IADIS International Journal on Computer Science and Information Systems
Vol. 2, No. 1, pp. 33-47
ISSN: 1646-3692

QUERY PROCESSING AND ACCESS PATH SELECTION IN THE RELATIONAL MAIN MEMORY DATABASE USING A FUNCTIONAL MEMORY SYSTEM

Jun Miyazaki *Graduate School of Information Science, Nara Institute of Science and Technology, Keihanna Science City, Ikoma, Nara 630-0192, Japan*

ABSTRACT

In this paper, we describe heuristic rules for choosing an appropriate access path for a given query in main memory databases using a functional memory system. We have asserted that conventional cache-based memory access is not suitable for query processing in main memory databases. To cope with this problem, we have proposed three hardware-supported memory access schemes. If we select an appropriate memory access scheme from the three proposed schemes and the cache-based access, the execution time of query processing can be reduced. In this paper, we derive heuristic rules on which access path should be selected and show its impact on execution time through experimentation.

KEYWORDS

main memory database, access path selection, query processing, memory system

1. INTRODUCTION

Database systems are crucial infrastructures of the recent IT society. As the amount of data stored in databases rapidly increases, the processing time of database queries increases as well. However, users continue to demand higher speeds for complex queries such as decision support and data mining. Even simple queries are required to be executed more quickly in the context of stream and/or sensor database processing.

Since most conventional databases store data on hard disks whose storage cost per bit is inexpensive, a large amount of data can be handled at a limited cost. Unfortunately, the I/O access time of disks is quite slow compared to memory access time. Therefore, optimizing I/O costs to reduce query-processing time is regarded as the most important issue in conventional database studies. In other words, researchers have not focused on exploiting CPU cycles.

On the other hand, due to rapid semiconductor technology developments, the cost of semiconductor memory, in particular dynamic random access memory (DRAM), is drastically decreasing, while its capacity is increasing. DRAM is now widely used as main memory in modern computer systems. Therefore, main memory databases (MMDBs) (Garcia-Molina, 1992), which enable high-speed query processing, can be achieved. However, compared to processors, the development of DRAM's access speed over the past decades has resulted in almost no progress, which leads to an increase in the access latency gap between processor and memory. Such a latency issue is generally called the *memory wall problem*, which also affects database operations (Ailamaki et al., 1999; Ailamaki et al., 2001; Boncz et al., 1999).

To cope with the essential memory wall problem for MMDBs, we have proposed three hardware supported memory access methods, in particular, for simple systems such as PCs and embedded systems that do not have a high degree of memory interleaving. However, we have not discussed which memory access method should be selected for a given query. This is known as the access path selection problem in query optimization.

In this paper, we discuss heuristic rules for access path selection for three memory access methods using a functional memory system that we have proposed, as well as a conventional cache-based one. Since each of them has advantages and disadvantages, if we can optimize a given query, i.e., an appropriate choice from available memory access methods, query-processing performance is maximized. We also show some experimental results to confirm the effect of the heuristic rules.

2. MAIN MEMORY DATABASE AND MEMORY WALL

2.1 Main Memory Database

Main memory databases (MMDBs) are suitable for applications requiring faster database processing. Unfortunately, because use of DRAM as main memory has been expensive, MMDB has only been applied for limited purposes. However, due to recent low cost and large-sized DRAM, a high performance MMDB on inexpensive PCs, even on embedded systems, can be achieved. For example, techniques of MMDBs are expected to be applied to systems that deal with stream and/or sensor data because such systems must quickly process such data on the fly.

MMDBs provide a physical design advantage for databases. They can use a more flexible data structure to store data in main memory than conventional disk resident databases that need to consider disk properties. The key idea of the data structure for MMDBs is to separate variable-sized objects from fixed-sized ones (Lehman et al., 1992; Jagadish et al., 1994). The variable-sized objects are typically referenced from fixed-sized ones by pointers. For example, a relational database table can be organized as illustrated in Fig. 1, which is a variation of the separated data structure scheme. In this data structure, the size of each attribute in a tuple is forced to be fixed, e.g., 32 bits for a 32-bit processor, so that the processor can efficiently handle each datum as a word, e.g., primitive data. If the size of an object of an attribute is more than that of a word, e.g., a string, the attribute stores a pointer that designates the object. Such a large object is stored in other memory space. Since a tuple is regarded as a unit of data in a relational data model, tuples are usually allocated in a continuous memory address, which is called the N-ary storage model (Gray et al., 1993).

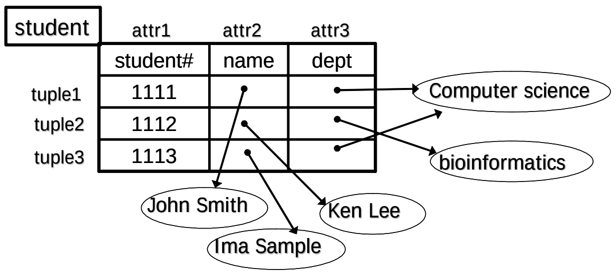

Figure 1. Example of data structure for MMDBs

Database operations do not always access data contiguously in a table, e.g., examining an attribute. This means that cache-based memory access, which is based on the assumption that data to be accessed are closely allocated in address space, is not efficient because the *load* instruction of modern processors fetches multiple words at continuous addresses associated with the size of a cache line, e.g., four or eight words at a time, even though only one necessary datum is included in the cache line. Therefore, inefficient data transfers from main memory take place during database operations. These data transfers in modern computers greatly deteriorate the performance because the speed of accessing main memory becomes much slower than that of processors, which is known as the *memory wall problem*. To cope with it, cache-conscious index structures have been proposed (Rao et al., 2000; Chen et al., 2001; Takamizawa et al., 2006). However, they do not consider general query processing.

2.2 Memory Wall

Over the last decade, the clock speed of modern processors has been greatly improved, though that of DRAM has changed less. The speed of accessing main memory becomes much slower than the processors, which is known as the *memory wall problem*. To cope with it, cache-conscious data structures and effective data prefetching have been proposed (Rao et al., 2000; Kim et al., 2001; Chen et al., 2001). However, in queries to large databases, cache memory has little effect.

In addition, scanning attributes in selection and join operations do not conform to cache-based normal memory access in modern computer architecture because the *load* instruction of modern processors fetches multiple words at continuous addresses associated with the size of a cache line, e.g., four or eight words at a time, even though only one necessary datum is included in the cache line.

Since the clock speed outside the CPU, called system bus cycle, is tens or hundreds of times slower than the CPU, unnecessary data retrieval only results in poor CPU utilization because main memory also works at the system bus speed.

3. EFFICIENT MEMORY ACCESS FOR MMDBS

3.1 Main Memory and Its Access

Main memory is usually configured with memory modules in which multiple DRAMs are installed, and works at the speed of the system bus, which is 5~10 times slower than that of CPU. Since the properties of a memory module are essentially identical to those of DRAM, the memory module is modeled as shown in Fig. 2. When a physical memory address is given, first a bank address is specified in which the requested data exist. Then, a row address followed by a column address is provided. After giving the column address, the data go to the I/O buffer after some system bus cycles, called CAS latency. Then, the data start to be transferred. The rest of the data to fill a cache line are continuously sent to the CPU. However, if a set of requested data belongs to the same row, it is possible to read the data by giving only the corresponding column addresses.

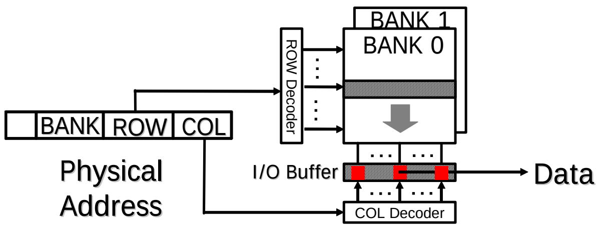

Figure 2. Model of DRAM

If column addresses are continuously provided, each piece of data can be read in every system bus cycle in a pipeline manner. This memory access method is called *page mode* access. Although SDRAM and older products were equipped with page mode, it seems to no longer be available in current DDR and DDR2 SDRAMs because they are specialized for cache-based access. However, it is still possible to implement page mode access even in current DRAMs.

Recall the data structure of the MMDB in Fig. 1 and its data access patterns, for example, scanning an attribute. The data required for database operations are located discontinuously. This means that a CPU cannot effectively retrieve the requested data because unnecessary data are also retrieved by the cache-based memory access in modern processors.

3.2 Efficient Memory Access Schemes with a Functional Memory System

In order to enable efficient data access in discontinuous address space, we have proposed three memory access schemes using a functional memory system: *stride data transfer* (SDT), *bitmap-based data transfer* (BDT) (Miyazaki, 2005), and *Comparator-based Data Transfer* (CMP) (Miyazaki, 2006). The essential idea of these memory access methods is to avoid transferring unnecessary data for computation over the low speed of the DRAM and system bus as shown in Fig. 3. This results in reduction of the overhead between the CPU and main memory. The functional memory system that we have proposed consists of an intelligent memory controller that translates discontinuous addresses to column addresses for DRAM and memory modules that support simple comparison operations.

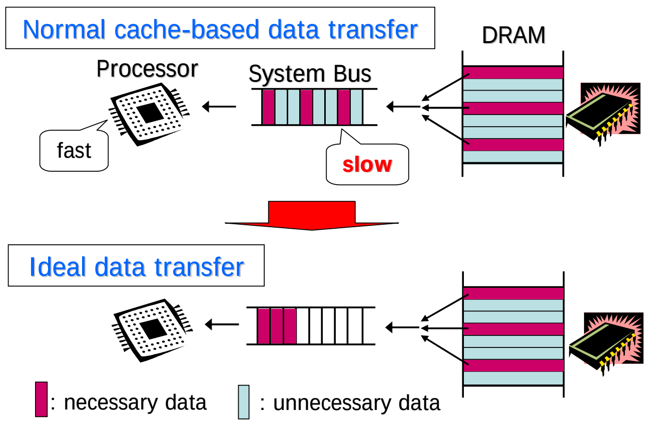

Figure 3. Conventional and ideal data transfer

Stride data transfer (SDT) contributes to retrieving multiple data at fixed stride addresses. In this memory access model, the CPU has only to send the first address of data and the stride to the memory controller. The memory controller can easily calculate the following column addresses for DRAM with the stride information. The column addresses are provided continuously to DRAM in page mode, and then, the requested data are retrieved in a pipeline fashion (see Fig. 4, left).

SDT can efficiently handle only the data at a fixed stride. However, necessary data are not always stored at the fixed stride addresses, for example, in a case in which two or more attributes need to be retrieved. *Bitmap-based data transfer* (BDT) can handle such non-fixed stride data efficiently. BDT retrieves data based on a start address and bitmap that specify which data are accessed in the same row of DRAM. Suppose there is a 32-bit processor and DRAM with 4 KB, i.e., 1024 words, of an I/O buffer. In this case, if 1024 bits of bitmap are given, any words in the I/O buffer can be specified for retrieval. The column addresses of DRAM can easily be calculated in the memory controller with the start address and bitmap. It continuously provides the generated column addresses to DRAM in page mode, and then, the requested non-fixed stride data are retrieved in a pipeline manner (see Fig. 4, center). BDT is very useful when two or more attributes are scanned and/or tuples are sparsely allocated in memory. Suppose that two attributes must be read by a selection. If SDT is used, it must be applied twice. In contrast, BDT can simultaneously retrieve two or more attributes, even irregularly allocated data.

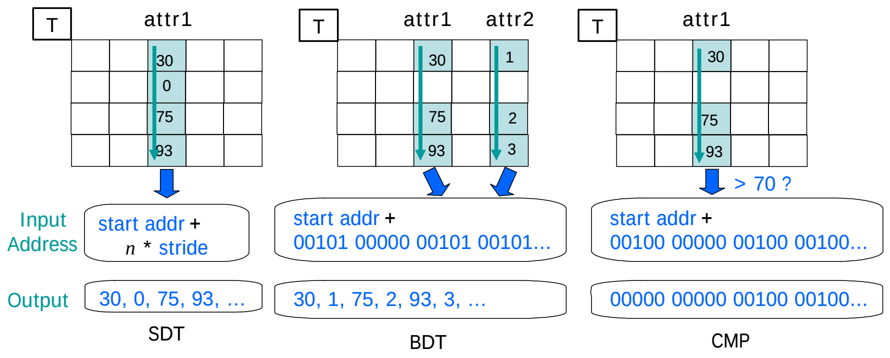

Figure 4. Concepts of SDT (left), BDT (center), and CMP (right)

Database operations, in general, consist of a large number of simple comparisons of two data. For example, selection compares data of an attribute with a constant value, and join compares data of two attributes. These operations do not require a datum itself but a Boolean value, i.e., either *true* or *false*. The comparison can be executed even outside of the CPU because comparing two primitive data is trivial with a hardware comparator. If hardware comparators are installed on a memory module and can process comparisons on it as shown in Fig. 5, the bit vector of the resulting maximum 32 comparisons can be transferred to the CPU at one system bus cycle with a 32-bit machine, though the conventional cache-based memory access transfers only a single 32-bit of primitive data at the same cycle. With this additional hardware on a memory module, not only parallelism in comparisons is obtained, but also the number of data transfers from main memory to the CPU decreases.

This scheme can be combined with BDT. For example, suppose that we need to check whether or not the third attribute values are more than 70 (see Fig. 4, right). If we set the bitmap as "00100 00000 00100 00100…," an operator as ">" and a constant value as 70, as well as the start address, the resulting bit vector "00000 00000 00100 00100…" returns. Bits of "1" in the resulting vector mean that the corresponding attributes are satisfied with the given condition. We call this access method *comparator-based data transfer* (CMP). CMP is regarded as an extension of BDT. The difference is the output form from main memory. CMP returns a compact bit vector data, while BDT does real values.

The size of the bit vector depends on that of the row of DRAM. Suppose that the row size is 4 KB, i.e., 1024 words. In this case, the size of the bit vector is 1024 bits, i.e., 32 words long. Even though only one bit is required, 32 words are mandatory for masking the bitmap for input and 32 words of a resulting bit vector are also mandatory for output. If the number of data compared is small, transferring 64 compulsory words between the CPU and main memory becomes overhead. However, even though the number of data compared is large, the overhead is fixed.

Access models of SDT, BDT, and CMP are different from the existing cache-based one. The data fetched by these access methods cannot be stored directly in the cache, because the data in a cache line are assumed to be located in contiguous addresses. In order to receive such data on a CPU, we assume an on-chip FIFO that behaves as a stream buffer to hold the data fetched by SDT, BDT, and CMP (see Fig. 6). Stream buffers are generally used and are often implemented in embedded processors and graphic accelerators.

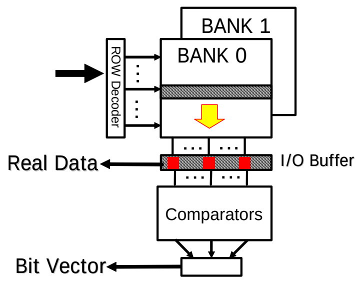

Figure 5. Configuration of memory module for CMP model

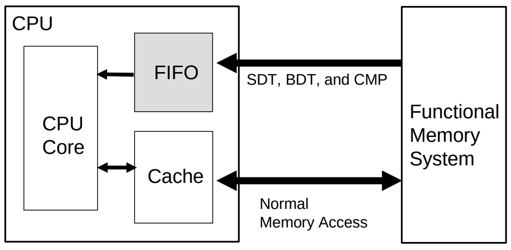

Figure 6. System model

4. ACCESS PATH SELECTION FOR MAIN MEMORY DATABASES

4.1 Access Paths

Query optimization is one of the most important tasks for MMDBs as well as conventional disk resident databases to process a given query with a shorter execution time. Query optimization is generally divided into two parts. One is building an optimal query tree, so that the total cost of the query can be minimized. The other is to choose appropriate access paths to efficiently retrieve data from physical storage. In both MMDBs and conventional databases, the former optimization process has no major difference. However, the latter one is completely different. Therefore, we focus on access path selection in the following discussion, as we introduced three new memory access schemes to reduce memory access latency in section 3.

The proposed SDT, BDT, and CMP are regarded as optimizations of scan-based data access, while conventional cache-based memory access allows both an index-based one (hereafter called Index, for short) as well as a scan-based one (hereafter called Normal, for short) as shown in Fig. 7. Therefore, we have to deal with a total of five data access methods for access path selection. Though only the Index and Normal access methods have to be considered in conventional MMDBs, we need to take into account SDT, BDT, and CMP as well. These additional memory access methods cause combinatorial complexity in access path selection for a given query.

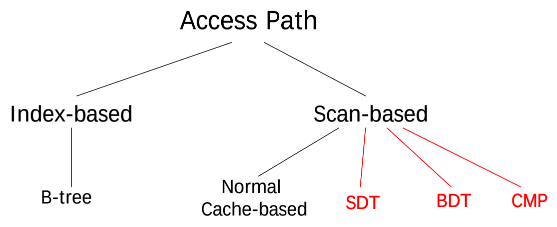

Figure 7. Access paths.

The three proposed memory access schemes are referred to as a stream data access model. As we assume simple computer systems such as PCs and embedded systems, the number of memory channels is limited to one or two. In this case, when main memory access occurs while stream data access is being processed, the stream access needs to stop temporarily because the main memory access must be caused by an occurrence of either an instruction cache miss or a data cache miss including data cache replacement. Neither operation can be delayed because they are the most important for avoiding CPU stalls. Resumption of the stream access requires set-up of the memory controller again. It takes a long time because the set-up process is done at system bus speed. However, an interruption of stream data access never happens while other instructions access only the cache.

It is difficult to inhibit such main memory access during stream data access for query processing. The major reason is that the size of intermediate query results exceeds the capacity of the cache, which causes cache replacement. However, the possibility of cache replacement can be reduced if CMP is used. For example, in the case of selection, CMP generates only a fixed size of bit vector data as a query result whether the selectivity is high or not, while other methods generate intermediate results whose size is proportional to the selectivity. Therefore, CMP is useful when the size of intermediate results is large.

As for index structures for the main memory database, cache-conscious indices have been proposed (Rao et al., 2000; Takamizawa et al., 2006). These indices are basically based on B+-tree, and are suitable for exact match queries. However, they do not seem to be adequate for range queries, because range queries need to access leaf nodes by following side links. When the number of side links followed increases, range queries tend to cause many cache misses, which leads to considerable degradation in performance. In such a case, use of efficient scan-based data access methods, e.g., SDT, BDT, and CMP, might be more reasonable than an index-based one. In particular, CMP is quite suitable for large range queries because comparisons for range conditions are calculated in the functional memory system. In addition, CMP reduces the possibility of cache misses because the size of the resulting bitmap is fixed and compact. Other access paths do not reduce the cache misses because the size of the results is larger and proportional to the selectivity of queries.

4.2 Effect of CMP for Queries

Suppose we are dealing with more general queries including join and selection. Join is usually processed after selection is done, as a result of query optimization. It is difficult for CMP to process join efficiently, while it can do so for selection. This is because join requires two materialized tables so that join conditions can be checked for each tuple. If CMP is still used to read a table for join, many translations from the resulting bitmap of CMP into materialized values are required because the bitmap only indicates which tuples or attributes are satisfied with a given query condition. When we assume a nested-loop as the join algorithm, CMP should not be used for the inner loop to avoid costly data translations. However, for the outer loop of join, CMP can be applied because each bitmap needs to be translated into materialized values only once.

Note that translation from the bitmap into materialized values is identical to what BDT processes. Therefore, we can derive an interesting heuristic rule for join after selection: selection is processed by CMP, and then, join reads its resulting bitmap with BDT for the outer loop. We call it the *CMP/BDT heuristic rule*. The CMP/BDT heuristic rule can reduce the size of the intermediate result of selection, and thus, the possibility of a cache miss decreases. Moreover, join can retrieve the selection result efficiently with BDT (see Fig. 8). Hence, the CMP/BDT heuristic rule is very effective when the selectivity of selection is high because the size of the resulting bitmap becomes larger than that of real data under low selectivity conditions.

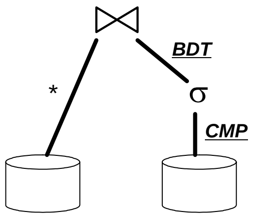

Figure 8. CMP/BDT heuristic rule

The cost of each access path can easily be calculated by making use of existing cost functions used in disk resident databases. However, I/O cost has to be modified to memory access cost, which can be calculated by using the cache miss rate, cache size, latency for accessing DRAM, the ratio of system bus cycle to CPU cycle, etc. For example, the cost of scanning a table by CMP is represented as

$$C_{CMP} = p\left((2 + \frac{L_{mem}}{W})T_{MC} + T_{latency} + \frac{L_{mem}}{W} r \right),$$

where $L_{mem}, W, T_{MC}, T_{latency}, r,$ and p are the line size of DRAM, the size of a word, latency to set word data to memory controller, latency for accessing DRAM, the ratio of system bus cycle to CPU cycle, and the number of pages to be read, respectively. Other cost functions can be derived in a similar way.

We summarize the above discussion and give heuristic rules for access path selection as follows:

1. Index is suitable for exact match queries and low selectivity range queries.
2. CMP is suitable for high selectivity range queries.
3. CMP/BDT heuristics is appropriate for queries of combinations of selection and join when the selectivity of selection is high (CMP/BDT heuristic rule).
4. Index is adequate for selection for a query containing selection and join when the selectivity of selection is low. This is identical to heuristic rule 1.

5. EXPERIMENTS

We evaluate the heuristic rules discussed in the previous section by a simulation using a subset of the Wisconsin benchmark database (Gray, 1993). All tuples are supposed to be densely stored in main memory, and no vacant tuple space exists, i.e., SDT can be used. Original queries were prepared to verify the effect of the heuristic rules. We employed a 32-bit SPARC processor simulator to investigate the total number of CPU cycles spent for benchmark queries.

5.1 Parameter Setting for Simulation

We assume that the speed of DRAM equals an SDRAM in which one datum can be read in a system bus cycle. However, the DRAM used for cache-based (Normal) access is assumed to be DDR DRAM, in which two data can be read in a system bus cycle, considering current trends. DRAM's row size is assumed to be 4 KB. Since we assume small-scale computers, we also assume the number of memory channels is one.

A CPU has two sets of 8 KB 2-way set associative cache memories: one for instructions and the other for data. Both cache memories adopt the writeback model. The size of a cache line is 32 bytes, e.g., eight words. In addition, 4 KB of FIFO is prepared for SDT, BDT, and CMP. Regarding the CPU and system bus cycles, we assume that the execution of an instruction requires one CPU cycle if no cache miss happens. Using the system bus cycle ratio to the CPU cycle *r*, the required clock cycles for operations are set as follows:

- read and write one word under a cache hit: 1,
- read latency under a cache miss: 10*r*,
- writeback a cache line to DRAM: 14*r*,
- set one word data to a register in the memory controller: 4*r*,
- latency to set up SDT and BDT: 12*r*, and
- read one word from FIFO: 1 (under a FIFO hit), and 2~*r* (under a FIFO wait).

Note that the maximum latency to read data from FIFO under FIFO wait is determined by a ratio *r* of system bus cycles to CPU cycles. The following experiments use *r*=10 as a ratio value.

As for an index, a cache conscious ABC-tree was employed, because it outperforms other existing cache-conscious index structures (Takamizawa et al., 2006).

Data structures to store intermediate and final results that we used are (i) an array of tuples satisfying query conditions, and (ii) a bitmap that indicates result tuples. The latter representation is not suitable for join results. It is difficult to represent a join result as a bitmap, because the join result is a set of tuple pairs, each of which comes from a table. Therefore, we utilized the bitmap representation only for selection results in the later experiments.

5.2 Evaluation of Range Queries

First, we examine a case for range queries. The query that we used is as follows:

```
Q1. SELECT *
      FROM  twok
      WHERE unique2 < CONSTANT ,
```

where the table “twok” contains 2,000 tuples. We changed the contestant value “*CONSTANT*” from 0 to 2,000, so that we can show how the selectivity affects the total execution time.

Figure 9 shows the results of query Q1. Horizontal and vertical axes mean selectivity and total spent CPU cycles, i.e., execution time, respectively. When the selectivity is very low, Index shows the best results, though we find no great difference between Index and SDT. When the selectivity is over 25%, CMP turns out to be the best because it generates a fixed size of bitmap as an intermediate result. The experimental results reveal that the heuristic rules for range queries are correct. Note that Index is inferior to scan-based CMP where the selectivity is more than 25%, and shows almost the same performance as scan-based SDT.

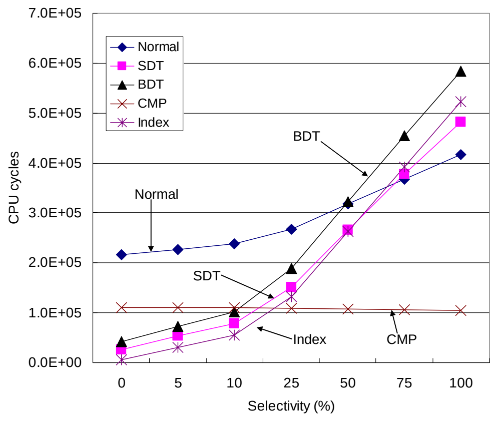

Fig. 9: Comparison of query Q1

5.3 Evaluation of Queries of Combination of Selection and Join

Next, we investigate two queries of a combination of selection and join. Regarding the join, we tuned up a nested loop join algorithm and made it cache conscious. More specifically, the table to be joined is divided into blocks, so that each block accounts for less than a half size of the cache. Then, block-wise comparisons are applied to the join. With this technique, cache misses decrease during the join.

The queries used for the experiments are as follows.

```
Q2. SELECT *
    FROM  ten t1, tenk t2
    WHERE t1.unique1 = t2.unique1
    AND  t2.unique2 < CONSTANT
```

```
Q3. SELECT *
    FROM  onek t1, tenk t2
    WHERE t1.unique1 = t2.unique1
    AND  t2.unique2 < CONSTANT
```

The difference of queries Q2 and Q3 is the size of the tables. Tables “ten”, “onek”, and “tenk” contain 10, 1,000, and 10,000 tuples, respectively.

In Figs. 10 and 11, the horizontal and vertical axes mean selectivity of selection and the total CPU cycles spent, respectively. The legend in graphs “X-Y” indicates that selection is processed by memory access method Y followed by being cascaded to the input of the inner loop of join, and that of the outer loop is by method X. The legends “*-CMP” mean that the CMP/BDT heuristic rule is applied. The selection result is stored as a bitmap representation when the CMP/BDT heuristic is used. The legend “Normal-Index” is a special case in which no selection result is produced. Selection is concurrently processed during join by using the index. Except for “*-CMP” and “Normal-Index”, the selection result is written as an array of real data. In these cases, SDT is used to read the selection result into the join operator.

As Fig. 10 shows and as we described in section 4, when the selectivity is low, Normal-Index is the best. This is quite natural because the index works fine under such a condition as described in section 5.2. Otherwise, when the selectivity is high, Normal-CMP is the best.

Specifically, when selectivity is less than 25%, all graphs are almost overlapped. Therefore, Normal-CMP is regarded as a better choice even in this range. When selectivity is high, BDT-CMP and SDT-CMP are inferior to Normal-CMP. They use two stream-based data transfers concurrently during join. Concurrent use of stream-based data transfer usurps each other's system bus. This causes frequent interruption of data transfers, and then reduces performance.

Note that use of an index should, in general, improve the performance of queries. However, as shown in Fig. 10, we found that the index is effective only under a quite limited condition. In most cases, scan-based memory access methods outperform the index. Since data retrievals by the index involve random access, frequent cache misses occur and cause performance degradation.

As shown in Fig. 11, in the case of larger tables, Normal-CMP is certainly the best when the selectivity is more than 10%, though the advantage of Normal-CMP is not prominent. The total CPU cycles spent by Normal-CMP, SDT-Normal, and Normal-Normal cannot be differentiated. BDT-CMP and SDT-CMP are worse than others due to the same reason described above.

The total CPU cycles of Q3 are of a two-fold magnitude greater than that of Q2. In other words, the execution time of join is dominant. In this case, the advantage of CMP is completely concealed by the join cost. Therefore, Normal-CMP, Normal-SDT, Normal-Normal, and Normal-Index show almost the same performance. If we use a sophisticated join algorithm in main memory like a radix-based one (Boncz et al., 1999), the effect of CMP would emerge even under heavy join cost.

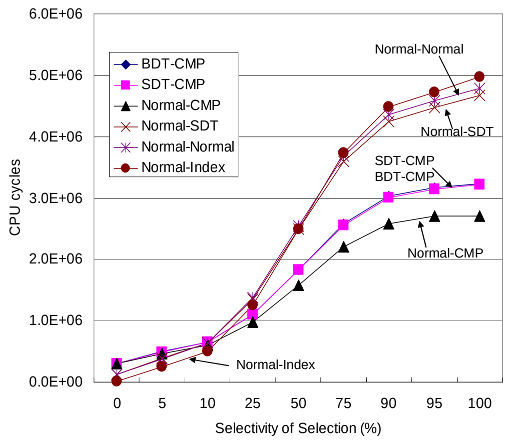

Figure 10. Comparison of query Q2

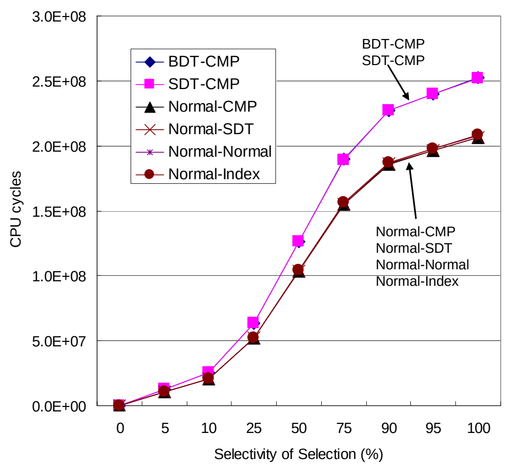

Figure 11. Comparison of query Q3

6. RELATED WORKS

Several main memory database systems were developed in the 1980s (Garcia-Molina et al., 1992). Their development mainly focused on concurrency controls and logging for transaction processing because CPUs were as slow as memory access speed.

To cope with the memory wall problem in the context of database systems, cache-conscious index structures have been proposed (Rao et al., 2000; Takamizawa et al., 2006). Boncz et al. (1999) investigated join algorithms in main memory, and Ailamaki et al. (2001) suggested that modification of page layout improves cache performance. All these studies aimed to overcome the memory wall with software.

Some research that deals with high performance memory access has been conducted. Impulse (Carter et al., 1999; Zhang et al., 2001) allows discontinuous data to be continuously transferred to the memory controller, where contiguous addresses in vacant virtual memory space are attached to the data. These data are treated in the same way as ordinary ones by the cache. The essential idea, which reduces unnecessary traffic in a system bus, is similar to ours. However, their approach did not refer to how physical memory is accessed. In contrast, our approach tries to make use of further intrinsic main memory performance using the page mode of DRAM, even with one memory module. In this sense, our approach can be widely applied even to PCs and embedded systems that have only one memory module.

There were ideas that some computation is executed in a storage device. Slotonik (1970) proposed logic per track, which checks partial query conditions to reduce the overhead of I/O channels. Though the essential idea of logic per track is quite similar to that of CMP, we assert that CMP can also be used for optimization in access path selection. Patterson et al. (1997) proposed a memory device with computation logic, called IRAM. A CPU often uses multiple IRAMs, which work as a bunch of co-processors. However, IRAMs cannot be differentiated from a heterogeneous multiprocessor system.

Proposals of database processing that uses special hardware include use of graphic processors (GPUs) (Govindaraju et al., 2004), and content-addressable memories (CAM)

(Bandi et al., 2005). The CAM approach also focuses on memory access but is more expensive than DRAM as well as being incompatible with DRAM. Therefore, the usage of CAM is rather restricted.

As for access path selection, most studies concern conventional disk resident databases (Graefe, 1993). Though Lehman et al. (1986) discussed query processing for main memory databases, the effect of access gaps between the CPU and main memory has not been considered.

7. CONCLUSION

In this paper, we discussed the heuristic rules for selecting an appropriate access path from SDT, BDT, CMP, Normal, and Index for main memory databases. In particular, an original CMP/BDT heuristic rule was introduced for queries including selection and join.

The experimental results revealed that the rules are reasonable. The derived access paths obtained up to a 4.1-fold faster speed for a range query and up to a 1.7-fold faster speed for a query containing selection and join, compared to only using the conventional cache-based memory access.

We did not mention how we determine the threshold of selectivity that is needed to change memory access methods in detail. We are investigating the formula that derives the threshold. Further studies remaining include developing high performance indices supported by SDT, BDT, and CMP, so that efficient exact matches and range queries can be implemented.

ACKNOWLEDGMENT

This work was partly supported by the PRESTO Program “Information Infrastructure and Applications” of the Japan Science and Technology Agency.

REFERENCES

Ailamaki, A. et al. 2001. Weaving Relations for Cache Performance. *Proceedings of VLDB 2001*, pp. 169-180.

Ailamaki, A. et al. 1999. DBMSs on a Modern Processor: Where Does Time Go? *Proceedings of VLDB '99*, pp. 266-277.

Bandi, N. et al. 2005. Hardware Acceleration of Database Operations Using Content-Addressable Memories. *Proceedings of ACM DaMoN 2005*, pp. 35-40.

Boncz, P. et al. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. *Proceedings of VLDB '99*, pp. 54-65.

Carter, J. et al. 1999. Impulse: Building a Smarter Memory Controller. *Proceedings of HPCA 1999*, pp.70-79.

Chen, S. et al. 2001. Improving Index Performance through Prefetching. *Proceedings of ACM SIGMOD 2001*, pp. 235-246.

Garcia-Molina, H. and Salem, K. 1992. Main Memory Database Systems: An Overview. *In IEEE Transaction on Knowledge and Data Engineering.* Vol. 4, No. 6, pp. 509-516.

Govindaraju, N. K. et al. 2004. Fast Computation of Database Operations using Graphics Processors. *Proceedings of ACM SIGMOD 2004*, pp. 215-226.

Graefe, G. 1993. Query Evaluation Techniques for Large Databases. *In ACM Computing Surveys*, Vol. 25, No. 2, pp. 73-170.

Gray, J. 1993. *The Benchmark Handbook*. Morgan Kaufmann Publishers.

Gran, J. and Reuter, A. 1993. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers.

Jagadish, H. V. et al. 1994. Dali: A High Performance Main Memory Storage Manager. *Proceedings of VLDB '94*, pp. 48-59.

Kim, K. et al. 2001. Optimizing Multidimensional Index Trees for Main Memory Access. *Proceedings of ACM SIGMOD 2001*, pp. 139-150.

Lehman, T. J. and Carey, M. J. 1986. Query Processing in Main Memory Database Management Systems. *Proceedings of SIGMOD '86*, pp. 239-250.

Lehman, T. J. et al. 1992. An Evaluation of Starburst's Memory Resident Storage Component. *In IEEE Transaction on Knowledge and Data Engineering*, Vol. 4, No. 6, pp. 555-566.

Miyazaki, J. 2005. Hardware Supported Memory Access for High Performance Main Memory Databases. *Proceedings of ACM DaMoN 2005*, pp. 41-46.

Miyazaki, J. 2006. A Memory Subsystem with Comparator Arrays for Main Memory Database Operations. *Proceedings of ACM SAC 2006*, pp. 511-512.

Patterson, D. et al. 1997. A Case for Intelligent RAM, *In IEEE Micro*, Vol. 17, No. 2, pp. 34-44.

Rao, J. and Ross, K. A. 2000. Making B+-tree Cache Conscious in Main Memory. *Proceedings of ACM SIGMOD 2000*, pp. 475-486.

Slotonik, D. L. 1970. Logic per Track Devices. *In Advances in Computers*, Vol. 10, pp. 291-296.

Takamizawa, H. et al. 2006. Array-based Cache Conscious Trees. *In IPSJ Digital Courier*, Vol. 2, pp. 25-38.

Zhang, L. et al. 2001. The Impulse Memory Controller. *In IEEE Transaction on Computers*, Vol. 50, No. 11, pp. 1117-1132.