifiers of those item sets that contain the subset *S*:

$$R = \left\{ v : \underset{i=1..n}{\forall} \underset{T \in D}{\exists} S.item_i = T.ITEM \wedge v = T.GROUP\_ID \right\}$$

Let us consider the following example of the Subset Search Problem. Figure 2 presents the database table *D* of three item sets. The item set identified by the attribute *GROUP_ID*=1 contains the items: 0, 2, 7, 12, 13. The item set identified by the attribute *GROUP_ID*=2 contains the items: 2 and 4. Finally, the item set identified by the attribute *GROUP_ID*=3 contains the items: 0, 7, 12, 13. Let the searched subset of items *S*={0, 7, 12, 13}. The solution of this Subset Search Problem will be *R*={1, 3} because only the item sets identified by the attribute *GROUP_ID*=1 and *GROUP_ID*=3 contain all items from the set *S*.

Table *SHOPPING*

TRANSACTION_ID	ITEM
1	bread
1	butter
2	bread
2	butter
2	milk
2	apples
3	bread
3	butter
3	milk
3	apples

Figure 3: Example storage structure for purchase transactions

The Subset Search Problem can be found in several data mining tasks. Let us first show some examples of database tables of item sets. The model of item set organization presented in Figure 2 is commonly used for storage of both supermarket purchase transactions and association rules. Figure 3 illustrates an example database table *SHOPPING*

that stores supermarket purchase data. The attribute *TRANSACTION_ID* identifies a purchase transaction and the attribute *PRODUCT* refers to a product purchased in the transaction. Figure 4 illustrates a data model and its example relational representation for association rules storage. Rule bodies and heads are placed in one database table, while the second table keeps specific rule parameters (e.g. support and confidence values). In this example, two rules from the Figure 1 are represented. Each item of the rule body or head is stored as a separate record in the table *ELEMENTS*. The attribute *RULE_ID* groups the rule items into rules and the attribute *ITEM* represents a rule body or head item.

For the above form of item set organization we can distinguish two basic types of queries that are usually issued in searching purchase transactions or association rules:

- retrieve all purchase transactions that contain given subset of items
- retrieve all association rules that contain given subset of items in their bodies or heads

We describe each of these queries in turn below.

1.Retrieval of all purchase transactions that contain given subset of items.

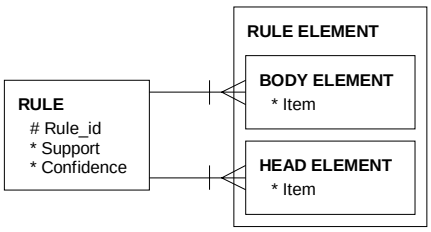
This type of retrieval can be used to determine the purchase transactions that satisfy or violate the specified rules. Finding the purchase transactions that violate strong rules allows deviation detection, described in the Introduction. This kind of queries could be also applied by a rule mining algorithm to discover association rules containing a specified set of items.

2.Retrieval of all association rules that contain given subset of items in their bodies or heads.

Queries for retrieval of association rules containing given items allow searching the database of rules for specific rules. Users impose the conditions on the contents of rules that are to be retrieved from the database, e.g. a user may look for all rules that have the items (`bread,butter`) in their bodies.

Notice that both types of the queries are in fact instances of the Subset Search Problem - they consist in finding item sets that contain a given item subset.

Let us analyze physical properties of the database tables that are subject to the Subset Search Problem. In practical applications the number of item sets n_{trans} may be very large (supermarkets incrementally register thousands of transactions every day) and the average size of an item set is of the order of 10-100. Therefore, the selectivity of the attribute *GROUP_ID* will be very high because only 10-100 records from a large database table of even millions of records will have the same value of the attribute



RULES

rule_id	support	confidence
1	0.83	0.90
2	0.25	0.13

ELEMENTS

rule_id	item	type
1	bread	body
1	butter	head
2	bread	body
2	butter	body
2	milk	body
2	apples	head

Figure 4 Example data model and tables for association rule storage

GROUP_ID. Moreover, since all items of an item set are usually being stored in a database at the same point of time, they are very likely to be located in one or very few adjacent disk blocks.

Let us now consider the properties of the attribute *ITEM*. In practical applications the number of all items n_{items} is relatively small (supermarkets sell a constant number of hundreds of different products), however, an individual item may occur in large number of item sets (e.g. milk is included in almost every transaction, beer may be included in 30% of transactions etc.). Therefore, the selectivity of the attribute *ITEM* is very low and its individual values are sparsely populated in a database.

2.2 Subset Searching Queries

The Subset Search Problem is not well supported by SQL query language and traditional query execution techniques. To illustrate the subset searching, we present below an example of an SQL query retrieving from a database table *D* the identifiers of item sets containing the items 0, 7, 12 and 13. We will further refer to this type of query as the *subset searching query*:

```
SELECT T1.GROUP_ID
FROM   D T1, D T2, D T3, D T4
WHERE  T1.GROUP_ID = T2.GROUP_ID
AND    T2.GROUP_ID = T3.GROUP_ID
AND    T3.GROUP_ID = T4.GROUP_ID
AND    T1.ITEM = 0
AND    T2.ITEM = 7
AND    T3.ITEM = 12
AND    T4.ITEM = 13;
```

To demonstrate the nature of the subset searching query, let us consider its execution plan, given in Fig. 5. In fact, any execution plan for the query will have the similar structure. It will consist of four selection and three join operations. For the considered execution plan, at the first step, identifiers of all item sets containing items 0 and 7 are selected. These sets can be very large due to the low selectivity of the selection predicates - it is very likely that most of item sets (e.g. purchase transactions) will contain

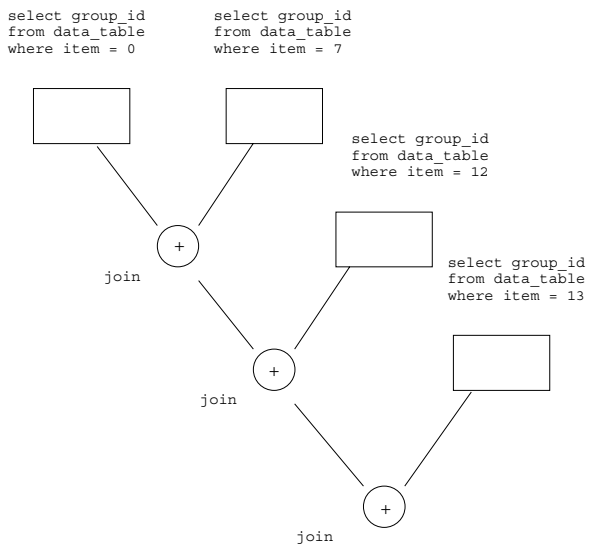


Figure 5: Execution plan for the example query

the items 0 and /or 7 (e.g. 0 may corresponds to “bread”, 7 may correspond to “milk”, etc.). Then, at the next step, the set of identifiers found in the previous step is joined with a set of identifiers of item sets containing the item 12. At the last step, it is joined with a set of identifiers of item sets containing the item 13. As the result, identifiers of all item sets containing items 0, 7, 12, and 13 are returned. There are two main disadvantages of these plans. First, as it can be seen, the number of joins in the query execution plan strictly depends on the size of the searched subset. Larger searched subset results in larger subset search query. For searched subsets of the size greater than 10 we deal with large join queries that are not well supported by traditional query optimizers [[8],[15]]. Second, the intermediate tables resulting from the selection operations may be very large due to the low selectivity of the selection predicates. The intermediate tables are joined to select those item set identifiers that are contained in all tables. This subset containment procedure is performed in step-by-step manner. So, it may occur that the system has to deal with very large intermediate tables at consecutive steps of the procedure. In fact, as we will show it later, sizes of intermediate tables have the main impact on the execution cost of this type of queries.

3. Database Accessing Methods Applied To The Subset Search Problem

In this Section we describe four most commonly implemented database accessing methods: sort-merge join with full table scan, hash-based join with full table scan, B^+ tree indexing and Bitmap indexing. We consider their application to improving the performance of the subset searching queries presented in Section 2.

The most primitive method of searching the database is full table scan. To perform a full table scan, DBMS reads all records of the table, examining each record to determine whether it satisfies the query's WHERE clause. Since DBMS reads every disk data block allocated to the table sequentially, a full table scan can be performed very efficiently using multiblock disk reads. During the full table scan, DBMS reads each disk data block only once. Although the full table scan is generally the worst method of searching the database, it may outperform the other methods when dealing with specific datasets (e.g. full table scan outperforms B^+ trees for selection attributes with high number of key values).

A popular method of performing database joins without help of any indexes is sort-merge join method [[9]]. In this method, the database tables are first sorted according to the value of the join attribute. Then a pointer is associated with each table. The pointers point initially to the first record of the respective tables. As the sort-merge algorithm proceeds, the pointers move through the table. A group of records of one table with the same value of the join attribute is read. Then the corresponding tuples of the other table is read. Since the tables are in sorted order, records with the same value of the join attribute are in consecutive order.

Another method of performing database joins is hash-based join [[17]]. The basic approach of several hash-based join methods is to dynamically build a hash table on the join attributes for one database table and then to probe this hash table using hash values on the join attributes of records from the other database table. The result of the join consists of the matching records. It is usually assumed that memory that is available is much smaller than the size of database tables to be joined, therefore it is impossible to build the hash table for the entire database table. The hash-join algorithms usually process the join in batches. In each batch, only a portion of a database table is read into memory and the corresponding hash table is built.

The role of indexing in query optimization is well-understood in the database community. The purpose of an index is to provide pointers to the records in a table that contain a given key value, thus enabling efficient access to a subset of a database. The most popular database indexing techniques are based on B^+ trees. Figure 6 shows an example of the B^+ tree. Each non-leaf node contains entries of the form (v, p) where *v* is the separator value which is derived from the keys of the records and is used to tell which subtree holds the searched key, and *p* is the pointer to its child node. Each leaf node contains entries of the form (k, p), where *p* is the pointer (not depicted in Figure 6) to the record corresponding to the key *k*. Searching a record in a database table with help of B^+ tree index is straightforward. Let us consider an example of searching the record which has the key *N*. Beginning from the root node, we find that $J<N<=S$, that leads to the node which contains the separators *N* and *Q*. Next, we find that $N<=N$, thus the target record should be in the leaf-node which contains the keys *L* and *N*. The searching of this leaf-node results in locating *N*.

B^+ tree indexes are very effective in performing point and range queries for highly selective attributes. However, their performance significantly decreases for attributes that have a small number of key values compared to the number of records and that are uniformly spread over the database table. In the context of the Subset Search Problem, a B^+ tree index could be used for improving the joins on the attribute *GROUP_ID* of the database table *D*. As we have already noticed, the attribute *GROUP_ID* joins the intermediate results of subset searching query and its selectivity is usually very high. A B^+ tree index should not be used for

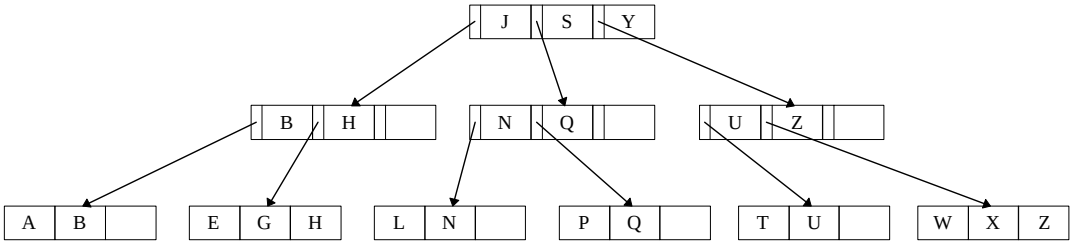


Figure 6: An example of the B^+ tree

selections of the attribute *ITEM* because of its low selectivity.

Another type of index that received recently significant attention and was implemented by most of commercial DBMSs is Bitmap index. Bitmap indexing benefits data warehousing applications which have large amounts of data and ad hoc queries, but a low level of concurrent transactions. The Bitmap index is a collection of *k* 0-1 vectors, called bitmaps, where *k* is a number of indexed attribute's key values. Each bit in the bitmap corresponds to a record, and if the bit is set, it means that the corresponding record contains the key value. Figure 7 shows an example of a database table and its Bitmap index. The example Bitmap index consists of three bitmaps, one for each key value of the attribute *COL1*. The data retrieval by means of the Bitmap index consists in fast scanning one or few of the bitmaps for non-zero elements. For example, searching the database table from Figure 7 for records that contain the key value *'B'* is done by finding for which records the second bitmap has its bits set to 1. If the number of different key values is small, bitmaps are very space and time efficient. Moreover, Bitmap indexing efficiently merges indexes corresponding to several conditions in the WHERE clause of an SQL query. Records that satisfy some, but not all the conditions, are filtered out before the table itself is accessed. As a result, response time is improved.

In the context of the Subset Search Problem, a Bitmap index could be used for selection improvement of the attribute *ITEM* of the database table *D*. As we have already noticed, the attribute *ITEM* is used to filter the database table *D* for only those items that appear in the searched subset *S*. Due to the relatively small number of the key values, its selectivity is usually very low. A Bitmap tree index should not be used for joins on the attribute *GROUP_ID* because of a huge number of its key values, that would result in huge number of bitmaps to be stored and processed.

Table

REC#	COL1
1	A
2	B
3	A
4	C
5	B
6	C

Bitmaps

COL1=A	COL1=B	COL1=C
1	0	0
0	1	0
1	0	0
0	0	1
0	1	0
0	0	1

Figure 7: An example of the Bitmap index

4. Experimental Results

To assess the performance of the database accessing methods we performed several experiments on Oracle 7.3.1 DBMS, working on 2-processor Sun SPARCserver 630MP, with 128 MB of main memory. In this Section, we describe the synthetic database and present the results of the subset searching using different database accessing methods.

4.1 Synthetic Database

Experimental data sets were created by synthetic data generator *GEN* from Quest project [[2]]. *GEN* generates

textual data files containing sets of numerical items. Several parameters affect the distribution of the synthetic data. These parameters are shown in Table 1.

To load the contents of the data files into the database, *Oracle SQL*Loader* program was used. The item sets were stored in database tables of the following structure:

attribute name	datatype
GROUP_ID	NUMBER(6,0)
ITEM	NUMBER(6,0)

parameter	value
n_{trans}	number of item sets, 50,000
n_{items}	number of different items, 100 to 500
t_{len}	average items per set, 15 to 30
n_{pats}	number of patterns, 500 and 10000
patlen	average length of maximal pattern, 4
corr	correlation between patterns, 0.25

Table 1 Synthetic data parameters

4.2 Experimental Results

The main performance metric of the subset searching was mean execution time of test queries. *Oracle* SQL query tracing and *TKprof* profiler were used to measure precisely query execution time. Four types of test queries were issued: the queries that used sort-merge join method, the queries that used hash-based join method and the queries using B^+ tree index on the attribute *GROUP_ID* and Bitmap index on the attribute *ITEM*.

Table 2 shows the explanation of labels from our experiment diagrams.

Our first impression is that the Subset Search Problem is generally very time-consuming. The test queries execution times for 10-item searched subsets were of the order of hundreds of seconds while simple single selections in the same database tables could take only few seconds.

Figure 8 shows the performance of database accessing methods for different sizes of a searched subset. For increasing size of a searched subset, the number of database accesses also increases because of a greater number of query joins to be performed. Thus, the performance of all the methods is getting worse when the searched subset size is increasing. In general, as we expected, usage of database indexing results in shorter query execution times. Both B^+ tree and Bitmap indexing significantly outperformed sort-merge join and hash-based join methods. The fastest database accessing method appeared Bitmap indexing, which provided four times faster retrieval than sort-merge join and hash-based join methods.

Figure 9 demonstrates the effect of the average size of item sets on the performance of database accessing methods. For increasing average size of item sets, the number of records in the database table also increases and the query execution time gets longer. For item sets of the size greater than 20 items, hash-based join method performs better than sort-merge join method. In general, when the item set size increases, the size of the database *D* increases but the number of key values of its attribute *GROUP_ID* remains constant, therefore B^+ tree index' performance is getting relatively worse. As we have observed, when the average size of item sets was greater than 25, then hash-based join method even outperformed B^+ tree indexing method. Again, the best database accessing method for all examined item sets sizes appeared Bitmap indexing.

label	description
sort-merge	results of subset searching with *Oracle* sort-merge join
hash-join	results of subset searching with *Oracle* hash-based join
B^+ tree	results of subset searching with *Oracle* B^+tree index for the attribute *GROUP_ID*
bitmap	results of subset searching with *Oracle* bitmap index for the attribute *ITEM*

Table 2 Labels from the experiment diagrams

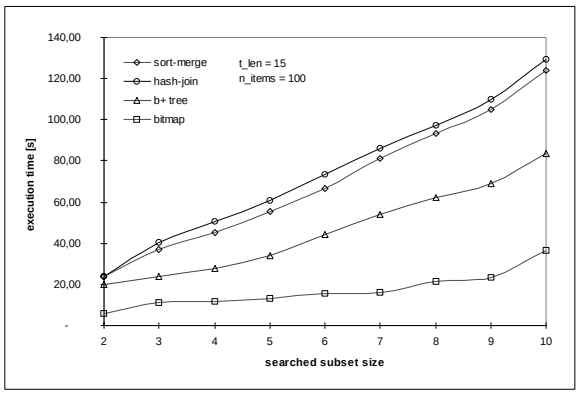


Fig.8: Query execution time vs. searched subset size

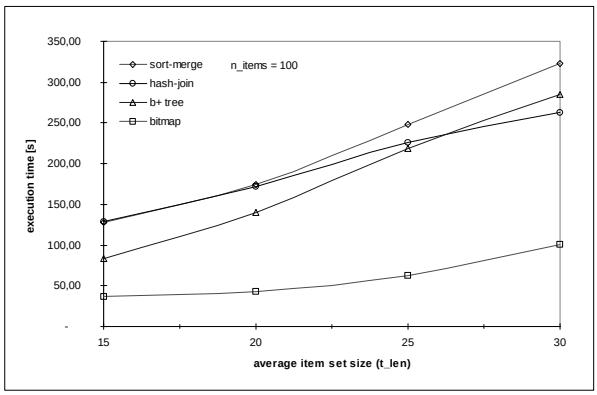


Fig.9: Query execution time vs. average item set size

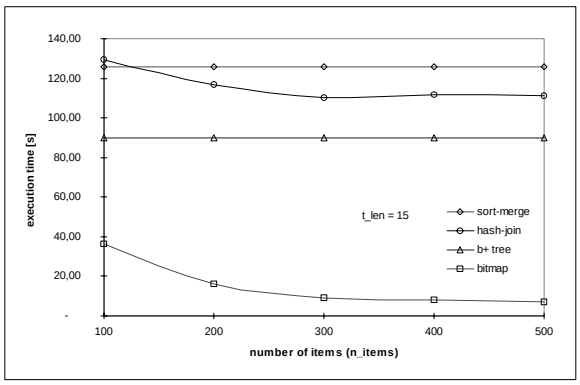


Fig.10: Query execution time vs. number of items

Figure 10 demonstrates the effect of the number of items stored in the database on the performance of database accessing methods. The number of items influences the size of intermediate join results during query execution process. When the number of items is greater, the selectivity of the attribute *ITEM* is higher, thus the intermediate join results are smaller. The experiment showed, that this property does not significantly improve the performance of sort-merge join, hash-based join as well as B^+ tree indexing. We have observed a lowering improvement of Bitmap indexing performance for increasing number of items. Once again, the best database accessing method for all examined numbers of items was Bitmap indexing.

5. Concluding Remarks

In this paper we introduced the Subset Search Problem in relational databases which consists in retrieval of multi-item sets that contain a given multi-item subset. The Subset Search Problem has many applications in the field of data mining. We analyzed and compared the performance of the subset searching queries for different database accessing methods. We showed experimentally that Bitmap indexing is the best method of improving the subset searching queries performance. However, we realize that even Bitmap indexing results in hardly acceptable query execution times.

The results of this paper can be extended in several directions. First, we may study any alternative plans of executing the Subset Search queries. As we have noticed, the number of joins in the presented query execution plan strictly depends on the size of the searched subset. Larger searched subset results in larger subset search query. This

property significantly reduces the performance of large subset searching queries. Second, new index structures could be invented to reduce time of large subset searching in large relational databases. We suspect that if an index structure was built over data sets instead of items, the performance of the subset searching could be significantly better. Furthermore, we would like to analyze different data storage strategies (e.g. index clusters, hash clusters) for their application to the Subset Search Problem.

6. Bibliography

[1] Agrawal R., Imielinski T., Swami A., Mining Association Rules Between Sets of Items in Large Databases, Proc. of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, May 26-28 1993,

[2] Agrawal R., Mehta M., Shafer J., Srikant R., Arning A., Bollinger T., The Quest Data Mining System,

[3] Agrawal R., Srikant R., Fast Algorithms for Mining Association Rules, Proc. of the 20th VLDB Conference, Santiago, Chile, 1994,

[4] Comer D., The Ubiquitous B-tree, Comput. Surv. 11, 1979,

[5] Fayyad U., Piatetsky-Shapiro G., Smyth P., The KDD Process for Extracting Useful Knowledge from Volumes of Data, Comm. of the ACM, Vol. 39, No. 11, November 1996,

[6] Han J., Fu Y., Discovery of Multiple-Level Association Rules from Large Databases, Proc. of the 21st VLDB Conf., Swizerland, 1995,

[7] Imielinski T., Manilla H., A Database Perspective on Knowledge Discovery, Comm. of the ACM, Vol. 39, No. 11, November 1996,

[8] Ioannidis Y.E., Kang Y., Left-deep vs Bushy Trees: An Analysis of Strategy Spaces and Its Implications for Query Optimization, Proc. of the ACM SIGMOD Int. Conference on Management of Data, Denver, USA, 1991,

[9] Korth H. F., Silberschatz A., Database Systems Concepts, McGraw-Hill, 1986,

[10] Morzy T., Zakrzewicz M., SQL-Like Language For Database Mining, ADBIS'97 Symposium, St. Petersburg, September 1997,

[11] O'Neil P., Graefe G., Multi-Table Joins Through Bitmapped Join Indices, SIGMOD Record, September, 1995,

[12] O'Neil P., Model 204 Architecture and Performance, Springer-Verlag Lecture Notes in Computer Science 359, 2nd Int. Workshop on High Performance Transactions Systems (HTPS), Asilomar, CA, 1987,

[13] O'Neil P., Quass D., Improved Query Performance with Variant Indexes, Proc. Of the 1997 ACM SIGMOD, Tucson, Arizona, 1997,

[14] Piatetsky-Shapiro G., Frawley W.J., editors, Knowledge Discovery in Databases, MIT Press, 1991,

[15] Salza S., Morzy T., Matysiak M., Tabu Search Optimization of Large Join Queries, Proc. of 4th Int. Conf. EDBT'94, Cambridge (UK), 1994,

[16] Savasere A., Omiecinski E., Navathe S., An Efficient Algorithm for Mining Association Rules in Large Databases, Proc. of the 21st VLDB Conference, Zurich, Swizerland, 1995,

[17] Shapiro L. D., Join processing in database systems with large memories, ACM Trans. Database Syst., Vol. 11, No. 3, 1986

[18] Srikant R., Agrawal R., Mining Generalized Association Rules, Proc. of the 21st VLDB Conf., Swizerland, 1995,

[19] Toivonnen H., Sampling Large Databases for Association Rules, Proc. of the 22nd VLDB Conf., India, 1996,