INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

CIRCUMSTANCE MINING AND ITS IMPLEMENTATION

Ming Hao Xie

A Major Report
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Science (Computer) at
Concordia University
Montreal, Quebec, Canada

March 2001

ABSTRACT

Circumstance Mining and its Implementation

Ming Hao Xie

This report is an explanation and exploration of circumstance mining and related algorithms. The emphasis is placed on design and implementation of algorithms for circumstance mining. Some of the issues covered in this report are: introduction to circumstance mining, analyses of related algorithms, design and implementation of new algorithm, and comparison of algorithms. Due to the exploratory nature of the study, some assumptions are introduced in algorithm implementation.

Acknowledgment

Great thanks go to:

Prof. Laks V.S. Lakshmanan, my supervisor, who guided me through this project and opened a new horizon for me;

Lili Fu, my wife, who has been supportive for a long time and who made this report possible; and

Xiaohong Wang, my partner, who inspired me so much in so much argument.

Table of Contents

Table of Figures

1 Introduction

1.1 Introduction to Data Mining and More

Data mining is a step-wise process of knowledge discovery in databases. It includes data-warehousing, statistical analysis, feature selection, knowledge extraction, etc. The objective is to find comprehensible rules and relations in a large number of item-sets. These rules can be used to predict future trends, to profile customers and clients, to assess risk in a wide range of areas and for the explanation of core components in large databases.

Here are several examples (queries):

Q1: Is a certain item-set {beer, potato chips} brought frequently?

Q2: People buy item-set like {bread, cheese} will also buy {milk}?

These comprehensible rules and relations are regarded as patterns. Interesting patterns like associations [1] in 1994, correlations [2] in 97, causality mining in 1997 and constraints on mining [11] in 1996 have been proposed in recent years. Answering the queries mentioned above is actually the process of finding, or mining corresponding patterns from a huge amount of data. Extensive work has been done on efficiently mining of these patterns and a number of algorithms have been proposed. These algorithms exploit pattern from different viewpoint, while all are bound to the same question "which patterns hold in the database".

But one important factor is ignored here, that is "circumstance". For example, {beer, potato chips} is sold frequently in North America, but how about in Asia? {Air conditioner} is welcome in summer but not in spring, etc. Therefore, it's useful to take the circumstance into consideration in data mining.

Recently, Laks V.S. Lakshmanan, Gosta Grahne and Xiaohong Wang [3] noticed and indicated that in addition to determining pattern, it's also needed to analysis the circumstance under which certain patterns hold. In other words, question like "under what circumstance a given pattern holds in the database" also deserves attention and studies.

Taking the circumstance into account, above-mentioned examples can be converted into the following queries:

Q3: Under what circumstance (say, on time of purchase, location) is the item-set {beer, potato chips} bought frequently?

Q4: Under what circumstances (e.g. climate, location) will people buying item-set like {bread, cheese} also likely buy {milk}?

With the introduction and application of circumstance mining, it becomes more complete and more meaningful to predict future trends, to profile customers and clients, to assess risk in a wide range of areas. Therefore it's worthwhile to exploit algorithms that are suitable to circumstance mining. In next section, a brief definition of the circumstance and problem will be given.

1.2 Scope and the problem

1.2.1 Definition of Circumstance

In traditional framework, item-sets can be organized into a huge relational table. Each transaction may be partitioned into several rows in the table. For example, customer Jack bought {Beer, Diaper, Cigar} on Wednesday, this transaction maybe stored in the following form:

Customer	Time	Item	Number
Jack	Wednesday	Beer	4
Jack	Wednesday	Diaper	20
Jack	Wednesday	Cigar	3
Tom	...		

Table 1 Example of item-set table

As far as data mining is considered, most algorithms will treat all attributes alike and process the whole table without distinguishes. [3] argues that "...it misses out the opportunities of exploring and exploiting the different structures and properties

association". The first step in circumstance mining is to distinguish different attributes in database. Two types of attribute are proposed: 1) *item attributes* and 2) *circumstance attributes*. An attribute $A \in R \backslash k$ is regarded as *circumstance attribute* in database r provided the functional dependency $k \rightarrow A$ holds for r. The rest can be treated as *item attributes*. In [3], *item attributes* can be further distinguished from *descriptive attribute*, which goes beyond my report here. Details please refer to [3] section 2.

1.2.2 Monotone and Circumstance

Given a certain pattern, there will be a number of circumstances under which the pattern can hold. Most of them have "stronger" or "weaker" relationship. For example, consider circumstance C1: *city = New York & whether = sunny & month = July* and circumstance C2: *city = New York*. Obviously C1 is stronger than C2. On the other hand, C2 is weaker than C1. If a frequent item-set {beer, cigar} holds under C1, obviously it will also hold under C2. More generally, if a pattern that holds under certain circumstance also holds under weaker circumstances, that pattern can be called monotone. As can be seen, to a monotone pattern, stronger circumstance implies weaker ones, which can be used to simplify the result of circumstance mining. When circumstance mining is processed, it's desirable to remove all circumstances that are implied by other circumstances. In other words, only the set of strongest circumstances is wanted. Circumstance mining algorithm should take it into account.

It's should be noted that there also exist anti-monotone patterns. Monotone or anti-monotone depends on the nature of the pattern. Because processing principles behind are almost identical, so in this report, only monotone patterns are discussed. Complete definition are described in [4] and [3]

1.2.3 CUBE Operation and Circumstance

Jim Gray et al.[5] introduced a relational aggregation operator of data cube. The CUBE operator is the n-dimensional generalization of the group-by operator. It computes group-bys corresponding to all possible combinations of a list of attributes. The whole groups-bys constitutes a lattice. On the other hand, the space of circumstances, in many applications, also forms a lattice, which is called circumstance lattice. It's noted that

circumstance lattice bears a strong similarity to the lattice used for data cube computation. Indeed, when computation is limited to circumstances in which only equality is involved, the circumstance lattice is identical to the lattice corresponding to the instances of various group-bys. This suggests that any of the algorithms developed for cube operations should be useful for mining circumstance.

The differences hold in processing the lattice. In circumstance mining, first, only minimal circumstances (border points) instead of whole lattice nodes are wanted; second, counter for each node in not necessary (we only care whether the node gets enough support or not).

1.2.4 Definition of the Problem

A generic circumstance mining question is "given a pattern, how to find its satisfying circumstances?" Here the circumstance means the strongest circumstance (mentioned in **1.2.2**). Also, "satisfying" means a conjunction of predicates of the form ‹ è*value*, where *attr* is a circumstance attribute, and *value* comes from its doma È here is a predicate, typically, a equality. In this report, for simplicity, only the equality is considered. As to "pattern", although there are many different patterns, this report picks frequency as a representative pattern for detailed study. The reason is frequent item-sets is the simplest among those discussed and yet epitomizes the issues that arise in studying other patterns. In next session, some algorithms related to cube computation will be introduced, which shares a lot similarities with circumstance mining.

The rest of the report is organized as follows: Session 2 reviews several related algorithms and their relationships with circumstance mining. Originally all these algorithms are dedicated for pattern mining. Sessions 3 proposes a new algorithm for circumstance mining. Some details in adapting cube computation algorithms to circumstance mining is also discussed in this session. Session 4 is for performance analysis.

2 Review of Related Algorithms

As mentioned above, many algorithms have been proposed for efficient data mining. In this session, Apriori[3] is first introduced which is the foundation of some other algorithms. Then come two popular CUBE computation algorithms. In many cases, circumstance lattice shares many similarities with CUBE computation lattice [5]. Mining circumstance for a given pattern can be generally viewed as cube operation with constraints, such as minimal count support. Some obvious adaptations were made to these algorithms in session 3.

2.1 Apriori

The beauty of Apriori is its pruning, which takes full advantages of the conception monotone and anti-monotone. More specific, every subset of a frequent item-set is also frequent. Therefore, a number of candidates can be purged by using this property without further calculation.

For sake of reference, the pseudo code is attached as Fig 1. Here, L_k is set of large k-item-sets. C_k is set of candidate k-item-sets (potentially large item-sets). Each member of L_k and C_k has two fields: item-set and support counter.

```
L1={large 1-itemsets};
for ( k=2;     ≠0; k++) do begin
        Ck=apriori-gen(      ··  //New candidates
        forall transactic    ∈D do begin
        Ct=subset(Ck, t)·    //Candidates contained in t
        Forall candidat      ∈Ct do
        c.count++;
end
Lk    ∈Ck|c.co      ≥minsup}
End
Answ      ∪kLk;
```

Figure 1 Pseudo code of Apriori

Based on Apriori, some efficient algorithms were proposed. For example, in [3, 4], Laks V. S. Lakshmanan et al. analyze the property of 1-variable and 2-variable constraints, produce the algorithm CAP for constraint frequent set queries. Also, this report proposes an Apriori based algorithm for circumstance mining.

2.2 Buc

Given a *minimal support* and a database, determining the frequent item-sets is equivalent to the following CUBE operation:

SELECT P, D, C, COUNT(*), Sum(S)
FROM Transactions of database
CUBE-BY P, D, C
HAVING COUNT(*) >= N

Here *N*= |*Transactions*| * *minsup; P, D* and *C* are database attributes.

In [6], Kevin Beyer and Raghu Ramakrishnan propose the algorithm BUC which is designed for computation of sparse and Iceberg-CUBE. It's a very elegant algorithm. Its pseudo code is attached here:

```
Procedure BottomUpCube(input, dim)
Inputs:
        input: The relation to aggregate.
        dim: The starting dimension for this iteration.
Outputs:
    One record that is the aggregation of input.
    Recursively, outputs CUBE (dim, ..., numDims) on each input.
Method:
1.  Aggregate(input); // Places result in outputRec
2.  If input.count() == 1 then
    Write Ancestors(input[0], dim); return;
3.  write outputRec;
4.  for d=dim; d<numDims; d++ do
5.  let C = cardinality[d];
6.  Partition(input, d, C, dataCount[d]);
7.  let k = 0;
8.  for i = 0 ; i < C ; i++ do // For each partition
9.  let c = dataCount[d][i]
10. if c >= minsup then // The BUC stops here
11. outputRec.dim[d] = input[k].dim[d];
12. BottomUpCube(input[k . . . k+c], d+1);
13. end if
14. k += c;
15. end for
16. outputRec.dim[d] = ALL;
17. end for
```

Figure 2 Algorithm of Bottom Up Cube

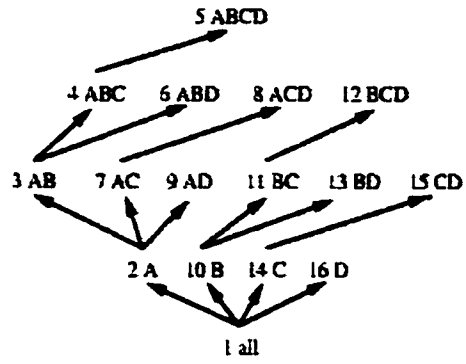


Figure 3 BUC processing tree

In the pseudo code listed in Figure 2 (above), dataCount[numDims][partitions] stores the size of each partition. dataCount[i] is a list of integers of size cardinality[i]. As can be seen, BUC invokes itself recursively. Figure 3 shows the BUC processing tree in a case of 4-dimensional Lattice. Note that the BUC processes the lattice depth-first, while Apriori processes the lattice breath-first.

In circumstance mining, the number of circumstance attributes is much less than that of item. Thus it will be efficient if we use BUC to mine interesting patterns in circumstance sets.

2.3 Chunk Array

Chunk Array is introduced in [7]. It's an array-based algorithm. The basic principal is it uses in-memory arrays to store the partitions and to avoid sorting. A chunk of an n-dimensional array is an n-dimensional sub-array.

Opposite to BUC, Chunk Array is a top-down algorithm, which means the lattice is collapsed from highest level. This unique feature can be used to construct strongest circumstances directly.

Because Chunk Array uses in-memory array to store, so generally it has good performance. But if the database can not fit the main memory, the performance de-grades a lot. The storage and index of data needed to be improved, especially for skewed database.

3 Algorithms for Circumstance Mining

In this report, 3 algorithms is implemented and compared. In order to make the comparison more objective, several conventions or assumptions are used:

1. All the 3 algorithms use same input and output modules, as illustrated in next figure,

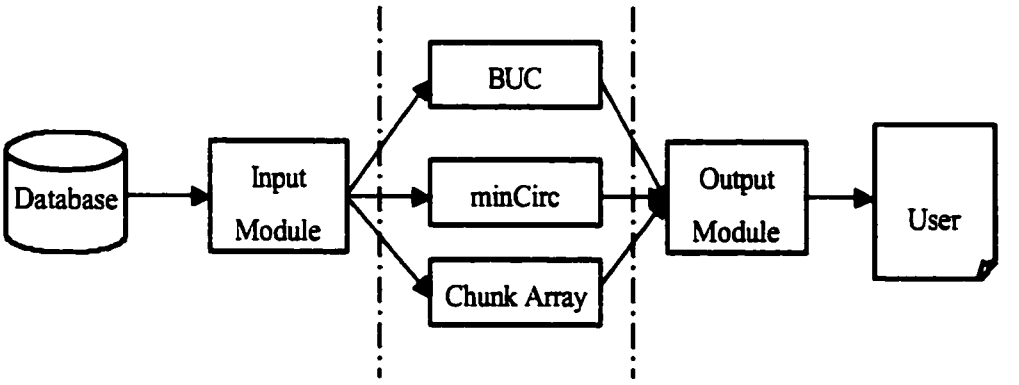


Figure 4 Data flow of algorithms

so the relative performance is isolated from details of database, from techniques used in file accessing.

2. Map string dimension attributes and types to integers between zero and the cardinality of the attribute. Mapping dimensions to integers reduces space requirements (i.e., no long strings), eliminates expensive type interpretation in the CUBE code. This is also called packing. Packing the domain between zero and the cardinality allows the dimensions to be used as array subscripts. In some papers [5], [6], it's called *packed* data.

   If the dimensions are not packed in the input, they can be packed when the input is first read by creating a hashed symbol table for each dimension as described in [5], and the mapping can be reversed when tuples are output. Or a simple pre and post-processing pass can be used.

3. All 3 algorithms use pre-screened database. Circumstance mining, to some extent, needs two steps: first picks up all circumstances satisfying the given pattern and then

process those pre-screened circumstances. In this report, it's assumed that database is already pre-screened.

3.1 BUC Adaptation

BUC was designed to incorporate constraints on groups-bys such as *count(*)>n*, which says the group-by must have at least *n* appearances. However, the express goal of BUC was to find all group-bys satisfying given constraints as opposed to minimal ones.

3.1.1 Reflection on BUC

As can be seen (Fig. 2), the strength of BUC is that it combines the pruning into a recursive computation therefore makes the whole algorithm clear and concise. As far as counting constraint is considered, BUC can be applied to compute circumstance mining with a little modification:

First, line 2 is not necessary in circumstance mining. It's an optimization for avoiding fruitless recursive computation in case of a single tuple, which is very uncommon in circumstance mining and won't help in performance testing.

Second, BUC will generate all qualified group-bys, while in circumstance mining, only the border points (minimal circumstance set) are wanted, which is a difference between cube and circumstance computation as mentioned before. So post-processing (discussed in next session) is needed to purge redundant circumstances.

Figured 3-2 shows the adaptation to the BUC algorithm for finding minimal circumstances. A minor modification consists in computing the support of S in every group-by (i.e. circumstance).

Algorithm Modified BUC;

Input: an itemset S, a support threshold s, and a transaction database db;

Output: the set of minimal circumstances in which S is frequent;

1. *run BUC algorithm, with the following modifications:*

   (a) no aggregate computation (other than verifying the support constraint) is needed;

(b) when processing a circumstance α, compute the support of S in α; if it's below s, circumstances that are imply α, including α, are pruned; otherwise, α is retained;

2. do post-processing to eliminate non-minimal circumstances;

Figure 5 Adaption to BUC Algorithm for finding minimal circumstances

3.1.2 Implementation details

Pseudo code:

```
1.  If the input (relation) is in not in main-memory
2.      Try to read whole relation into main-memory
3.      If can't, read in the first dimension of the relation
    endif
4.  for d=dim; d<numDims; d++ do
5.      Partition the input (relation) according to value of first dimension.
6.      let C = cardinality[d];
7.      Partition(input, d, C, dataCount[d]) using counting sort;
8.      let k = 0;
9.      for i = 0 ; i < C ; i++ do // For each partition
10.         let c = dataCount[d][i]
11.         if c >= minsup then // The BUC stops here
12.             outputRec.dim[d] = input[k].dim[d], feed it to post-processing;
13.             BottomUpCube(input[k . . . k+c], d+1);
            end if
14.         k += c;
        end for
15.     outputRec.dim[d] = ALL;
    end for
16. post-processing outputs minimal circumstances
```

Figure 6 Modified BUC Algorithm

3.1.2.1 Partitioning and Counting sort

As can be seen from BUC algorithm (Fig. 6), BUC refine a relation to many partitions and recursively compute each partition (relation). No doubt that majority of the time in BUC is spent on partitioning the data, so optimizing Partition() is important. Generally, hash structure is used to improve performance, but BUC in [6] proposed that if *'input'* fits in main memory, counting-sort [8], [9] is better.

Counting sort is a linear sorting method, sorting data in surprisingly $\tilde{I}(n)$. The algorithm requires that the sort key be an integer value between zero and its cardinality, which

exactly the case when rule (2) in section 3.1 is applied. In other words, when data is already pre-processed or packed. Another point is, counting sort is an in-memory sorting method. Therefore, implementation of BUC has to distinguish data according to its size and has to have two versions: BUC-External and BUC-Internal [6], which is difficult to accomplish in short time. Actually, Kevin Beyer, the author of BUC, said that [10] "... First, I only implemented the in-memory portion of the algorithm. My recommendation for when the data does not fit in memory is to use your best external sort/hash on it... I expect that external hashing would work the best..."

Another benefit from using counting sort is that partition boundaries can be found easily, because the counts computed in counting sort can be saved in BUC (line 11 in Fig 6 for example).

No doubt the use of counting sort is an important optimization to BUC. "When sorting one million record with widely varied key cardinality and skew, QuickSort ran between 3 and 10 times slower than counting sort" [6]. The penalty is its limited application. In my implementation, counting sort is still used because of its performance. In order to let my implementation also process disk-residential database. I adjusted the partition and used the first dimension in each relation to do some assistant work. With an example depicted

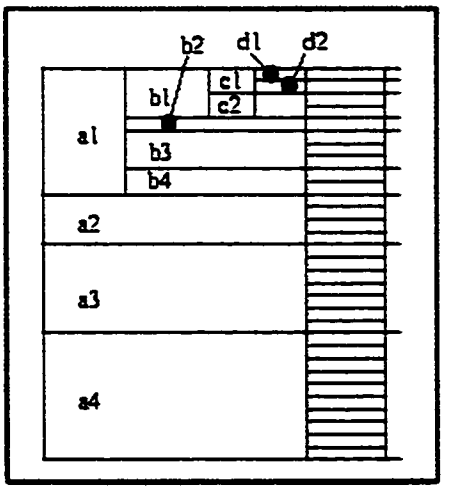


Figure 7 BUC partition map

in Fig 7, it would be easier to explain this idea.

To process a database *r* illustrated in Fig 7, BUC Adaptation (introduced in Fig 6) will first try to swallow the whole database. If it can, counting sort will be used directly without any problem because since then, no data loading is needed; if it can't, only the first dimension in *r*, in this example dimension *A* is read in. Since the whole dimension is in memory, counting sort can be applied again and generate several partitions, particularly 4 partitions *a1*, *a2*, *a3* and *a4* in this example. To each partition, BUC Adaptation can be recursively applied. Very hopefully, when a partition is refined enough, each part can be fitted into main-memory. With this modification, BUC adaptation can avoid applying hash partitioning while still enjoy the high performance of counting sort.

On the contrary, original BUC is much stricter on size of database. With same example in Fig 7, BUC-Internal (counting sort) can be applied if and only if the whole database can fit in main-memory otherwise BUC-External (hash partitioning) is recommended. User has to make decision to pick up a certain version in advance.

3.1.2.2 Memory Requirement Analysis

BUC Adaptation imposes similar memory requirement on the system with BUC does. A memory requirement analysis will make it clear.

- In case of BUC (in-memory version only)

  Let p = the size of a pointer (assumed to be 4 bytes before)
  Let i = the size of an integer (assumed to be 4 bytes before)
  let d = the number of dimensions
  let m = the number of measures
  let N = the number of (in-memory) records.
  let T = the number of bytes for each input record

  Here Pointer could be a tuple id or index or physically memory address, depending on the implementation.

  Assuming dimensions and measures are integers, then we have:

  $T = i*(d+m)$

So, we need N*T bytes for the data, N*p bytes for pointers to the data, and N*p bytes for the second set of pointers used in counting-sort (In implementation of counting-sort, the sorted pointers can be copied back to the original pointers). Totally, counting sort alone needs memory (in byte):

$$N * (T + 2*p) = N(T+8) \qquad <3.1>$$

During counting-sort, it's also needed to keep the number of records with each value. The maximum number of counters kept is C_{max}. So counting-sort needs the memory for the second set of pointers mentioned above plus memory for the counters is:

$$N(T+8) + i*C_{max} = N(T+8) + 4*C_{max} \qquad <3.2>$$

To track a partitioned set of pointers, BUC keeps the count of the number of records in each partition. When it reaches its maximum level of recursion, it is storing a partitioning for each of the d dimensions.

$$i*sum(C) = 4*sum(C)$$

So, final total memory required in BUC is

$$N(T + 2p) + i*C_{max} + i*sum(C) \qquad <3.3>$$

- In case of BUC adaptation:

  Notations are identical with BUC.

  When a partition can fit in memory, counting sort in BUC adaptation has same memory requirement as that in BUC. Note that when a partition can't fit in memory, one more value array is needed which consists of N elements. Therefore in worst case, memory needed by BUC adaptation is:

  $$N * (T' + 2*p + i) = N(T'+12) \qquad <3.4>$$

  Compared with <3.1>, another difference is T'. The value of T' ranges from 0 to T. When partition is refined again and again (Fig. 7), the value of T' becomes smaller and smaller. This explains why BUC adaptation has more flexible memory

requirement than BUC and also guarantees that eventually a partition will fit in the main memory.

Because the BUC adaptation is implemented using dynamically allocated memory, therefore after a partition fits in memory, memory requirement becomes:

$$N * (T' + 2*p) = N(T'+8) \qquad <3.5>$$

T' here has same meaning as in <3.4>.

Memory used for counters is identical with that in BUC. So totally, BUC adaptation needs memory as follows:

When partition doesn't fit in memory:

$$N(T' + 2p + i) + i*C_{max} + i*sum(C) \qquad <3.5>$$

When partition fits in memory:

$$N(T' + 2p) + i*C_{max} + i*sum(C) \qquad <3.6>$$

Note:

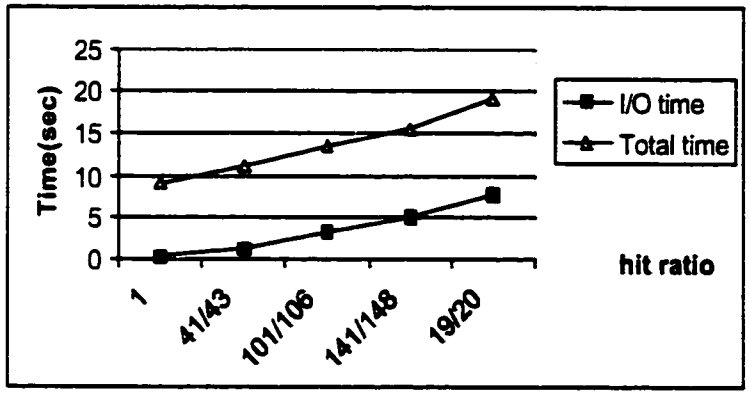
1) I do not count any "small" amounts of data like space for the current aggregation, or the recursive stack. I only count space that depends on the big variables: N and the Cs.]

2) Both BUC and BUC Adaptation invokes itself recursively, still T and T' are only allocated once (similar to global variable). Once allocated, all the following computation can take advantage of it.

3.1.3 Some Features

BUC Adaptation, overall, is an in-memory algorithm. It performances very well only when the whole database can fit in memory. If the database is too huge to fit in memory, partitions has to be read in memory one by one and generally each partition means one pass of the database. Here one pass means to rewind the file pointer once when accessing data in the disk. Times of pass is not a precise measure, but can reflect the around time used for disk access. My experiments show that when hit ratio decreases, the

performance of BUC Adaptation degrades terribly, as illustrated in Fig. 8. Here hit ratio *n/m* means BUC Adaptation tries to access data *m* times while *n* times finding data needed is already in memory (refer to line 1, 2 and 3 in Fig. 6). BUC Adaptation features this because it is a depth-first algorithm. Every time when BUC Adaptation processes a



transaction, only one dimension (attribute) is used. Some other algorithms, like minCirc

Figure 8 Hit ratio and performance

or Chunk Array try to retrieve as much information as possible from each transaction read in. Therefore, BUC Adaptation is doomed to have bad performance in disk-residential database.

Another feature of BUC Adaptation is about pruning. The pruning in BUC Adaptation is similar to the pruning in Apriori [1]. The major difference between Apriori (appropriately adapted for Iceberg-CUBE computation) and BUC Adaptation is that Apriori processes the lattice breadth first instead of depth first. Therefore BUC Adaptation is always one step slower than Apriori [6].

3.2 Chunk Array (CA) Adaptation

The basic idea of adapted chunk array (described in session 2) is to perform a search in the circumstance lattice from the strongest circumstances to the weaker ones. As soon as S (a given item-set) is found to be frequent in a circumstance, it is guaranteed to be minimal. Clearly, all minimal circumstances can be found in this way.

3.2.1 Pseudo Code of CA Adaptation

Algorithm Modified Chunk Array

Input: an item-set S, a support threshold s, and a transaction database D;

Output: the set of minimal circumstances in which S is frequent;

1. run chunk array algorithm, with the following modifications:

(a) no aggregate computation (other than verifying the support constraint) is needed;

(b) when processing a circumstance ø, compute the support of S in ø; if it exceeds s, add ø to the output; circumstances that are implied by ø are pruned;

Figure 9 Adaptation to Chunk Array Algorithm for Finding Minimal Circumstances.

The pseudo code here can be explained with an example. Assume database D consists of 3 dimensions *A*, *B* and *D*. The cardinality of each dimension is, let's say, 8. CA Adaptation will check strongest circumstances (length = 3) first. The number of all combination of 3-dimension circumstances is *8*8*8 = 512*. If one combination, say, *a2b3c2* is frequent, then all its subset, like *a2b3*, *b3c2*, can be pruned. CA Adaptation

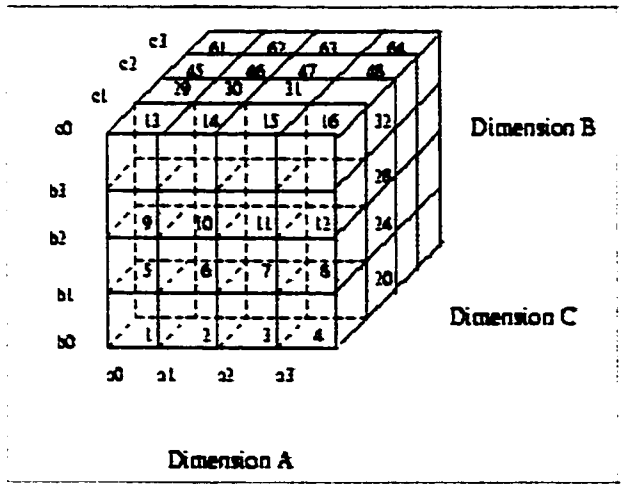


Figure 10 3-D array

then check next stronger circumstances till the weakest circumstances (length = 1).

3.2.2 Data Structure

In order to organize and store data efficiently, chunk cube [7] is used. Y. Zhao et al. in [7] proposed that if data is stored in an n-dimensional chunk cube (Fig. 10 here is an example of a 3-dimensional cube), the collapsing order is very import to memory requirement. Detailed can be referred in [7]. The following is the description of how to implement a chunk cube.

In the computation of CA Adaptation, a node tree (Fig. 11 is an example) will be constructed dynamically. Each node in the tree is a group-by. A node has data consisting of a chunk cube and pointers pointing to its child nodes. For example, in Fig 11, node *A* and *B* are child nodes of node *AB*. A node also contains several operations to process its chunk cube, like collapsing, applying and releasing chunks (memory is allocated and released dynamically with chunk as basic unit). All the data and operations are encapsulated as an object. Data flow from object to next lever objects along the lines in Fig 12. The number below the node denotes how many chunks needed by that node.

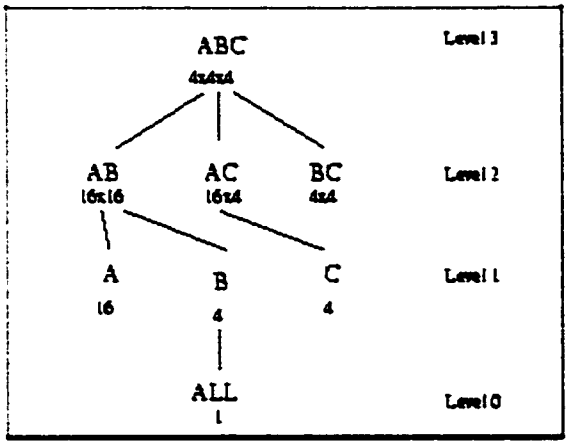


Figure 11 Collapsing order

Chunk is a basic unit for memory allocation. Each chunk is actually a small cube itself. Take the example mentioned in **3.2.1**, if a user prescribes that all dimensions (A, B and C) have 4 chunks, then the cube is exactly like the picture in Fig. 12. Each chunk has 2*2*2 = 8 item-sets or more particular, circumstances in circ. mining because cardinality of each dimension is 8.

The introduction of chunk is a trade-off between performance and memory requirement. From performance's viewpoint, it's desirable to have a huge n-dimensional cube in memory. If each transaction is imaged as a point, then it's easy to find a "square" in the cube to put the point on. However, the number of possible combination of all dimensions (square) is much much larger than the number of transaction in a database even database is huge and computer usually can't accommodates so many squares. So it's wanted to let one square to accommodate more "points" and allocate memory for square only when that square is needed. Therefore, the memory requirement can be reduced dramatically.

Size of a chunk ranges from 1 to cardinality and the selection of the size will affect the performance and memory requirement.

3.2.3 Some features of CA Adaptation

One advantage of CA Adaptation is that it can generate circumstance border directly; no post-processing is need. This feature is almost self-evident because CA is a top-down algorithm. In the processing of CA Adaptation, all circumstances implied are purged (see (b) in pseudo code for reference) in advance.

CA Adaptation is sensitive to the ordering of the dimensions. The reason is discussed partly in last section. Details can be got from [7]. The optimal dimension order Ó is (D1, D2, ..., Dn), where |D □ |D2 □ ... □|Dn|. Here, |Di| denotes cardinality of the

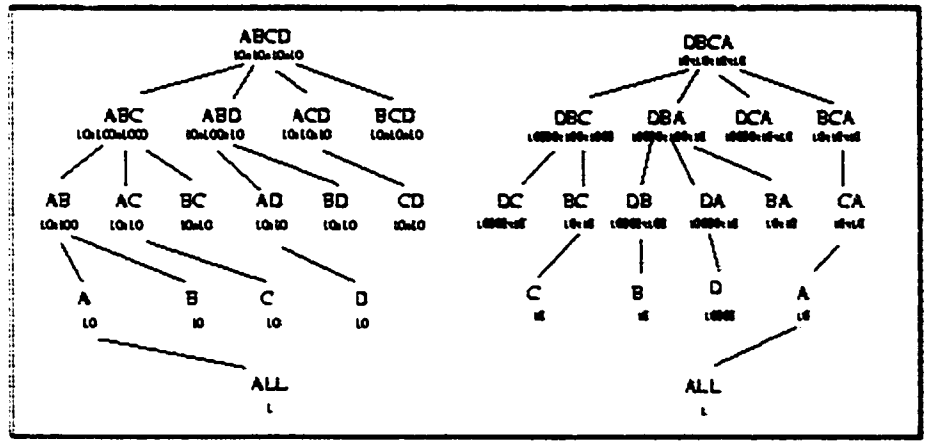


Figure 12 Mem requirements for different order

dimension Di. So the dimensions should be ordered incrementally in the dimension order

Ó. To illustrate this, A 4-dimension database is used. Assume the cardinality of dimensions A, B, C, and D are 10, 100, 1000, and 10000. The difference between dimension order (A, B, C, D) and (D, B, C, A) is shown in Fig. 12. The number below each node in the figure is the number of units of array element required by the node.

The last feature of CA Adaptation is its high memory requirement on system and the requirement increase exponentially with the number of dimension. Compared with BUC and minCirc, CA Adaptation consumes around 2 magnitudes more memory than the formers. Actually, in the experiments, it's found that when number of dimension is more than 5, the performance of CA Adaptation is very poor.

3.3 minCirc

The basic idea of minCirc is since frequency is c-monotone. Transactions can be scanned repeatedly to compute the support of S corresponding to all circumstances of a given length in each iteration, starting from length 1. When S is infrequent in a circumstance of a given length in a circumstance, all other circumstance starting with that circumstance can be pruned. For example, if *a1b1* is infrequent, then *a1b1c1* can be ignored.

This algorithm is in some sense inspired by Apriori.

3.3.1 From Apriori to minCirc

As mentioned before that the strength of Apriori is its pruning. Every time Apriori scans database, all candidates will be checked to see if they get minimum support. Before the next scan begins, new candidate set will be generated based on the current large item-sets. For example, *a3b2c5* is a candidate in the third pass if and only if *{a3b2, a3c5, b2c5}* all made minimum support in second pass.

The problem with using Apriori is, first, the candidate set usually cannot fit in memory because little pruning is expected during first few passes of the input; second, set operation is time consuming and difficult to implement.

Different algorithms address those problems using different methods. BUC, as mentioned early, is one step slow in pruning, but it trades pruning for locality of reference and

reduced memory requirements. minCirc, on the other hand, tries to keep pruning pace with Apriori and spends much effort on data structure to avoid set-operation which is time and memory consuming. Detailed description will follow.

Another point taken into consideration is the memory. Since BUC adaptation is kind of in-memory algorithm, minCirc is expected to emphasize on disk-residential database. As analyzed in **3.2.3**, it should take full advantage of each transaction in memory.

3.3.2 Pseudo Code of minCirc

Algorithm minCirc;

Input: an itemset S, a support threshold s, and a transaction database D;

Output: the set of minimal circumstances in which S is frequent;

1. choose some circumstance attribute order, say D1, D2,... Dn ;

2. scan D once and obtain counts of S in all atomic circumstances, i.e. in all D1 circumstances, ..., Dn circumstances;

3. let F 1 be the set of all atomic circumstances in which S is frequent;

4. hash the circumstances and write the counts;

5. create a linked list of candidates as follows:

(a) for (1 <= i <= n- i + +) {

let x_i F1 be any Di value;

create a linked list with head x_i;

for (i < j <= n- i + +) {

let x_j F1 be any Dj value;

append a node containing x_j to the linked list with head x_i;

} }

(b) k = 2;

(c) while the current set of linked lists is non-empty {

scan D and obtain counts of all k-circumstances appearing in the linked lists;

purge each node corresponding to a circumstance in which S is infrequent;

construct linked lists for (k + 1)-circumstances (candidates) as follows:

for each linked list L with head H of size k {

if (length(L) >= 2) {

for each node (containing a circ) x in L {

temporarily create a new list with head Hx, which contains all nodes of L after node x, that correspond to later circumstance attributes;

//since length(L) >= 2, there will be at least one //such node;

delete from this list those nodes y such that some k-subset of Hxy does not appear in any linked list with head size k;

//this test can be performed quickly by sorting;

} } } }

6. eliminate non-minimal circumstances via post-processing;

Figure 13 A Linked List Based Mining Algorithm for Finding Minimal Circumstances.

In Fig. 13, the following notations are used:

Suppose *D1,...,Dn* is a chosen order on the circumstance attributes and and let *c* and *d* be values of any circumstance attributes. Then $c \sqsubset d$ if and only if *c* is a *Di* value and *d* is a *D j* value, for some $i < j$; *c* and *d* are said to be incomparable otherwise. Sometimes it is convenient to write circumstances such as $A = a1 \ \& \ B = b1 \ \& \ C = c1$, simply as *a1b1c1*, relying on the fixed chosen order of circumstance attributes. Linked lists are used with heads and nodes (head is not a node). For a linked list, its length represents the number of nodes it has. Each list represents a set of candidate circumstances of a given length. E.g. a list with head *a1b1* and nodes *c1,.... c5, d2, d6, d7* (where $c \in dom(C)$ and $c \in dom(D)$) represents the set of circumstances *a1b1c1d2; a1b1c1d6, ..., a1b1c5d7.*

A circumstance $x_1, ..., x_k$ appears in a linked list provided its head contains $x_1, ..., x_{k-1}$ and one of its nodes contains the circumstance x_k. In this algorithm, a circumstance is referred as a frequent circumstance only when in which *S* is frequent.

3.3.3 Data Structure

All large item-sets and their candidates are stored in a multiply hashed data structure, as illustrated in Fig. 14. The first level hash stores the large item-sets, indexed with the first item (note that each item in minCirc has a unique number). For example, item-set *a3b5c1* is stored in the second bucket, assuming id of a3 is 2. The second level hash stores all candidates. Continued with the above example, let's say *a3b5c1d2*, *a3b5c1d8*, *a3b5c1e7* and *a3b5c1f3* are candidates, then large item-set *a3b5c1* has 4 buckets. *d2*, d8, *e7* and *f3* are stored in the 4 buckets with their corresponding counters. With this structure, large item-sets can be located quickly while keeping minimal memory requirement for the system. Every transaction, when read in, is first converted to a list of item id. With the first id in the list, a large-item bucket can be located quickly. Large item-sets in bucket are stored in lexical order, so binary search [12] can be applied here to efficiently find the candidates bucket. After that, it's a matter of math. When a pass of database finishes, the counters of all candidates are checked. If a candidate is frequent, a new entry in a suitable large item-sets bucket will be allocated and new candidates bucket will also be generated for it.

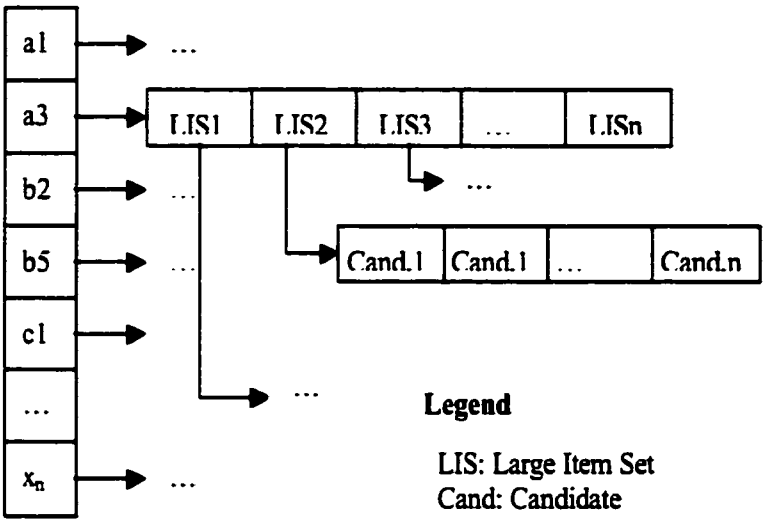


Figure 14 Data structure of minCirc

The hash data structure is implemented using array, list and pointers.

3.3.4 Some Features

Compared with BUC adaptation, minCirc can avoid too much redundant circumstances although it cannot generate minimal circumstance border directly. For example, assume

$\{a1b2\}$ is a large item-set. In CUBE algorithm, both $\{a1, a1b2\}$ will be output, but in minCirc, a large item-set won't be output until it can't "grow" anymore. Therefore, in

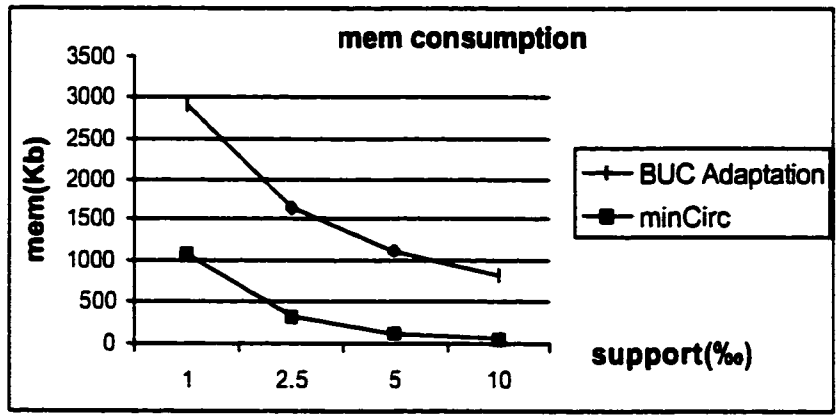


Figure 15 Memory consumption

this example, $\{a1\}$ can be removed and only $\{a1b2\}$ is output.

minCirc is also good at processing disk-residential database. Actually, even in worst case, the time of pass of the database is proportional to the length of the longest item-set, which is similar to Apriori. On the other hand the performance of minCirc degrades in case of skewed data. Simply because in skewed data, some large item-sets maybe extremely long and consume several more passes.

As to memory consumption, minCirc uses less memory, compared with BUC adaptation as showed in Fig. 15. One disadvantage of minCirc is that if the whole data structure (described before) does not fit in memory, the algorithm makes no provision for it.

3.4 Monotone and post processing

Post processing is proposed and implemented by Wang, Xiaohong with STL, which generates border from answer space, as depicted in Fig. 16.

Post processing follows the following facts: (i) a circumstance is never explicitly pruned unless S is infrequent in it; (ii) circumstances that are suffixes of those ending in the

last circumstance attribute value are never considered for counting, e.g., if circumstance attributes are ordered as *ABCD*, then there is no need to consider suffixes of $b1d1$; every

suffix of such circumstances are covered by some circumstances that already considered somewhere else; e.g., the suffix $bldl \square blcldl$, and one of its (not necessarily proper) prefixes, e.g., bl, is always considered by the algorithm.

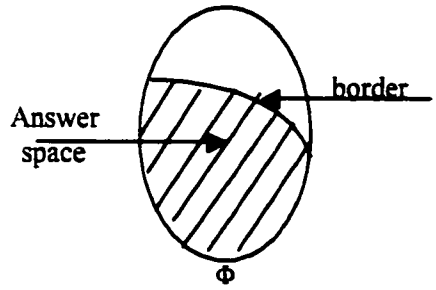


Figure 16 Circumstance lattice and border

4 Experimental Results

In the experiments, performances of all the 3 algorithms feeding with different data are recorded and compared. Performance are measured with requirement of peak memory and CPU time.

Note that here the peak memory instead of total memory allocation is used. In all 3 algorithms, most memory is applied and released from heap. Those operations can be caught by overriding function *new()* and *delete()*. By analyzing those operations, the high water mark of memory usage (peak memory) can be retrieved. It denotes real memory requirement.

4.1 Experiments

A Pentium II 200 processor with 64-KB memory is used for all the experiments. All the algorithms are implemented in C++ running on a Windows NT platform and all algorithms share same I/O module (described in chapter 2). Test material is synthetic sets of transactions generated for various cases. A Zipf distribution is used to control the skewness of the data, as commonly used in experiments of this kind, see e.g. [6]. To be fair to all the algorithms, both the cases where the transaction database fits in main memory and resides on disk are tested. Number of circumstance attributes ranges from 1 to 7. Each attribute has a domain of cardinality 100. The number of transactions varies form 10,000 to 500,000. The support threshold is from 5 to 1000. And skewness varies from 0, which is the uniform distribution, to 3. In each of the experiments, the number of candidate sets generated, the peak memory allocation, the total time, the I/O time, and the number of passes over the transaction database are recorded for analysis. However, for the sake of space and simplicity, not all of the experiment results are showed.

Note that here the peak memory instead of total memory allocation is used. In all 3 algorithms, most memory is applied and released from heap. Those operations can be caught by overriding function *new()* and *delete()*. By analyzing those operations, the high water mark of memory usage (peak memory) can be retrieved. It denotes real memory requirement.

4.2 Experimental evaluation

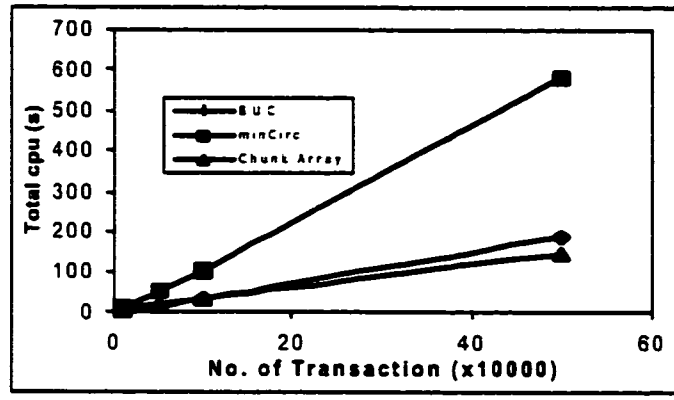


Figure 17 Algorithms Comparison (memory resident, skewness=1, threshold = 100, No. of Circ. = 5)

The first set of experiments is conducted where transaction databases can fit in main memory. All other parameters are varied as described above. The total CPU time is measured as a function of the number of transactions. We have chosen 5 circumstance attributes, threshold 100, and skewness 1, as a typical representative of this set of experiments. The results are shown in Fig. 17. It is fairly uniform. With the increase of the number of transactions, there is a linear increase in the total time for all three algorithms. But the slope for minCirc is steeper than the slope for BUC and Chunk Array. This is due to the fact that BUC and Chunk Array make only one pass over the transaction database, whereas minCirc makes up to as many passes as the number of circumstance attributes. Other results that have been observed from this set of experiments are: with the increase of support threshold, the number of candidates generated and the total time are decreasing for both BUC and minCirc. For Chunk Array, on the other hand, with the increase of the support threshold, the number of candidate sets generated is slightly increasing and the total time is almost constant. When skewness is increased, the total time for both BUC and minCirc is increasing while the total time for Chunk Array is decreasing. This is because Chunk Array works top-down, the number of atomic circumstances will dominate the number of candidate sets generated, and the total time will mainly be affected by the number of circumstance attributes and their cardinalities, once the number of transactions is fixed. As the data becomes more skewed,

the number of atomic circumstances that need to be considered decreases. The overall performance nevertheless only increases slightly.

The second set of experiments conducted is on disk resident transaction databases. When

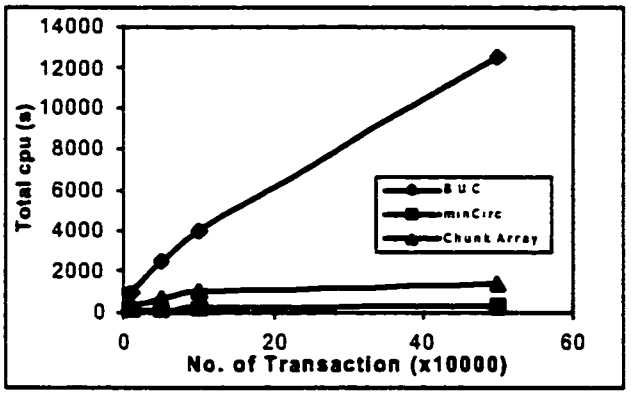


Figure 18 Algorithms Comparison (disk resident, skewness = 0, threshold = 100, No. of Circ. = 5)

the data is uniformly distributed (skewness = 0), minCirc gives the best performance among the three algorithms. Fig. 18 displays the results for support threshold at 100, and number of circumstance attributes at 5. With the number of transaction increases, the total time increases rapidly for BUC. When the number of transaction is equal to 500,000, the total time for BUC is 50 times larger than minCirc. Even for Chunk Array

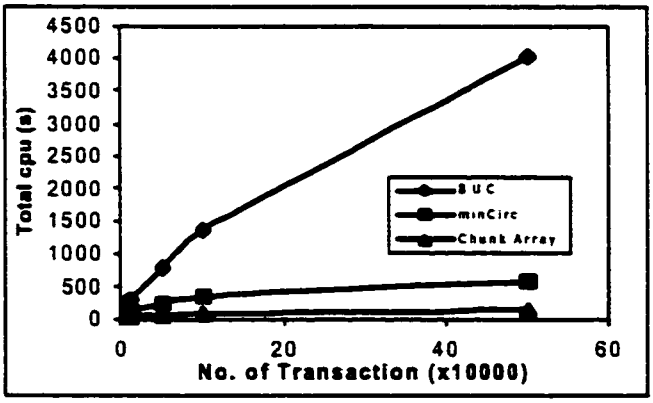


Figure 19 Algorithms Comparison (disk resident, skewness = 1, threshold = 100, No. of Circ. = 5)

the total time is 5 times bigger than minCirc. For minCirc, the upper bound of the number of passes over the transaction database equals to the number of circumstance attributes. On the other hand, for BUC the only upper bound is the size of the circumstance lattice. Furthermore, as the data is uniformly distributed, Chunk Array has to do much more computations for each of the atomic circumstance sets, so minCirc also performs better than Chunk Array.

However, with the increase of the skewness, as the data is not uniformly distributed, the performance of Chunk Array will improve quite drastically since some of the atomic circumstance sets will not appear in the transaction database. Fig. 19 gives the results for the same condition as Fig. 18, except that the skewness is equal to 1. Similar performance can be observed for BUC and minCirc. Under these conditions, Chunk Array is the algorithm of choice. Note that the time scale is different in Fig. 18 and Fig. 19.

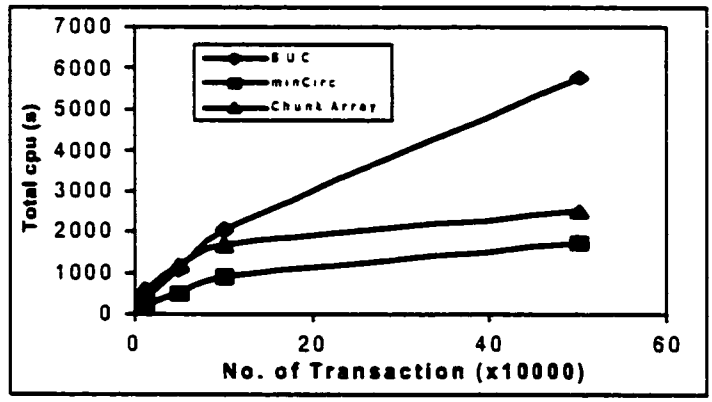


Figure 20 Algorithms Comparison (disk resident, skewness = 1, threshold = 100, No. of Circ. = 6)

Another factor that affects the performance of these algorithms, especially Chunk Array, is the number of circumstance attributes. When the number is 5, and the domain of each circumstance attribute has cardinality of 100, there is potentially 10^{10} atomic circumstance sets appearing in the transaction database. If increasing the number of circumstance attributes to 6, there might be up to 10^{12} atomic circumstance sets. This is a large increase for Chunk Array, but for minCirc this just implies at most one more pass over the transaction database. For these conditions minCirc becomes the best choice. Fig. 20 shows the results for 6 circumstance attributes. When the number of transactions is

smaller than 300,000, the total time for BUC and Chunk Array is similar, and both about twice the total time of minCirc. When the number of transactions increases to 500,000, the total time for minCirc is 5 times smaller than the other two algorithms.

So it is interesting to observe the effect of varying number of circumstance attributes. Fig.

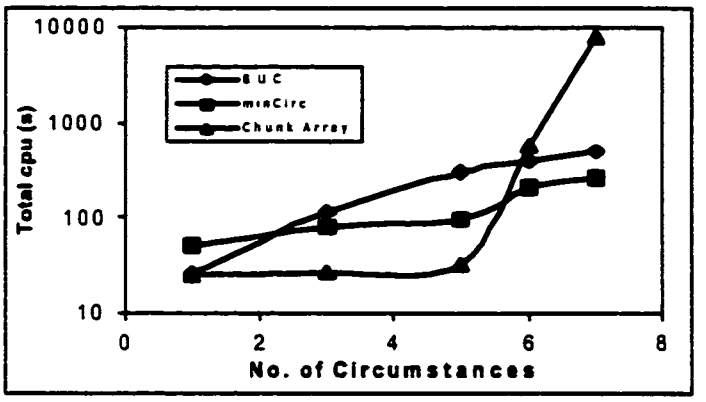


Figure 21 Algorithms Comparison (disk resident, skewness = 1, threshold = 100, No. of Trans. = 100000)

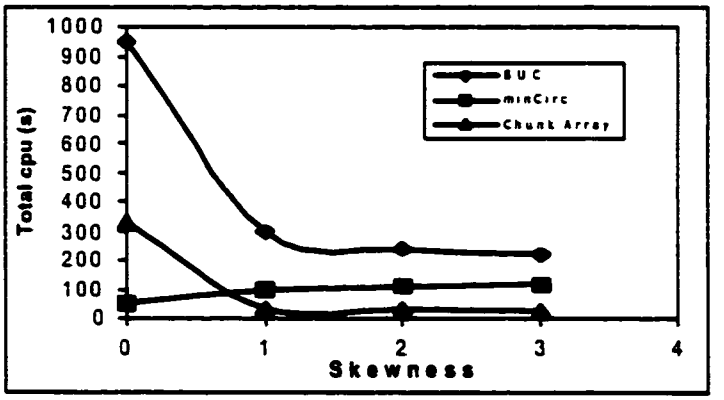


Figure 22 Algorithms Comparison (disk resident, No. of Circ. = 5, threshold = 100, No. of Trans = 100000)

21 shows such an experiment. When the number is smaller than 5, Chunk Array has the best performance and the slope of the curve is very gradual. However, once the number of circumstance attributes is greater than 6, there is a dramatic decrease in the performance of Chunk Array and it becomes the worst among the three algorithms. With the increase of the number of circumstance attributes, the total time for both BUC and minCirc increase much more slowly compare to Chunk Array.

In the third and last set of experiments the effect of the skewness of the data is tested. In Fig 22, when the data is uniformly distributed and is disk resident, minCirc has the best performance and BUC gives the worst performance. When skewness is greater than 1, the curve for all three algorithms become flat. Chunk Array shows the best performance

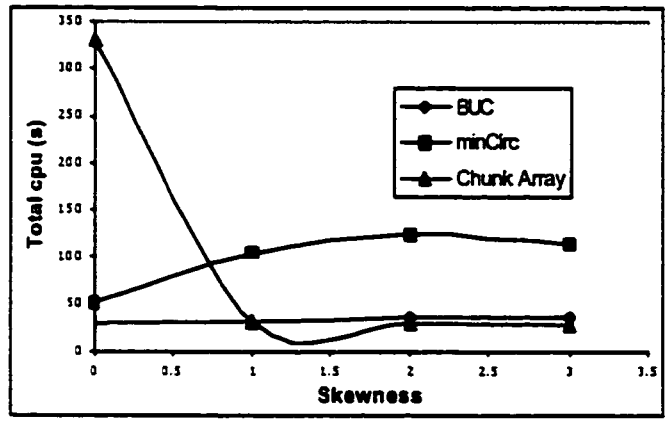


Figure 23 Algorithms Comparison (memory resident, No. of Circ. = 5, threshold = 100, No. of Trans = 100000)

while BUC is still the worst one. However, in another experiment, as showed in Figure 23, main memory-resident transaction databases is tested, while keeping the other parameters the same as in Fig. 22, it is observed that BUC almost unaffected by the

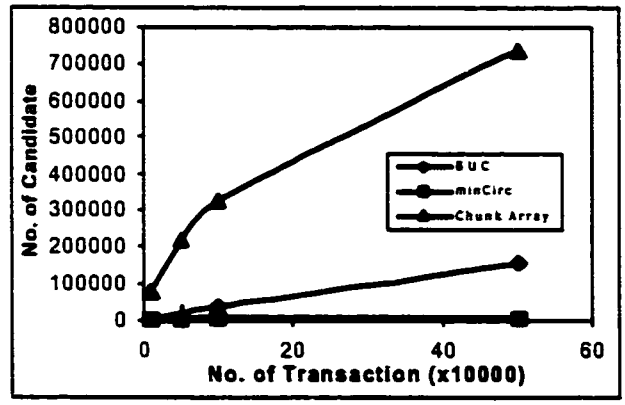


Figure 24 Algorithms Comparison (memory resident, skewness = 1, threshold = 1000, No. of Circ. = 5)

skewness of the data, and gives the best performance among the three algorithms.

Another experiments in this set. The number of candidate sets generated is tested. The residency of the transaction database does not affect this measure. The skewness of the

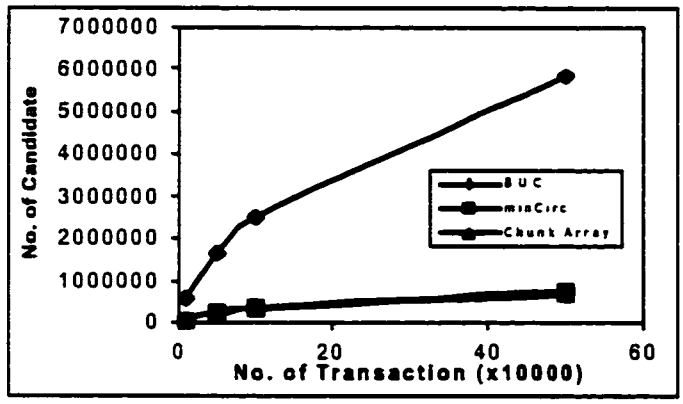


Figure 25 Algorithms Comparison (memory resident, skewness = 1, threshold = 5, No. of Circ. = 5)

data does affect the performance, but slightly. Two experiments, namely Fig. 24 and Fig. 25 are showed for the number of candidate sets generated vs. the number of transactions. Skewness is at 1, and the number of circumstance attributes is 5. Figure 24 shows the result for the threshold at 1000, and Figure 25 shows the result for the threshold at 5. When the threshold is high, the pruning strategies for BUC and minCirc are very effective, so at 500,000 transactions the number of candidate sets minCirc is over 500 times smaller than that of Chunk Array, and BUC generates 80 times fewer candidate sets than Chunk Array. However, when the threshold is small, the pruning strategy for BUC loses its efficiency, and the number of candidate sets becomes the biggest. minCirc also lose some of its pruning efficiency, but it still comparable to Chunk Array.

4.3 Summary

The prescriptions emerging from our experiments are described in the decision tree of Fig. 26.

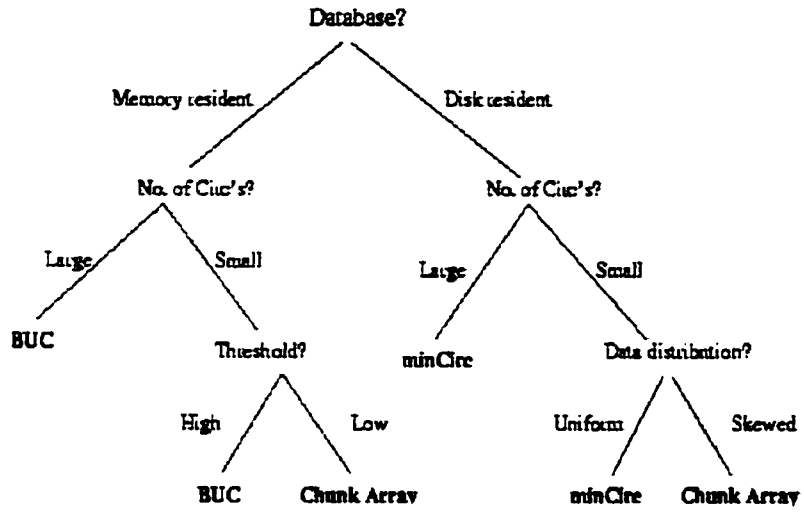


Figure 26 Decision Tree for Choosing a Mining Algorithm

Bibliography

[1] Rakesh Agrawal, Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994, pp. 487-499.

[2] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In Proc. AAAI'94 Workshop Knowledge Discovery in Database (KDD'94), Seattle, WA, July 1994, pp. 181-192,

[3] Laks V.S. Lakshmanan, Gosta Grahne, Xiaohong Wang, Ming Hao Xie. On Dual Mining: from Patterns to Circumstances, and Back..

[4] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, Alex Pang. Exploratory Mining and Pruning Optimizations of Constrained Associations Rules. SIGMOD 1998, pp. 13-24.

[5] J. Gray, Adam Bosworth, Andrew Layman, Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Proc. Of the 12th Int. Conf. On Data Engineering, pp 152-159, 1996.

[6] Kevin Beyer, Raghu Ramakrishnan. Bottom-up Computation of Sparse and Iceberg CUBEs. SIGMOD'99 Philadelphia PA. pp 359-370.

[7] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. Of the ACM SIGMOD Conf.*, pp 159-170, 1997

[8] R. Sedgewick. Algorithms in C, Chapter 8, page 112. Addison-Wesley Publishing Company, 1990.

[9] Web page http://www.umiacs.umd.edu/research/EXPAR/papers/3548/node8.html

[10] Kevin Beyer. Personal Communication, November 1999.

[11] Gosta Grahne, Laks V.S. Lakshmanan, Xiaohong Wang. Efficient Mining of Constrained Correlated Sets. IEEE Int. Conf. on Data Engineering (ICDE'00), 2000.

[12] Leen Ammerall. STL for C++ Programmers. John Wiley & Sons Ltd. 1997