산업기술연구(강원대학교 산업기술연구소 논문집), 제28권 B호, 2008.
Journal of Industrial Technology, Kangwon Natl. Univ., Korea, No. 28 B, 2008.

집합 값을 갖는 애트리뷰트에 대한 수직적으로 분할된 블록 중첩 루프 조인

Vertically Partitioned Block Nested Loop Join on Set-Valued Attributes

황 환 규*
Whang, Whan-Kyu

Abstract

Set-valued attributes appear in many applications to model complex objects occurring in the real world. One of the most important operations on set-valued attributes is the set join, because it provides a various method to express complex queries. Currently proposed set join algorithms are based on block nested loop join in which inverted files are partitioned horizontally into blocks. Evaluating these joins are expensive because they generate intermediate partial results severely and finally obtain the final results after merging partial results.

In this paper, we present an efficient processing of set join algorithm. We propose a new set join algorithm that vertically partitions inverted files into blocks, where each block fits in memory, and performs block nested loop join without producing intermediate results. Our experiments show that the vertical bitmap nested set join algorithm outperforms previously proposed set join algorithms.

키워드 : *집합 값 애트리뷰트, 역 파일, 비트맵, 블록 중첩 루프 조인, 겹침 조인*
Keywords : *set-valued attributes, inverted file, bitmap, Block Nested Loop Join, overlap join*

1. Introduction

As database systems are used in various applications, there is an increase in modeling real world complex objects with set-valued attributes [1,2,3,4]. Using set-valued attributes can be found in a variety of application domains such as goods purchased by customers, web pages visited by users, and keywords contained in documents.

* 강원대학교 IT 대학 교수, 공학박사

Current object-relational DBMSs support set-valued attributes in a relational table, but efficient support for joins on set-valued attributes is limited. In this paper, we study the efficient processing of set join operators. We assume that the set-valued attributes are stored together with other attributes in a single table. The set join operators include the set containment join, the set equality join and the set overlap join. Previous work on set joins has concentrated on containment join, but real-life queries in many applications require overlap join. As an example of a set overlap join, consider the join of a job table R with a job-hunting

table S such that R.required_skills ∩ S.skills ≥ T , where R.required_skills stores the required skills for each job and S.skills stores the skills of job-hunters. This query returns qualified job and job-hunter pairs that overlap in at least T skills for each job. In this paper we are concerned with efficient algorithms for efficiently processing of the set overlap join.

In the set containment join block-nested loop join method performs well, but in the set overlap join the method turns out to be bad. The reason is that join cost increases as the intermediate result size increases during join processing. This paper proposes a new vertical partition-based block nested loop join for set overlap join method, which is not required to generate intermediate results and thus turns out to be superior compared to previous block nested loop join.

This paper is organized as follows. Section 2 introduces the set index method using inverted files. Also, it describes the block nested loop join method. Section 3 proposes the improved block nested loop join. The performance of the algorithms is evaluated experimentally in Section 4. Finally Section 5 concludes the paper with future work.

2. Related Work

2.1 Inverted File

The inverted file is extensively used in information retrieval [5]. The inverted file stores the location of each item in a document and returns the location when a specific item is queried. This kind of characteristics of inverted file is used as index of set valued attributes. For each element in the domain D, where |D| assumes the cardinality of domain, the inverted file is created with the record ids of the sets that contain this element in sorted order. The inverted file is depicted in Figure 1. If we want to find tuples that contain the set elements {e_2, e_5, e_{10}} with T=3, we retrieve the rids of e_2, e_5, and e_{10} and intersect them. The result of intersection is as follows: S_1-2, S_3-3, S_4-1, S_5-1, S_6-1, S_9-3, S_{10}-1. The final answer is S_3 and S_9.

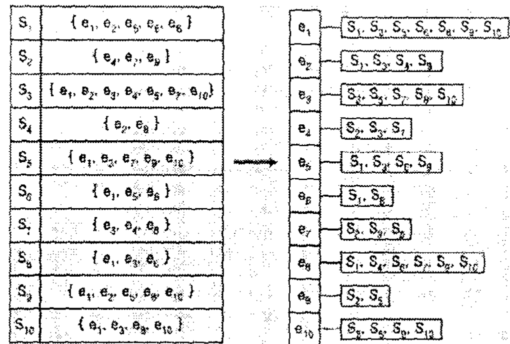


Fig 1: An Inverted File.

2.2 Block Nested Loop: BNL

2.2.1 Simple Block Nested Loop

A simple approach to do overlap join using an inverted file is to join with inverted file for each tuple. However, limited memory does not allow the whole size of inverted file to be on memory. If the inverted file does not fit in memory, the I/O cost is severe and in the worst case it has to be read once for each tuple in R. To overcome the problem, the block nested loop join is proposed [1].

2.2.2 Block Nested Loop Join

Since the cost to scan the whole inverted file for each tuple from R is expensive we partition the inverted file into blocks that fit in memory and scan R for each block. The Block Nested Loops (BNL) algorithm reads inverted file sequentially in blocks B_1, B_2,..., B_n, where each block fits in memory, as in Figure 2.

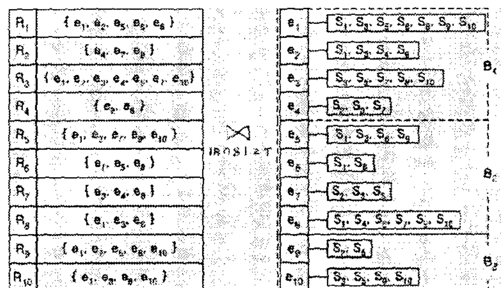


Fig 2: Partitioning Blocks for Nested Loop Join.

Assume that currently block B_i is loaded in memory, and let L_i be the set of elements whose inverted list is in B_i. Let R_i be each tuple of R and set of elements of R_i be $R_{i.s}$. For each R_i, three cases may occur.

1. $R_{i.s} \subseteq D_i$: the lists of all elements of $R_{i.s}$ are in B_i. In this case the lists are joined and the qualifying results are output.

2. $R_{i.s} \cap D_i = \varnothing$: the lists of no element in $R_{i.s}$ in B_i. In this case R_i is ignored and we go to the next tuple.

3. $R_{i.s} \cap D_i \neq \varnothing \wedge R_{i.s} \not\subseteq D_i$: the lists of some (but not all) elements in $R_{i.s}$ are in D_i. This is the hardest case to handle, since we may need information from other blocks in order to verify whether the cardinality of $S_{i.rid}$ is more than threshold T. We count $S_{i.rid}$ and consider two cases.

1) cardinality of $S_{i.rid} \geq T$: Since this case satisfies the overlap join condition, the result is stored in partial result file. For example, if B_i is in memory and $R_{i.s}$ is $\{e_2, e_3, e_1\}$, then S_3 is stored in partial result file since it overlaps 3 times.

2) cardinality of $S_{i.rid} < T$: Although current B_i does not satisfy the join condition, it may satisfy further operation. Hence the form of $<S_{i.rid}, n>$ is stored in temporary files, where n is the number of tuples from S that currently match with R_{rid}.

In the above, cases 1 and 2 do not generate problems because we can immediately get join results. Although case 3 with T=1 does not make temporary files, in case with T>1 we have to store the count in the form of $<R_{i.rid},: (S_{a.rid}, n1), (S_{b.rid}, n2),...>$ in the temporary files. where n1 and n2 represent the number of occurrences, which is less than the threshold T.

Temporary files and partial result files resulted from block join results are finally merged in order to give rise to final results.

3. Proposed Method

In this section, we propose a novel method that does not generate temporary files during set overlap join. In previous work, block layout for inverted file is organized horizontally, but in the proposed method, one block, which fits in memory, is organized vertically. Block is represented using bitmap. Since the vertical bitmap block representation can contain all the elements in a set, it does not generate temporary files. Bitmap is generated such that the bit positions corresponding to the given elements of the set S are set to '1'. For example, if relation cardinality is ten and one of the lists in the inverted file is $<s_1, s_3, s_7>$, then the list is represented by being set to '1' in the first, third and 7th in the bit string with 10 bits. Bitmap is constructed in the same way for all the lists in the inverted file.

Directory is organized by storing <e> in the proposed algorithm, unlike <e, offset> in the block nested loop join algorithm. Since each list requires same size of bit string, the information for the list position is not needed. As a result, we can save storage space. The reason for storing <e> is that some elements in the domain does not appear in the inverted file. For example, if the cardinality in the domain is ten and the elements appear only eight in the inverted file, then it is necessary to represent the eight elements that appear in the inverted file. The block organization of bitmap representation is depicted in Figure 3.

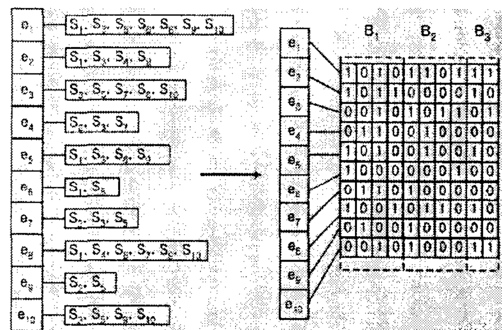


Fig 3: Bitmap Representation of Vertically Partitioned Blocks for Nested Loop Join.

The bitmap representation, called SBM, is constructed vertically. The join process in the proposed algorithm is performed as a block unit in the same way of block nested loop join algorithm. Vertical block nested loop join algorithm is given in <Algorithm 1>.

Algorithm Vertical_BNL(R, SBM, T) {

```
i = 0;
initialize partial result file F;
D = read directory in SBM;
if |Uel| - |D| ≥ |D| then { .................. (1)
   D' = Uel - D;
   keep D' in memory;
 }
 else                        .................... (2)
   keep D in memory;
 while there are more block in SBM {
   i = i + 1;
   Bi = read next block of SBM that fits
        in memory;
   Initialize partial result file Pi;
   for each tuple tR ∈ R do {
     if D is in memory then   ............. (3)
        tR.set = tR.set ∩ D;
     else if D' is in memory then ....... (4)
        tR.set = tR.set ∩ (Uel - D');
     for each bit string s ∈ Bi do {
       compare s and tR.set;     ........... (5)
       if tS.rid appears at least T times
       then
         append <tR.rid, tS.rid> to Pi;.. (6)
     }
   }
 }
 union all Pi, 1≤ i ≤ n to produce F;.. (7)
}
```

Algorithm 1: Vertically Partitioned Block Nested Loop Join.

The algorithm proceeds as follows. First, it reads the directory of the bitmap and computes the difference between the number of all the elements in the domain (|Uel|) and the number of the elements stored in the directory (|D|). The difference represented by D' means that the elements are in the domain but not in the directory. To save the memory space, we keep in memory the smaller one between D and D'. Next, block is read sequentially from bitmap file SBM. Each tuple in R is joined with each block. In this process, among the elements set of R ($t_{R.set}$), the elements that are not included in the bitmap are excluded (3)-(4). For the elements of each tuple in R, we count the number of the bit that is set to '1' in the vertical bitmap of S. If this number is larger or equal to the threshold, the pair <$t_{R.rid}$, $t_{S.rid}$> is stored in a partial temporary result file (5)-(6). After join operation of all the blocks, the final result is obtained by union of partial result (7).

4. Performance Results

The experiments were conducted under Linux kernel 2.6.15-1 with Pentium 4 1.8 GHz processor and 512 MB memory.

4.1 Data Generation

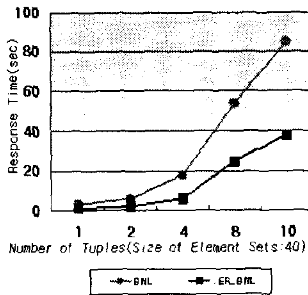
Data sets used in the experiments were generated synthetically. The parameters of input data sets are the average set size of each tuple, the domain size of set elements, and a correlation ratio. The domain size |D| is represented by the set of integers ranged from 1 to |D|. Set values represented by integers can simplify real applications in terms of reducing the space requirements of the tuples and the comparison costs. Each tuple has a set composed of integers picked from domain. In order for each tuple not to have skewed elements in a specific range, the domain is split into 50 equal-sized sub-domains. Elements in the same sub-domain represent correlated sub-elements.

The correlation ratio is used to represent the number of elements in a set which are correlated. The elements of a set in a tuple are generated as follows. For each tuple, we pick a sub-domain randomly from the 50 sub-domains. Then the set elements are chosen from this sub-domain as much as the correlation ratio and the rest of them are randomly chosen from the remaining 49 sub-domains. For example, if the domain size is 1,000, the average set size is 20, and the correlation ratio is 10%, then 2 elements, which are the correlation ratio of the set size, are chosen from a randomly chosen sub-domain out of the 50 sub-domains, and 18 elements are randomly chosen from the remaining 49 sub-domains. In the experiments, the relations R and S were generated to have the same cardinality, the domain size |D| is set to 1,000, and the correlation is set to 10%.

4.2 Results Analysis

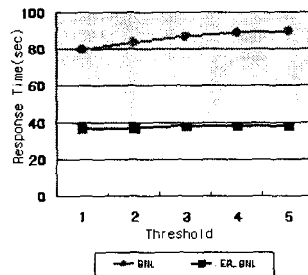
In Experiment 1, we compare the running time of each algorithm when we increase the

relation cardinality. The experimental settings are as follows: the average set cardinality is set to 40, the relation cardinality ranges from 1K to 10K, the memory buffer is set to 10% of the size of a relation on disk, and the threshold is set to 3. Experiment 1 shows that the proposed vertical BNL outperforms BNL as much as 2 orders of magnitude. This is due to the fact that the vertical BNL does not require any additional operations from join results. On the other hand, in BNL the inverted file size increases when the relation size increases. Thus the block size increases, the temporary files become increase and the I/O cost increases.



Experiment 1: Effect of Relation Size.

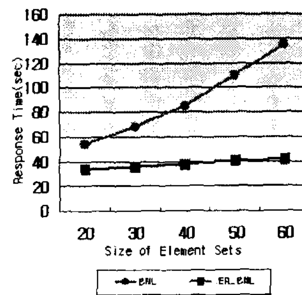
Experiment 2 compares the response time under various threshold values. The relation cardinality is set to 10,000, the number of set elements is set to 40, and the memory buffer is set to 10% of the size of a relation.



Experiment 2: Effect of Threshold.

The figure shows that the running time of BNL increases as the threshold values are increased. The reason is that as the threshold values increases, temporary files become large due to increase of the condition range. On the other hand, since the vertical BNL does not produce the temporary files, it is not affected due to the varying of the threshold values.

Experiment 3 compares the response time under various elements set size. The experimental settings are same as Experiment 2 with threshold being set to 3. Experiment 3 shows that the proposed vertical BNL outperforms BNL as much as 3 to 4 orders of magnitude.



Experiment 3: Effect of Element Set Size.

5. Conclusions and Future Work

In the previous study, BNL outperforms a signature-based approach for set-valued containment joins, typically by an order of magnitude. However, BNL does not show a good performance on set-valued overlap joins. In this paper, we have proposed the vertical BNL method to reduce the intermediate result sizes resulted from the join processing of BNL. The vertical BNL method which produces bit-mapped representation of the inverted file and a vertical representation of blocks, reduces the size of the previous inverted file approach and does not produce intermediate results due to the vertical bit-map representation. In consequence, we can achieve a good performance on the vertical BNL method compared to the previously proposed method.

In the future, we plan to study the effects according to the varying size of domain in set elements. Another study is to improve performance on compression of bit map and on indexing. We plan to study the efficient processing of various set join predicates such as

containment and equality on set-valued attributes.

참 고 문 헌

[1] Nikos Mamoulis , "Efficient Processing of Joins on Set-valued Attributes", *In Proc. 2003 ACM SIGMOD In Conf. Management of data,* California, pp. 157-168, June, 2003.

[2] Mikolaj Morzy, Tadeusz Morzy, Alexandros Nanopoulos and Yannis Manolopoulos, "Hierachical Bitmap Index: An Efficient and Scalable Indexing Technique for Set-Valued Attrubutes", L. Kalinichenko et al. (Eds) : *ADBIS 2003, LNCS 2798*, pp. 236-252, 2003.

[3] Sunita Sarawagi and Alok Kirpal, "Efficient set joins on similarity predicates", *In Proc. SIGMOD 2004 In Conf. Management of data, France*, pp. 743-754, June 2004.

[4] Yangjun Chen and Yibin Chen, "On the Sign- ature Tree Construction and Analysis", *IEEE Transactions on Knowledge and Data Engineering*, Vol 18, No. 9, pp. 1207-1224, 2006.

[5] J. Zobel, A. Moffat, and K. Ramamohanarao, "Inverted Files versus Signature Files for Text Indexing," *TODS*, 23(4), pp. 453-490, 1998.