Relational Algebra Machine

GRACE

Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-oka

University of Tokyo
Faculty of Engineering, Department of Information Engineering
Bunkyo-ku, Tokyo, 113 Japan

Abstract

Most of the data base machines proposed so far which adopts a filter processor as their basic unit show poor performance for the heavy load operation such as join and projection etc., while they can process the light load operations such as selection and update for which a full scan of a file suffices. These machines which incorporates n processors takes it $O(N*M/n)$ time to execute join of two relations whose cardinalities are N and M respectively.

GRACE adopts a novel relational algebra processing algorithm based on hash and sort, and can join in $O((N+M)/n)$ time. GRACE exhibits high performance even in join dominant environment. In this paper, hash based relational algebra processing technique, its implementation on parallel machine, and architecture of GRACE are presented.

1. Introduction

1.1 History of Data Base Machine

The relational model proposed by Codd has saddled a software implemented data base management system with a big performance burden in return for its high data independence, simplicity, uniformity in the operator set and its firm logical foundation. This is one of the largest motivation that stimulates the research on the data base machine.

The origin of Data Base Machine(DBM) is considered to be 'Logic per Track' concept by Slotnick[1]. Many of the machines proposed so far adopted this idea as basis with some enhancement. As is shown in RAP.1[2] or CASSM[5], each head of a disk is equipped with some simple logic and it can efficiently perform selection of records which satisfy a certain condition. Since this logic can filter out the unnecessary records, it is called a filter processor. The filter processing can reduce the amount of data that must be transferred between the secondary storage device and the main memory of a host computer. Later the storage media are changed from disks to electronic devices such as CCD and magnetic bubble memories. And the machine of this type, namely the one which is constructed by many identical cells, where a cell is composed of a pair of a processor and a memory bank, comes to be known as 'the cellular logic type data base machine'[7]. In the support of a relational data base, this outperforms the conventional one by orders of magnitude in the execution of relatively light load operations such as selection and update. Because there is no auxiliary data structure such as indexes in many of the machines, the heavy overhead caused by the consistent update of the indexes disappears. Elimination of the large software for the complex access method is supposed to be another merit of these machines. In heavy load operations such as join and projection including duplicate elimination, however we can not expect large scale of performance improvement but only slight one[3]. In RAP.1 many revolutions of the disk are required to implement an implicit join. The brute force application to join of the filter processing approach pioneered by Slotnick, which is very powerful to the operation for which one scan of the file is sufficient, has gradually revealed its limitation. That is, most of the machines are regarded as basically filter processors and it is difficult to judge that they hold a full efficiency in join and duplicate elimination which are essential operations for a relational data base management. There seems to be no machine which provides sufficient performance for join.

1.2 Join Processing Techniques

Here we examine the join processing method on several machines. There might be lots of another interesting features peculiar to the individual machine but we ignore them and concentrate on join.

In the cellular logic type DBM, the processing load of join is proportional to the product of each relation's cardinality, and it is processed in parallel on each cell. That is, tuples of the source relation is broadcast to the cells comprising the target relation, and then all target cells simultaneously compare the obtained tuples with its own tuples. Therefore the processing time is proportional to N^*M/n^*k *where M and N are the cardinalities of both relations, n is the number of cells and k is the number of comparators per cell. RAP[2,3,4] and EDC[8] etc. belong to this category. As was shown in performance evaluation of RAP[3], DBM of this type is not always suited for join operation and the performance gain against a conventional machine is not so definite.*

In RELACS[9,10] which is characterized by its extensive use of associative processors, the processing load itself is same but the parameter k is relatively large, while in cellular approach the processing power of one cell is small and simplified because its cost increases linerly as the volume of the data base becomes enlarged. Anyway it also employs exhaustive matching algorithm, so it is difficult to attain high performance. The separation of the high performance processing unit and the memory banks necessitates the data transfer path having high bandwidth between them.

In DIRECT[11,12] which actualizes the page level control, the load of join is $O(n^*m)$ *where n and m are the number of pages occupied by two relations respectively, and it is processed by using* $max(m,n)$ *processors in* $O(min(m,n))$ *time. Join is implemented in the manner that many processors activated by the controller are assigned one page of the source relation and scan all the pages of the target relation. This machine does not adopt a special processing unit such as a filter processor but employs a general purpose* μ *processor and the page itself is processed in the conventional manner; the page processing consists of 3 phases, page loading, page processing based on sort, and page storing.*

Systolic array based DBM[13] relies on technological advances in VLSI circuitry. The special purpose VLSI chip activates many comparators in pipeline fashion and can join two relations by only feeding them in counter direction. But this dose not generate a joined relation but only a joinability matrix. Highly Concurrent Tree Machine[14] also assumes the VLSI implementation and can join in about 2(N+ no. of result tuples) where N is the cardinality of both relations. While these two machines are very fast when the related data can be accommodated in the machine, there remains a partitioning problem for cases where the problem size exceeds its capacity. The fact that both require at least O(N) comparators should be noticed.*

Join processor in DBC[15,16] also relies on the development of VLSI

technology and takes $O(N/n+M)$ time in join. The join algorithm is basically the same as the cellular logic type DBM, namely exhaustive matching of both relations.

Data stream data flow data base computer[17] is composed of two main functional modules, sort engine and search engine, which realize $O(N)$ sort and $O(\log N)$ search respectively. On join processing, the source relation is fed into a sort engine and its sorted data is then led to a search engine. When loading of a search engine completes, the target relation is put into it, and a join operation is performed in pipeline fashion. In this manner, two pages of both relations are joined in $O(N)$ time where N is the page size. The control scheme at page level is not clarified.

DPNET based DBM[18] which takes similar approach to ours, completes join in $O(N*M/n^2*k)$ time. The data streams from each head of a disk are partitioned dynamically and sent to the same number of processors. It assumes to use an associative memory in the processing unit in order to follow the data transfer rate of the disk. Therefore its capacity is limited and it needs several revolutions like RAP when the source relation cannot be fitted into.

CAFS[19] employs an unique algorithm based on the hashed bit array for join operation which is examined in detail in the next section. It can be regarded as joinability filter; it can filter out many of tuples which cannot be joined. This leads to a large load reduction but actual join must be done at a host machine. The same approach can be found in LEECH[20] and CASSM[6].

It is obvious from the above survey that there is no machine that can perform efficient execution of the relational algebra operation, especially join. In this paper, we propose a novel relational algebra machine based on hash and sort algorithm[21,22], which can join in $O((N+M)/n)$ time where N and M are the cardinalities of two relations and n is the number of memory banks.

2. Hash Based Relational Algebra Processing

2.1 Utilization of Hash

Hash is one of the efficient search strategies, where necessary records can be obtained with almost one access provided that load factor remains low, and this is one of the fastest access method. It is not such a conventional static usage as an access method but a dynamic usage of hash technique that we utilize in our machine. The clustering feature of hash operation can reduce the load of join.

The simple join algorithm takes time proportional to the product of two relations' cardinalities. However, if two relations are clustered on the join attribute, that is, the tuples are grouped into disjoint buckets based on the hashed value of the join attribute, there is no joining between tuples from buckets of different id(hashed value of the join attribute). Tuples of i-th bucket in one relation cannot be joined with those of j-th bucket($j \neq i$) in the other relation but with only i-th bucket. So the total load of join operation is reduced into join between buckets of the same id. Let

$$N = \sum_{i=1}^{s} n_i \ , \quad M = \sum_{i=1}^{s} m_i$$

where N and M are the cardinalities of two relations, n_i and m_i are the sizes of the i-th bucket in each clustered relation , and s is a number of buckets. The total processing time T can be expressed as follows

$$T \propto \sum_{i=1}^{s} n_i * m_i$$

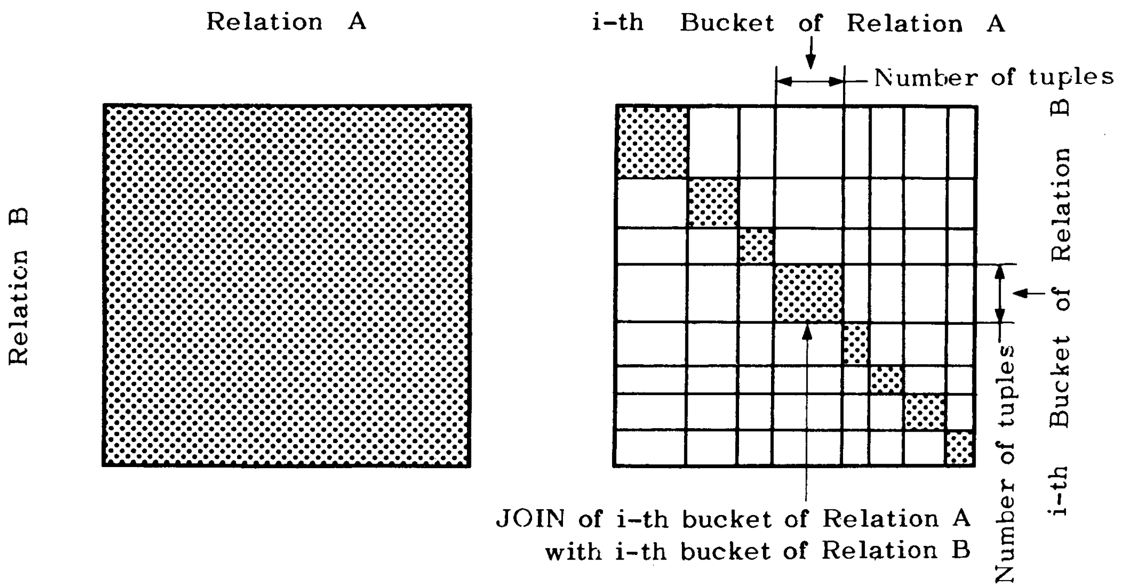
This load reduction effect is depicted in fig.1, where two axes denote two relations which are divided into s intervals and the cross section reveals the join load of $n*m$. The shaded areas correspond to the processing load. Therefore, this clustered approach can dramatically diminish the load in comparison with nonclustered ordinary approach.

Duplicate elimination task in projection also used to be a big burden in relational data base systems. The above approach can be applied to projection quite as well as join. Through hashing extracted fields of the tuples, the given relation is broken up into many disjunctive buckets. All the same tuples fall into the same bucket. Therefore duplicate elimination can be done in each bucket

independently. No need for inter-bucket comparisons. A database machine utilizing this clustering approach would attain a very rapid relational algebra execution.

Another technique utilizing hash is 'Joinability Filter' which can be found in CAFS and LEECH. In this approach tuples from one relation are hashed on the join attribute and its corresponding bit store is set. Then the other relation is read and hashed also on the join attribute, and the proper bit in the hashed bit array is checked. If the bit is set, it is assumed that this tuple has possibility to be joined. This hashed bit array as a joinability filter makes many tuples that cannot be joined sieved out and thereafter the cardinalities of both relations fall into small sizes, which results in large load reduction. While this method is very powerful as preprocessing, remaining tasks such as elimination of spurious tuples and tuple concatenation in explicit join must be done on the host machine. As was shown clear, this method is quite different from the previous clustering approach.

Two approaches discussed above are mutually independent, so both joinability filter processing which decreases the candidate tuples and clustered processing can be integrated together



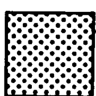


Fig. 1. Clustering Effects By Hashing In JOIN Operation

2.2 *Implementation on Parallel Processing Machine*

The great reduction of processing load of join and projection was shown to be actualized through the hash based relational algebra processing method. Now, we will consider how to materialize this method for a database machine.

The buckets generated by hash are independent each other. Therefore, rather than processing them serially using only one processor, the relational algebra operation can be executed much faster by processing each bucket in parallel using many processors. Note that there is no inter-processor communication during bucket processing. Some problems can be identified in designing a relational algebra machine which realizes this bucket parallel processing.

First problem is how to exploit the bank parallelism. All the tuples of the relation could be processed serially. But the stream may become very long for large data bases, and it is desirable to divide a long stream by distributing the relation over several memory banks and to process segmented streams in parallel. By exploiting bank parallelism, if the relation could be staged in the working page space which consists of multiple memory banks, processing time can be independent of the cardinality of the relation and is determined by the memory bank capacity which is constant.

Bucket processing is second problem. By hash, processing load can be reduced from $O(s^2)$ *to* $O(s)$ *at bucket level, where s is the number of buckets. Processing of a bucket itself should be fast in order not to disturb the data stream. Namely* $O(n)$ *processing of a bucket rather than* $O(n^2)$ *is preferable where n is a size of a bucket. Integration of both* $O(s)$ *processing at bucket level and* $O(n)$ *processing within bucket makes it possible to organize a relational algebra machine of high performance.*

In order to attain high degree of parallelism, efficient bucket allocation to processors must be achieved, which is the third problem. Buckets can be processed in parallel by allocating them to many processors. In this bucket allocation, it is necessary that each processor can gather the data of the allocated bucket efficiently. Generally relations might be very big and the number of buckets generated by hash is larger than that of available processing units. So buckets beyond the number of available processors must be processed serially, where bucket serial data stream needs to be generated efficiently by memory banks.

The size of buckets might vary so much. The mechanism is required to handle the fluctuation of processed objects. Handling of nonuniformity in bucket size is the forth problem. The capacities of buckets are not necessarily uniform but may differ each other so much. There might be a bucket overflow, which here stands for the case that the bucket size is larger than the capacity of a processor. This phenomenon of nonuniformity caused by hash function is inevitable. On the other hand, for the machine architecture the fluctuation of the amount of each object to be processed generally leads to degradation of

resource utilization and system performance. We have to manage these problems inherent to hash and to provide means to handle such an exceptional phenomena as bucket overflow.

To sum up, we can enumerate four major problems to be considered in the design of a data base machine which realizes parallel execution of our hash based processing method. These are utilization of bank parallelism, processing of a bucket, bucket allocation to processors, and nonuniformity of buckets. We will discuss these problems in more detail at next section.

3. Design Consideration

We will explain our treatment of problems described at the previous section.

1) Bank Parallelism

As is known from the brief survey of join processing in section 1, Systolic Array, Tree Machine and SOE-SEE method can execute join in $O(N)$ time, but it is difficult to do in $O(N/n)$ time with n modules. We say that 'Bank Parallelism' is fully exploited in case $O(N/n)$ time is achieved with n banks. For example, DBC Join processor cannot execute join in $O((M+N)/n)$ time but in $O(M/n+N)$ time. In DIRECT, if as much processors as the product of the number of pages occupying the operand relations were to be activated and each processor could process a page in liner time, $O((M+N)/n)$ join would be possible. However it is too expensive. Our aim is to seek a machine which can reflect bank parallelism at reasonable cost.

We proposed the hash based join method in section 2, where the relations are hashed into several disjunctive buckets and thereafter each of them is processed serially. Incorporating bank parallelism in this method, we can identify two approaches.

a) Bucket converging method

b) Bucket spreading method

Suppose the relation stored over multiple source memory banks are to be hashed and transferred to the same number of destination banks. If the correspondence between the source and the destination is fixed, the tuples composing a certain bucket would be spread over banks. On bucket processing, a processor has to gather the tuples of its allocated bucket from all the banks. In order to avoid this situation, the bucket converging method can be derived naturally, where tuples of a bucket over source banks are converged into a single destination bank. There are two major problems in this approach. One is a bank overflow problem which is a conventional one caused by nonuniformity of the hash function. Since the data distribution can not be uniform and a bank capacity is limited, a situation would occur where some buckets have to accept the tuples beyond its capacity. At least we have to prepare a larger space than the actual capacity of the relation. The determination of load factor is very hard, which is also related to the efficiency of the storage utilization. This difficulty in memory management is crucial in the bucket converging method. DPNET which adopts this method provides no solution to this problem, In DPNET, the distribution terminates when one of banks overflows in source loading. Another facet of this nonuniformity is discussed in the next paragraph. The other problem is data confliction during transfer. The conflict occurs when a number of tuples are sent to the same bank at the same time. We have to manage this problem by an

appropriate method such as introducing some buffer. This problem is due to the fact that multiple data streams are hashed simultaneously.

In the bucket spreading method, tuples of a bucket have to be gathered from banks since they spread over them. But the gathering process itself can be pipelined, that is, a processor visits memory banks serially and a bank outputs the tuples of the bucket allocated to that processor (bucket allocation mechanism is discussed at 2) of this section.), so a processor runs through the pipe composed of banks. Thus we can activate the same number of processors as the banks and can expect fair performance improvement exploiting bank parallelism. Moreover there is no memory management difficulty of the overflow as is found in the bucket converging method. This approach also accompanies some problems. Pipeline processing works well when each segment time of it is equal. If some segment time is very large, it'll become a bottle neck of the pipeline. In the case where the correspondence between the source and the destination is fixed as described before, the number of tuples which belong to a certain bucket differs much among banks. This means that the segment time of the pipeline varies dynamically. It is anticipated to cause performance degradation. Provided that the hash function is random, it'll not so large. In order to resolve this segment time fluctuation in pipeline, we have to make the tuples of a bucket almost equally spread over the banks and to make all the buckets themselves almost equal in size. We do 'bucket flat distribution' to satisfy the first condition where a tuple emitted by a source bank is controlled to fall into the destination bank which has the bucket of that tuple least in number. We do 'bucket size tuning' to guarantee the second condition, which is discussed in the later paragraph. With the adoption of this bucket spreading approach with some enhancement, we can attain high performance by activating banks in parallel.

2) O(n) processing of a bucket

The processor is required to process a bucket in O(n) time not to disturb the data stream. And it is evident that the relational algebra operation can be completed in O(n) time when the relations are sorted on the attribute participating in that operation. So we decided to attach the processor with the O(n) hardware sorter. (which is discussed in section 5.1) Of course an associative processor or some functional unit with the capability of parallel comparison also may be possible, which in fact many of the proposed machines such as RAP, RELACS, and DPNET, etc. employ. It can process the data stream very rapidly and keep up with the stream completely. Its capacity, however, is generally limited in current technology and this imposes the condition that at least the cardinality of one relation should be very small. On the other hand, in our approach of using a sorter there is not so severe capacity limitation and operand relations can be treated equivalently. So either of operand relation need not be so small. The sorter can not complete sorting until the last data item arrives, so it takes longer to process a bucket than the

associative processor. But this hardly affects the performance because buckets are processed in a pipelined fashion.

3) Bucket Allocation

Bucket serial processing is necessary under the environment of finite resources of processors. In order to complete an operation in O(n) time, efficient generation of bucket serial data stream must be realized. It is required to output tuples of a bucket continuously, which are not necessarily placed adjacently each other inside the memory devices. Of course it is possible to do some preparatory processing when a memory bank inputs the data, but it also needs to be performed not disturbing the input stream. It is almost impossible to do with magnetic disk. On the contrary, it is clearly possible to achieve it by RAM Recently the development of semiconductor technology is extraordinary and it is found in a disk cash even at present time, so it will be feasible to use RAM as the staging buffer. However the size of the relation is very large even after the filter processing by an associative disk, so it is meaningful to seek a memory device lying between RAM and disk on memory hierarchy. We can find a magnetic bubble memory satisfying our conditions. The chip capacity of bubble is at present four or more times greater than that of semiconductor RAM and much more development is expected by contiguous disk technology. Transfer rate is relatively low but it can be improved by activating multiple chips in parallel. And dual conductor technology also seems to make a big contribution on it. Recent rapid progress promises well for the future. Moreover bubble has another advantages such as nonvolatility and flexible start/stop mechanism. Considering above features, we think magnetic bubble memory is also a good candidate for a staging buffer medium. We put some modification to a conventional major/minor bubble chip and make it suitable for efficient bucket serial generation.

4) Nonuiformity of Buckets

In a Direct Access Method the bucket overflow often degrades its performance largely because the storage medium is disk. In our case we are going to use magnetic bubble memory or large capacity and low speed RAM for storage media of staging buffers where a bucket is constructed using linked list in the former and mark bits in the latter, so there is no conventional overflow problem such as degradation in access time and memory efficiency. But the bucket size fluctuation itself arises another type of problems discussed before. In pipeline processing of bucket gathering, it is desirable that the size of each bucket is uniform. And the size itself had better be close to the processor capacity from the point of the processor utilization efficiency. If it takes $O(n^2)$ time to process a bucket, the smaller the bucket is, the faster a bucket can be processed. On the other hand, in the environment of our case where a bucket can be processed in O(n) time, we do not have to have the size of a bucket so small. Excessive bucket generation with the purpose of bucket size reduction

incurs extra overhead rather than the load reduction because the bucket allocation cost must be paid. Therefore it is desirable that the size is close to the processor's capacity. However, it is difficult to find a hash function dynamically which generates buckets with the size of processor's capacity. So we do 'bucket size tuning'. Namely we at first partition the relation into more buckets and then integrate some of them into a larger bucket with the volume less than the capacity of a processor. Through this preprocessing, we can have buckets of near uniformity. This 'bucket size tuning' process is well known as Bin Packing Problem which is NP-complete, and the pseudo optimal solution is already obtained. The overhead brought about by bucket integration is not so large. This can be overlapped with the data stream generation of the bucket. Once the first integrated bucket is obtained, then the data stream generation of the bucket can be overlapped with the bucket integration. Even by this bucket size tuning, it is impossible to make the size of each bucket completely uniform. It might still occur bucket overflow, in which case a large bucket is processed by concatenating a number of processors dynamically through one channel of a ring bus with a few overhead time.

4. Query Execution on Abstract Architecture of GRACE

4.1 Abstract Architecture

As shown in fig.2-1, the abstract architecture of GRACE is composed of three major components; DSP(Data Stream Processor), DSG(Data Stream Generator), and SDM(Secondary Data Manager).

DSP processes an allocated bucket sent from DSGs and produces result tuples on another DSGs. DSP consists of hardware sorter, filter processor, tuple manipulation unit, and hash unit etc. Each bucket is at first sorted with O(N) hardware sorter and relational algebra processing is applied to the sorted data stream in the tuple manipulation unit in DSP.

DSG provides the working space for the temporary relation to the subsequent processing and for the result relation. The relation filtered and hashed on SDM, as well as the result relation generated in DSP, is transferred into DSGs. DSG generates the bucket serial data stream to DSPs. Semiconductor RAM or magnetic bubble memory seems to be the most appropriate for its medium at present.

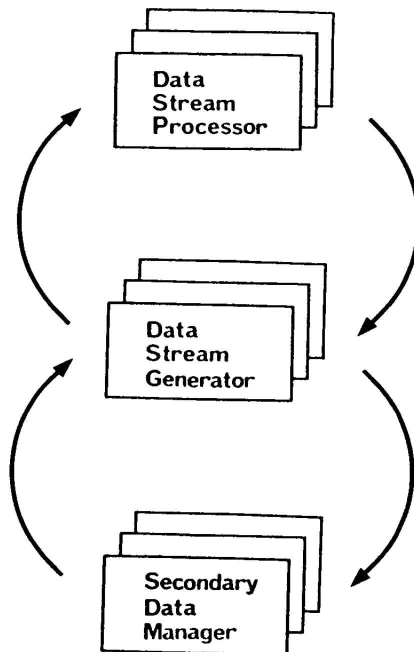


Fig.2.1 Abstract Architecture Of GRACE

SDM stores the relations of the data base on disk. The disk can transfer data from all heads of a cylinder in parallel, and it is equipped with filter processing function containing selection, restriction, bit map manipulation, and simple projection(attribute selection). On-the-fly processing limits the allowable complexity of the predicate in selection. The remainder of the predicate which cannot be evaluated in the filter processor of SDM is rendered to DSP. The hashing unit hashes the tuples on specified attribute and generates bucket ids.

4.2 Query Execution on GRACE

Here we consider how the query is executed on GRACE. The query is assumed to be complex and has many joins and projections, etc. The query processing consists of two major phases: staging phase and processing phase.

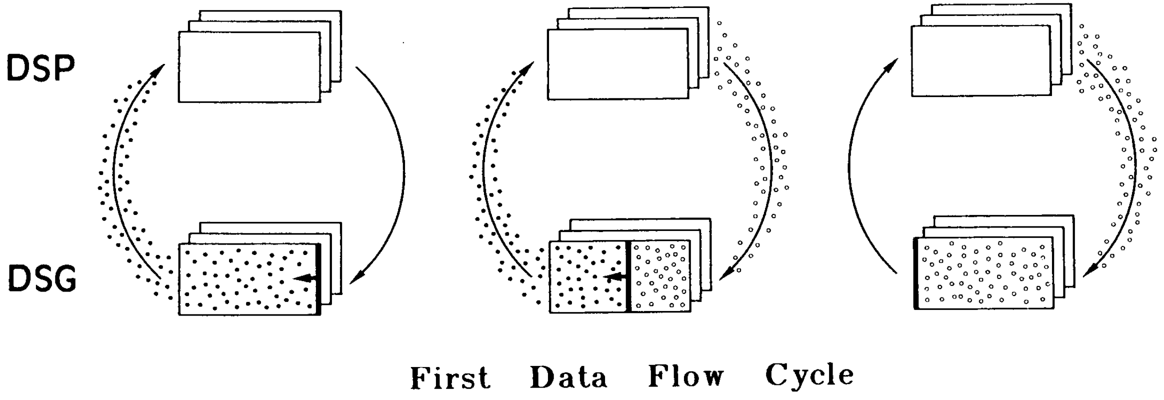
At the staging phase, relations necessary for the first join operation are staged from SDMs into DSGs. The data stream from disk is led to the filter processor in SDM, where selection and simple projection are performed on the fly. And then the filtered stream is hashed on the attribute which participates in that operation and hashed id is attached with each tuple. These hashed data streams are transferred from SDMs to DSGs over network between them. Once the SDMs begin to output their data streams, DSGs receive the tuples and maintain buckets corresponding to the hashed value. The relations are clustered overlapped with data transfer during the staging phase. When DSG completes data stream input, the clustered relation depicted in fig.1 is conceptually produced in DSG.

At the processing phase, actual processing is performed between DSGs and DSPs. After the staging phase DSGs literally generate data stream bucket-serially to DSPs. As a bucket is equally spread over DSGs, the DSP has to attach the appropriate data stream and gather the tuples which belong to that bucket. This proceeds in pipeline fashion and the data streams generated by DSG are not disturbed so much. The data gathering process itself is overlapped with the sorting process. A hardware sorter in DSP sorts the input tuples keeping up with the stream. When all the tuples of a bucket are taken in a DSP, it begins to operate on the sorted data stream from the sorter. Most of the operations necessary for relational data base support can be performed efficiently on the sorted stream. After DSP processes a bucket, it then proceeds to next bucket. Buckets are processed in parallel using multiple DSPs. One relational algebra operation terminates when all the buckets are consumed.

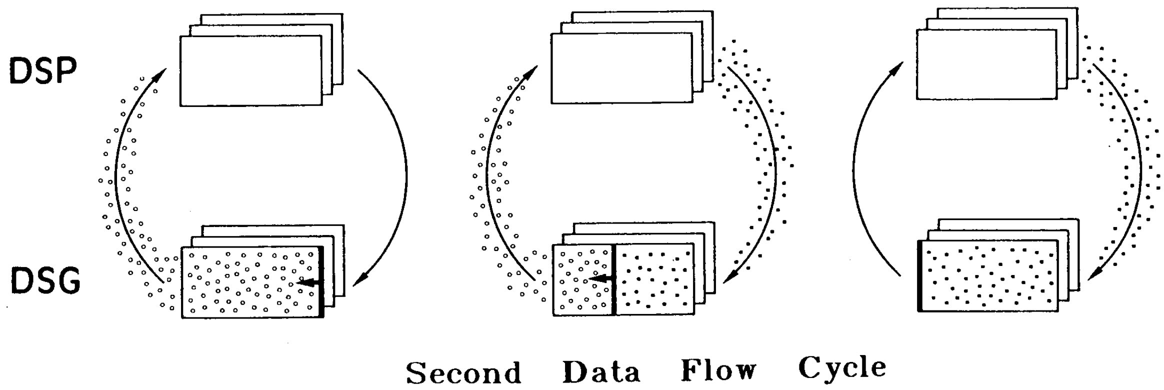
4.3 Operator Level Pipeline

Each operation in a query tree is executed as described above. As is shown in fig.2-2, one operation corresponds to one data flow cycle: a data stream is generated at first in DSG and then passes through DSP and at last is returned back to DSG. In a cycle, all the tuples in the source DSGs is transferred to the target DSGs. A complex query comprising multiple operations is implemented by repeating such cycles. As we can see in fig.2-2, once a data flow cycle terminates, new cycle of the next operation begins. Here we should notice that we don't have to interleave the hashing cycle of a result relation for the next operation. When a DSP processes a bucket, it hashes the result tuple on the subsequent operation and outputs a result data stream to DSG. Namely clustering operation of a result relation for the next operation is overlapped with the actual processing of the present operation. We named this 'Operator Level Pipeline'. By this processing schema, vanishes the overhead which we are afraid to be caused by clustering as preprocessing. The first clustering processing is overlapped with the staging phase. We don't have to execute operators one at a time for the cases where sufficiently large space in DSG is available. More than one operation could be performed simultaneously. In that case, as many cycles as the height of a relational algebra tree for the query would be required to get the result. For example, a relational algebra tree for the query in fig.2-3 is executed in a way shown in fig.2-4. The query includes join of four relations. For simplicity selections for the base relations and for the derived relations are not included explicitly. The operations which can be processed on the fly is executed implicitly overlapped with the join operation.

As mentioned above, our machine GRACE can execute a complex query very efficiently with repetitive data flow cycles. Accordingly, we can expect that GRACE can execute join sequence much faster than any DBM proposed so far.



First Data Flow Cycle



Second Data Flow Cycle

Fig. 2.2 Execution Overview

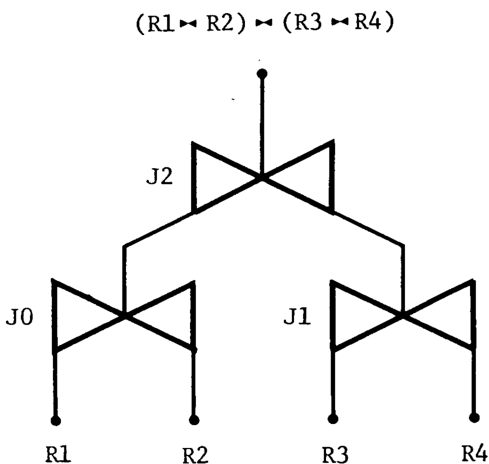


Fig.2-3. An Example of Relational Algebra Tree with 4 Joins.

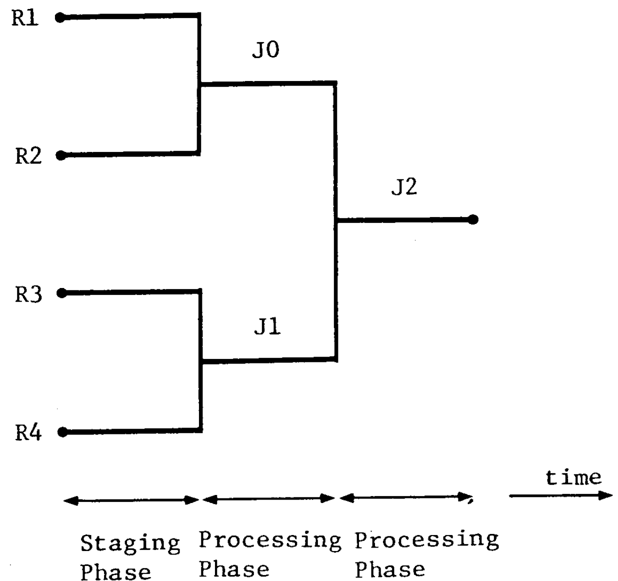


Fig.2-4. Execution Overview for The Query in Fig.2-3.

5. Hardware Architecture of GRACE

Here we proceed the hardware architecture of GRACE. As shown in fig.3, GRACE consists of four kinds of fundamental modules: processing module, memory module, disk module and control module. The first three correspond to DSP, DSG, and SDM in the abstract architecture of fig.2-1 respectively. The modules are connected with two ring buses. The processing modules and memory modules are connected by a processing ring and memory modules and disk modules are connected by a staging ring. The relations stored in disk modules are staged into memory modules with staging ring and then the data streams generated in memory modules are processed in processing modules through the processing ring. The time division multi-channel buses make many modules run simultaneously. Here we look at the organization of the individual modules.

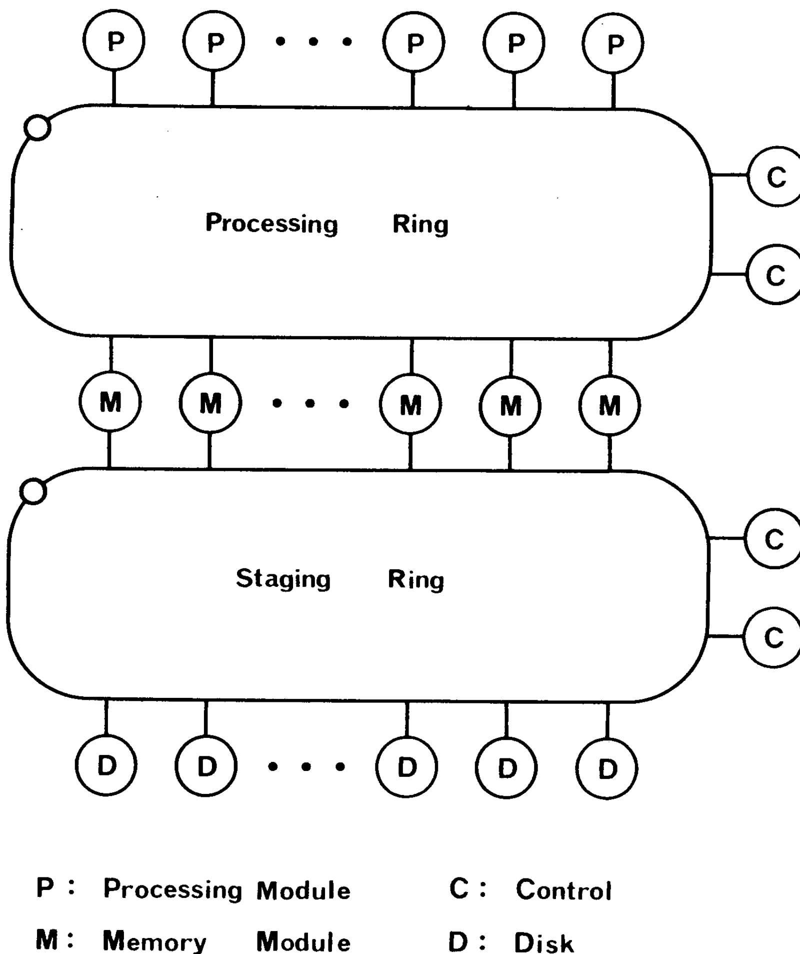


Fig.3 Global Architecture Of Data Manipulation Subsystem In GRACE

5.1 Processing Module

The role of processing module is to process a bucket efficiently. As was discussed in the previous section we determined to equip it with O(n) hardware sorter (it completes the sort in O(n) time) in order to realize data stream oriented processing. The tuples are generated serially by memory module, so it is required that the sorter can sort the stream of tuples efficiently. The sorters which assume that all the data to sort must be in its memory beforehand are not adequate. Among hardware algorithms, we choose the pipeline merge sort[24][25] which satisfies our conditions. This can sort the data keeping up with the input data stream and when it inputs all the data, then the sorted data stream is available after only a few delay.(see fig.4) The hardware structure of this sorter is depicted in fig.5. In a k-way merge sorter, the i-th processor has memory with the capacity of $k^{i-1}*(k-1)$ *items and the total capacity of memory also sums up to O(n). We constructed this sorter of microprogrammed control. The processing rate is mainly depends on the access time of the memory and our experimental one attained the rate of 3 MB/sec. The detail design considerations about it is to be discussed in the future paper. Anyway the bucket-serial data stream generated in memory module is led into this sorter of processing module. The key fields, that is, the join or projection attributes are arranged to be at the top of a record and after the key the relation id is added so that the join of the relations with the attribute of the same domain can be performed at once. Once the bucket input completes, its sorted stream enters a tuple manipulation unit where join or projection are performed. A qualification predicates over the joined tuple is evaluated and the result tuple is formed from only necessary attributes. Another functional unit in processing module is hashing unit. The tuple in an appropriate format from the tuple manipulation unit is sent to hashing unit and is hashed over the attributes for the next operation. After a hashed value is added to a tuple, it is transmitted into the processing ring through a ring bus interface unit.*

5.2 Memory Module

The role of a memory module is to generate an efficient bucket-serial data stream to processing module. Since the processing module can process the data at the rate almost same as that of the input data stream, performance of GRACE mainly lies upon the effective transfer rate of the stream.

Memory module is required to provide a large space for temporary relations and result relations. Semiconductor RAM and magnetic bubble memory are candidates for the medium of the memory module. Since the usage of RAM is straightforward, here we discuss the bubble memory based organization of the module. The pseudo random access mechanism of major/minor magnetic bubble memory is quite different from that of disk. Namely the unnecessary record

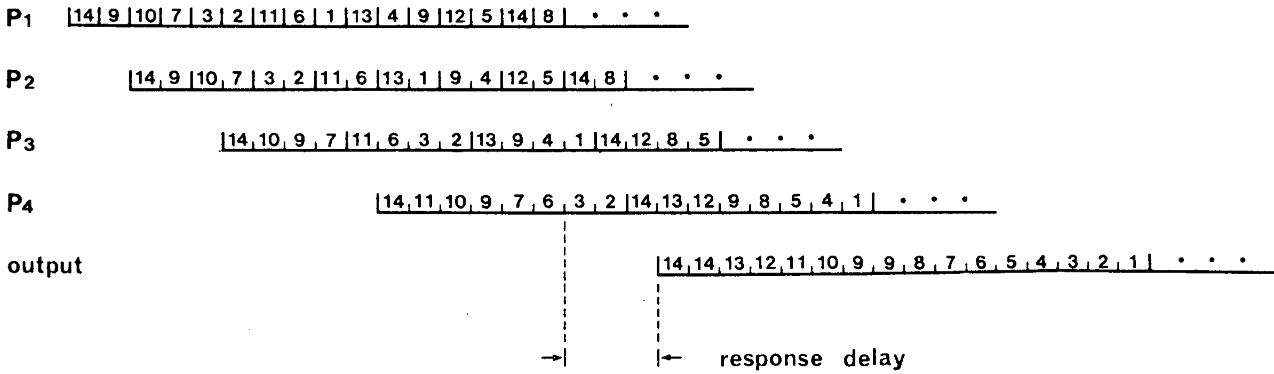


Fig. 4 Sorting Process Overview Of Stream Driven Sorter

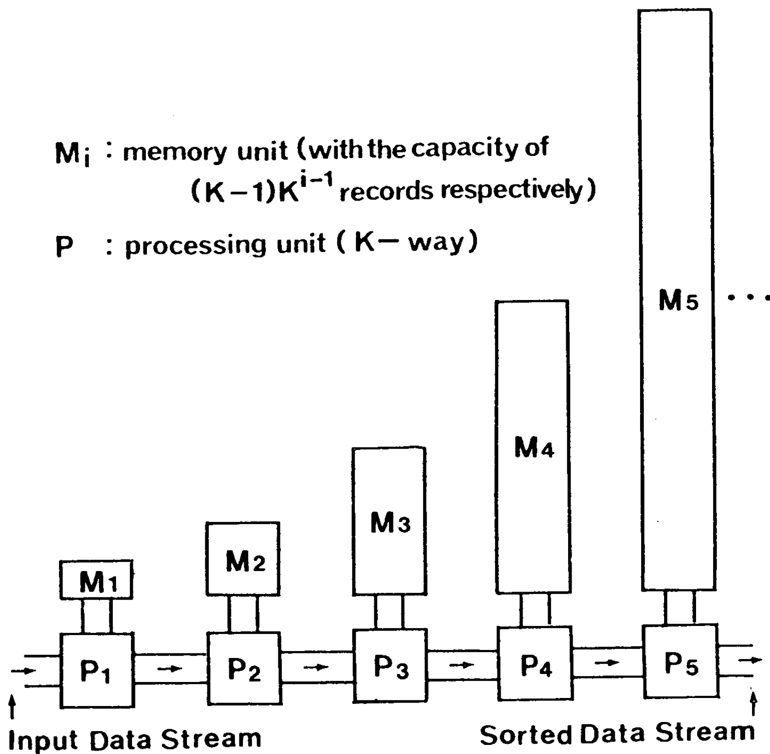


Fig. 5 Global Architecture Of Stream Driven Sorter

stored over minor loops can be skipped in a period of 1 bit field rotation time, while a head has to traverse all the fields of the record in disk whether the record is necessary or not. This quick skip mechanism improves the access time and the effective transfer rate of the magnetic bubble memory. Especially in data base processing, instead of individual record access by an address, set oriented access is more usual where a set of records satisfying some conditions are required and the retrieval sequence itself is not a matter. Therefore the effective transfer rate rather than the access time is much more influential.

We modified a conventional major/minor magnetic bubble memory to obtain heigher transfer rate. In our bubble chip organization, small buffer loops are added between major line and minor loops, in which records can be buffered while another record is output. Record output through major line and transfer of records from minor loops to buffer loops can be overlapped and this can improve effective transfer rate of the bubble for the set oriented access of records. Since the buffer loop length is short, tuples can be output almost continually. A block replication/transfer gate and swap gate are used between major line and buffer loops and between buffer loops and minor loops respectively. With swap gate, look ahead buffering of the bucket into buffer loops is possible and the bucket serial stream can be generated efficiently.

The gates of the bubble memory are controlled using mark bit RAM which synchronizes the magnetic field rotation of the bubble. The hashed value of a tuple is stored in this RAM when it enters memory module. The bubble control unit controls the gates at the appropriate time by referring the hashed value of the record in the RAM. To generate bucket-serial data stream, all the tuples with the same hashed value which belong to a bucket have to be processed before the next bucket processing. So the number of tuples in each bucket is maintained.

We described bubble memory based organization for the memory module. The modified magnetic bubble memory chip with buffer loops is available at present. But it's also possible to use semiconductor RAM. Future development of the these memory technology has to be observed.

5.3 Disk Module

GRACE adopts disk as medium for the secondary storage which makes the capacity of the data base very large. Relations are stored on the disk and are staged into memory modules through the staging ring. Disk module employs a filter processor which evaluates simple predicate on the fly where all the track can be read in parallel. This mechanism can be found in many of the machines proposed so far such as RAP and TIP in DBC. The irrelevant attributes in the tuples are also removed. This reduce the amount of the data which must be transferred between disk module and memory module. If the complexity of the predicate is beyond the filter processor's ability, it can be processed in processing module. The data stream which passed through the filter processor is led into the

hashing unit which can be also found in processing module, where tuples are hashed over the attributes for the next operation such as join or projection. After a hashed value is added to the tuple, it is sent to memory modules through the staging ring.

As for the storage organization in disk, various kinds of mapping from the logical structure of relation into the physical storage media are possible. Many file organizations have been investigated so far. In most of data base machines such as RAP there is no auxiliary data structure and the tuples are arranged in the form like a sequential file. This eliminates the software burden caused by a complex access method in secondary storage mapping. However any retrieval requires to scan all the tuples of the relation. As the data base expands, full scan of the large file becomes prohibitive. On the other hand DBC employs structure memory(SM) where indexes are maintained and a partitioned content addressable memory is realized with disk and TIP. GRACE adopts multi-dimensional clustering technique in secondary storage mapping and reduces the number of pages to be retrieved. Disk module also manage the auxiliary data structure consistently.

6. Summary

According to the classification by Paula Hawthorn[23], some of the data base application provides the join dominant environment. For a simple query containing selection, the previous machines based on exhaustive matching with full scan of the file would suffice. For a complex query which has many joins and projections, however, it is difficult to attain a high performance with the ordinary approach of the filter processor. GRACE adopted a novel relational algebra processing algorithm based on hash and sort, and can execute not only simple query but also complex one comprising many joins or set operations rapidly. While the data streams are fixed in a secondary storage in the previous machines, the clustered data streams appropriate for the given operation are generated dynamically in GRACE and can be processed keeping up with the stream. This allows a complex operation also to complete with one data flow cycle, and moreover due to the operator level pipeline, time overhead caused by hashing is effectively canceled.

In this paper, we have described the processing algorithm and it has been shown that GRACE can execute a relational algebra complex very efficiently. The abstract architecture and execution overview on it are only briefly explained and the details about the actual implementations, data stream control mechanism, and performance evaluation are prepared in the future paper.

Acknowledgement

The authors would like to acknowledge S.Suzuki, H.Akamatsu, Y.Ogi, S.Fushimi, and S.Sakai for their valuable contribution to the data base machine project.

References

1. Slotnick,D.L., Logic per Track Devices, Advances in Computers, Vol.10, J.Tou,ed., Academic Press, New York, pp.291-296 (1970)
2. Ozkarahan,E.A., Schuster,S.A. and Smith,K.C., RAP-An Associative Processor for Data Base Management, Proc. AFIPS NCC, Vol.45, pp.379-387 (1975)
3. Ozkarahan,E.A., Shuster,S.A. and Sevcik,K.C., Performance Evaluation of a Relational Associative Processor, ACM Trans. Database Syst., Vol.2, No.2, pp.175-195 (1977)
4. Oflazer,K. and Ozkarahan,E.A., RAP.3-A multi-micro cell architecture for the RAP database machine, Proc of the Int. Workshop on High Level Language Computer Architecture, pp.108-119 (1980)
5. Copeland,G.P., Lipovski,G.J. and Su,S.Y.W., The Architecture of CASSM: a Cellular System for Non-numeric Processing, Proc. 1st Annu. Symp. Computer Architecture, pp.121-128 (1973)
6. Su,S.Y.W., Nguyen,L.H., et al., The Architectural Features and Implementation Techniques of the Multicell CASSM, IEEE Trans. Comput. Vol.C-28, No.6, pp.430-445 (1979)
7. Su.S.Y.W., On Logic-per-Track Devices: Concept and Applications, IEEE COMPUTER, Vol.12, No.3, pp.11-25 (1979)
8. Uemura,S., Yuba,T., Kokubu,A., et al., The Design and Implementation of a Magnetic-Bubble Database Machine, IFIP 80, pp.433-438 (1980)
9. Oliver,E.J. and Berra,P.B., RELACS A Relational Associative Computer System, Proc. of the Fifth Workshop on Computer Architecture for Non-Numeric Processing, pp.106-114 (1980)
10. Berra,P.B. and Oliver,E.J., The Role of Associative Array Processors in Data Base Machine Architecture, IEEE Computer, Vol.12, No.3, pp.53-61 (1979)
11. DeWitt,D.J., DIRECT-A Multiprocessor Organization for Supporting Relational Database Management Systems, IEEE Trans. Comput., Vol. C-28, No.6 (1979)
12. DeWitt,D.J., Query Execution in DIRECT, Proc. ACM-SIGMOD 1979, pp.13-22 (1979)
13. Kung,H.T. and Lehman,P.L., Systolic (VLSI) Arrays for Relational Database Operations, Proc. of ACM-SIGMOD pp.105-116 (1980)
14. Song,S.W., A Highly Concurrent Tree Machine for Database Applications, Proc. of the 1980 Int. Conf. on Parallel Processing, pp.259-268 (1980)
15. Banerjee,J., Hsiao,D.K. and Kannan,K, DBC-A Database Computer for Very Large Databases, IEEE Trans. Comput., Vol.C-28, No.6, pp.414-429 (1979)
16. Menon,M.J. and Hsiao,D.K., Design and Analysis of a Relational Join

Operation for VLSI, Proc. Int. Conf. on Very Large Data Bases, pp.44-55 (1981)

17. Tanaka,Y., Nozaka,Y., et al., Pipeline Searching and Sorting Modules as Components of a Data Flow Database Computer, IFIP 80, pp.427-432 (1980)

18. Oda,Y., Database Machine Architecture using Data Partitioning Network, IECEJ Technical Group Meeting, EC80-72 (1981) (in Japanese)

19. Babb,E, Implementing a Relational Database by Means of Specialized Hardware, ACM Trans. Database Syst., Vol.4, No.1, pp.1-29 (1979)

20. McGregor,D.R., Thomson,R.H. and Dawson,W.N., High Performance Hardware for Database Systems, Systems for Large Data Bases, North-Holland, pp.103-116 (1976)

21. Kitsuregawa,M., Suzuki,S., Tanaka,H., and Moto-oka,T., Application of Hash to a Data Base Machine, The 23rd Information Processing Society National Convention (1981) (in Japanese)

22. Kitsuregawa,M., et al., Relational Algebra Machine based on Hash and Sort, IECEJ Technical Group Meeting, EC81-35 (1981) (in Japanese)

23. Hawthorn,P., The Effect of Target Applications on the Design of Database Machines, Proc. of ACM-SIGMOD, pp.188-197 (1981)

24. Kitsuregawa,M., Fushimi,S., Kuwabara,k., Tanaka,H., and Moto-oka,T., Organization of Pipeline Merge Sorter, IECEJ Technical Group Meeting, EC82-32 (1982) (in Japanese)

25. Todd,S., Algorithm and Hardware for a Merge Sort Using Multiple Processors, IBM J.RES.DEVELOP., Vol.22, pp509-517 (1978)