Performance Evaluation of KD-Join Algorithm

3N-10

Lilian HARADA, Masaru KITSUREGAWA, Mikio TAKAGI

Institute of Industrial Science
University of Tokyo

1. Introduction

In previous papers [1,2] we have introduced a join algorithm for very large relations indexed by KD-trees (KD-join algorithm) and shown its analytical evaluation and simulation results. This paper presents the implementation and performance evaluation of the KD-join algorithm. We show detailed measurement results of the implementation on top of SUN4 with analysis of each of the cost components. We show that with an efficient implementation, very large relations can be joined with the minimum I/O cost of one scan for each relation, which is the lower bound.

2. KD-Join Algorithm

In the KD-join algorithm we introduce two new concepts: the *wave* and the *join range*. In the KD-join algorithm, the join of two relations R and S is divided in steps. The *wave* is the set of pages of relation R loaded in main memory in each step, which is the unit of processing in each step. The *join range* is the range of the join attribute value to be processed in each step, which is derived from the wave. The basic idea of the algorithm is to load a wave in the main memory, and based on this wave, determine the join range of this step. Then, in the remaining space of the memory, all the pages of relation S whose join attribute value are in the join range are loaded and the join processing of data on memory is done. In an ordinary strategy, because data is loaded from disk in page units, the main memory is managed at page level. However, in order to increase the effective space of main memory in each step and allow the loading of more tuples, a garbage collection mechanism which dynamically discards the tuples that are no longer necessary after the processing of each step is used. Fig.1 shows an example of the processing overview of the KD-join algorithm. The figure illustrates the tuples of both relations on main memory in each step. After processing the join range in a step, the effective memory space for the next step is enlarged if the dotted portion is discarded as garbage and only the dashed portion is maintained in main memory. In [2] we analyzed the KD-join algorithm and showed that it performed the join with one scan, that is, the minimum number of page accesses.

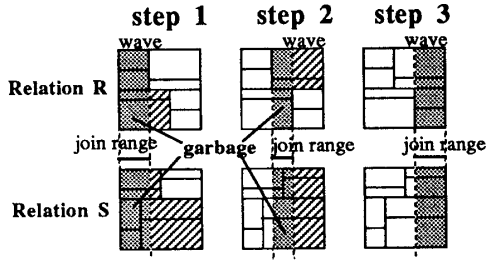

Fig.1 Processing Overview of KD-Join Algorithm

3. Implementation of KD-Join Algorithm

In order to take advantage of the order-preservance presented by KD-tree indexed relations, we choose the sort-merge algorithm to join the data on memory. In Table 1 we show the implementation procedure of the KD-join algorithm.

Because in the KD-join algorithm unnecessary tuples are dynamically discarded from main memory, the memory management is not at page level as in ordinary systems, but at tuple level. The memory management at tuple level has to be implemented efficiently so that the increase of main memory space compensates for its cost. In the main memory, both relations R and S are managed as linked lists of tuples. Let us see how these two linked lists are joined by using the sort-merge algorithm. First, the KD-tree index of relation R is searched (index search phase - R) to determine the wave and the join range of the step. Then, each page of relation R is loaded into an input buffer (disk read phase - R) and sorted there, before transferred to the main memory. Thus, besides buffering the I/O page, this buffer is also used as a pre-sorting area. Then, the tuples in the input buffer are transferred to the main memory tuple by tuple (move phase - R), constructing the linked list of relation R, in ascending order of join attribute value (sort phase - R). Concerning relation S, its linked list is constructed in the same way. Thus, first the KD-tree index of relation S is searched (index search phase - S) and the pages whose join attribute value are within the join range are determined. Then, each page is loaded into the input buffer (disk read phase - S) and when transferring each tuple to a free tuple space in main memory (move phase - S), the linked list is constructed in ascending order of join attribute value (sort phase - S). Finally, the linked lists of relations R and S are merged (join phase). Because we are using the sort-merge algorithm to process the data on memory, and because the data are organized as linked lists of tuples in sorted order, unnecessary tuples are dynamically discarded simply by unlinking the top tuples in the lists of both relations (discard phase - R and S). In this way, we efficiently implement the KD-join algorithm with a dynamic garbage collection mechanism, by utilizing an appropriate data processing and data management methods.

```
For each join step
    index search phase for relation R
      /* determine wave of relation R and join range */
    disk read phase for relation R
      /* load the pages of wave which are not loaded in
         main memory yet into the input buffer */
    move phase for relation R
      /* move each tuple of relation R from input buffer to
         main memory */
    sort phase for relation R
      /* sort tuples of wave according to join attribute value */
    index search phase for relation S
      /* determine pages of relation S whose join attribute value are
         within join range and are not loaded in main memory yet */
    While there are pages of relation S to load
        If there is no free space in main memory
            discard phase for relation S
              /* unload one page of relation S */
        disk read phase for relation S
          /* load each page of relation S into the input buffer */
        move phase for relation S
          /* move each tuple of relation S from input buffer
             to main memory */
        sort phase for relation S
          /* sort tuples of each page of relation S according
             to join attribute value */
        join phase
          /* join tuples of wave and each page of relation S
             whose join attribute value are within join range */
        discard phase for relation S
          /* unload already processed tuples of relation S */
    discard phase for relation R
      /* unload already processed tuples of wave */
```

Table 1. Implementation Procedure of KD-Join Algorithm

4. Performance Evaluation of KD-Join Algorithm

4.1. Evaluation Environment

To precisely evaluate the KD-join algorithm and to measure the CPU and I/O utilizations, we implemented it on SUN4 with Fujitsu M2382K disk. In the following, we present measurement results when the benchmark relations consist of 64K tuples each, the join attribute is a 4-byte

integer, and each tuple is 128 bytes long. The page size is 2K bytes. The relations are indexed by a 2-dimensional KD-tree and the data has an uniform distribution.

4.2. Measurement Results

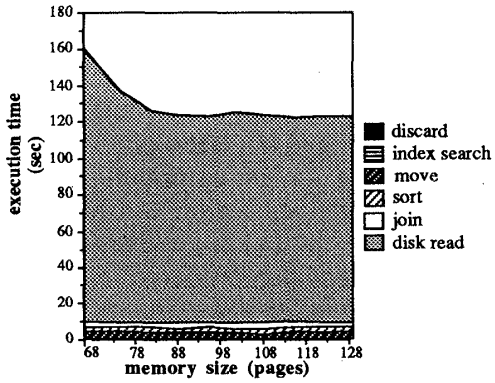

Fig.2 Execution Time for KD-Join Algorithm

In Fig.2, the execution time and its components are shown as a function of the amount of available memory space. As expected, the execution time decreases when the main memory size is enlarged. We can see that the discard phase cost is negligible and that the index search phase, the move phase, the sort phase and the join phase costs are very small and invariable with the main memory size. The disk read phase cost decreases when the main memory is enlarged. For a large range of the main memory considered, the number of page accesses reduces to one scan for each relation. When the available memory space is not enough to hold all the necessary tuples of relation S in each step, some pages of relation S are read into the input buffer more than once, causing a higher disk read cost. However, as we have already observed, the move phase, the sort phase and the join phase costs are invariable with the main memory size. This is because only the tuples which have not been processed yet are transferred from the input buffer to the main memory. Thus, the strategy to maintain only the necessary tuples in the memory, besides increasing the efficiency of the memory, also avoids unnecessary movement and processing of data in memory. As clearly observed from Fig.2, almost all the execution time is represented by the disk read time, which is the cost to scan each relation only once. However we find that an efficient implementation of the memory management at tuple level with garbage collection mechanism requires a very small CPU cost, compared with the resulting I/O cost savings.

5. Effects of Access Dimension

The KD-join algorithm itself does not depend on the access dimension, that is, which indexed attribute to join. At the logical level of the algorithm, the I/O cost is the same for any attribute. However, although not evident at the logical level, the access dimension is a physical parameter that influences the performance. The disk read time is affected, since it is greatly dependent on the physical disposition of the pages in the disk.

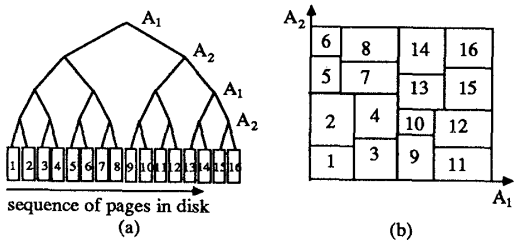

Fig.3 An Example of KD-tree (K = 2)
(a) KD-tree index
(b) Space Partitioning of KD-Tree Indexed Relation

Fig.3(a) shows an example of a 2-dimensional KD-tree index, and the corresponding data pages in the disk. The data pages are stored in the disk, in the sequential numbered order. Fig.3(b) shows the space of the relation, and the corresponding subspaces of its data pages. It can be observed that, concerning attribute a_1, the first wave is composed of pages 1, 2, 5 and 6. On the other hand, the first wave on attribute a_2 is composed of pages 1, 3, 9 and 11. Thus, the pages of a wave on attribute a_1 are sequential in the disk in pairs, while for attribute a_2, they are all non-sequential.

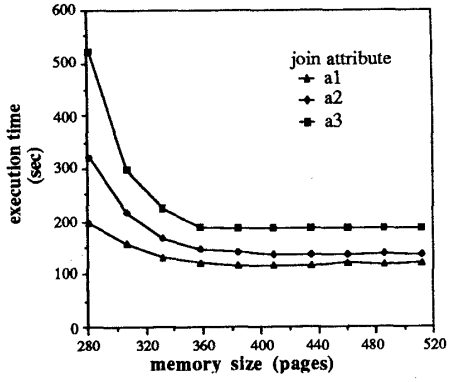

Fig.4 Execution Time Varying Access Dimension

Fig.4 shows the execution time to join two relations of 64K tuples each, with a 3-dimensional KD-tree index on attributes a_1, a_2 and a_3. The tuple length is 128 bytes, and the page size is 2K bytes. We can observe that the execution time is not the same for any join attribute and increases in the order of join attribute a_1, a_2 to a_3. For a 3-dimensional KD-tree, a wave on attribute a_1 contains 4 pages which are sequential in the disk, a wave on attribute a_2 contains pairs of pages which are sequential, and a wave on attribute a_3 contains all pages in a non-sequential order in the disk. Although the number of page accesses is the same for all the three attributes, we find that the performance is better for attribute a_1. This is because the disk controller has a buffer and the attribute a_1, whose pages are accessed sequentially, is the most privilegied by this buffering. The least favored is attribute a_3. Thus, we observe that although at the logical level the I/O cost is equivalent for any join attribute, the disk read time and the total performance are affected by the effect of buffering in the disk controller.

6. Conclusions

In this paper we present details of the implementation of the KD-join algorithm and its performance measurements. The memory management at tuple level with a garbage collection mechanism is more elaborated than the conventional management at page level. However, we found that with a good implementation of these mechanisms, the implied overhead was very small in comparison with the disk read time, and the cost of join operation was completely I/O bound. Therefore, the KD-join algorithm was shown to be very efficient since it reduces the total execution time to one scan I/O cost. Finally we present measurement results of the access dimension and show the performance sensitivity to such physical parameter.

We are now planning to study the data clustering in parallel disk modules, in order to further reduce the I/O cost of the KD-join algorithm.

[References]

[1] L.Harada, M.Kitsuregawa, M.Takagi, "Join Strategies on Multi-Dimensional Clustered Relations", 第37回情報処理全国大会7Q－7, 1988

[2] M.Kitsuregawa, L.Harada, M.Takagi, "Join Strategies on KD-Tree Indexed Relations", Proc. of the 5th Int. Conf. on Data Engineering, 1989