数理解析研究所講究録
第482巻 1983年 23-40

Relational Algebra Machine

GRACE

University of Tokyo
Faculty of Engineering

Masaru Kitsuregawa
Hidehiko Tanaka
Tohru Moto-oka

1. Introduction

'Logic per Track' concept by Slotnick[1] is considered to be the origin of Data Base Machine(DBM). 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. This 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. 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]. 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.

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 because of its cost. 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 page the 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 allocated 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 which activates many comparators in pipeline fashion 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(logN) 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 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. Relational Algebra Execution based on Hash

Hash is well-known as a direct access method in the data base storage organization, where necessary records can be obtained with almost one access provided that load factor remaines low, and this is one of the fastest access method. Besides these static data management method, hash technique is applied dynamically in data base machine. Following two methodologies can be identified in supporting relational algebra operations of heavy load.

1. *Joinability Filter Approach*
2. *Clustering Approach*

The first approach is taken in CAFS where a hash bit array is used. 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.

In the second approach which we adopted, not the number of tuples in two relations but the load of join itself can be reduced by the clustering feature of hash operation. 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. 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. Accordingly, 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, which we discuss in the next section.

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

3. Parallel execution of the hash based method and its some problems

In the previous section, shown was the fact that great reduction of processing load of join and projection, etc. is actualized by dividing relations into many buckets through hash. Here, we will consider how to materialize this method for 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. Following problems can be identified in designing a relational algebra machine which realizes this bucket parallel processing.

1) Utilization of Bank Parallelism

If a relation is stored over many memory banks, execution could be faster than in the case where a relation is stored in one memory bank.

In database processing, data stream is a main constituent, and several kinds of processing are applied along the stream. However, even if the data stream could be processed following the stream, that is, $O(N)$ processing be realized, the stream may become very long in the case of large databases, 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.

2) $O(n)$ processing within a bucket

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.

3) Bucket Allocation to Processors

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.

4) Nonuniformity of Generated Buckets

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 inherent problems and to provide means to handle such an exceptional event as bucket overflow.

4. Design Consideration

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

1) On 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, which we mean here by 'bank parallelism'. For example, DBC Join processor cannot execute join in $O((M+N)/n)$ time but in $O(M/n+N)$ time. In DIRECT, if n*m processors 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 in section 2 the hash based join method, 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, where 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, 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. 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, which is not so large provided that the hash function is random. 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) On 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. 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 follow 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. 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) On Bucket Allocation

Bucket serial processing is necessary under the environment of finite resources of processors. And as a whole 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 even at present time it is found in a disk cash, so it will be feasible to use RAM as the staging buffer. However the capacity 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) On Nonuniformity 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 capacity less than that 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.*

5. Abstract Architecture of GRACE and Execution Overview

5.1 Abstract Architecture

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

SDM employs disks as secondary storage devices and stores the relations on them. The disk can transfer data from all heads of a cylinder in parallel, and it is equipped with filter processing function which includes 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.

DSG employs RAM or magnetic bubble memory and provides the working space. To DSG, the relation filtered and hashed on SDM is staged and the result relation generated in DSP is returned. From DSG, the bucket serial data stream is generated and sent to DSPs.

Each DSP consists of hardware sorter, filter processor, data manipulation unit, and hash unit etc. It processes a bucket sent from DSG using such units and produces a result relation.

5.2 Query Execution on GRACE

Here we consider how the query is executed on GRACE. The query is assumed to be a complex one and have 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 SDM to DSG 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 took 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.

5.3 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. As many cycles as the height of a query tree which could be optimized as low as possible would be required to get the result.

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 the any DBMs proposed so far.

6. Summary

Most of the data base machines proposed so far adopted a filter processor as their basic component, and selects necessary records by evaluating qualification predicates based on exhaustive matching with full scan of the file, by which the overhead caused by the auxiliary data management such as indices is intended to be reduced. For a simple query which includes only selection, this approach would suffice. There are join-dominant environment[23], however, and 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.

According to the taxonomy by O.H.Bray[24], GRACE is classified into MPCS (Multiple Processor Combined Search) machine since direct search by the filter processors in SDM and indirect search by DSP and DSG are combined and both are processed by using not a single but multiple processors. GRACE filters out the unnecessary records on the fly in SDM and processes the survived records on DSG and DSP with repetitive data flow cycles, namely it employs the combined architecture which integrates the merits of direct search and indirect search.

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 such as the structure of hardware sorter, bucket-serial data stream generator in magnetic bubble memory unit, and data stream control mechanism are prepared in the future paper.

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 Techinical 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. Bray,O.H. and Freeman,H.A., Data Base Computers, Lexington Books, (1979)
25. Wah,B.W. and Yao,S.B., DIALOG-A Distributed Processor Organization for Database Machine, Proc. AFIPS NCC, Vol.49, pp.243-253 (1980) (not cited)
26. Leilich,H.O., Stiege,G. and Zeider,H.C., A Search Processor for Data Base Management Systems, Proc. Int. Conf. on Very Large Data Bases, pp.280-287 (1978) (not cited)

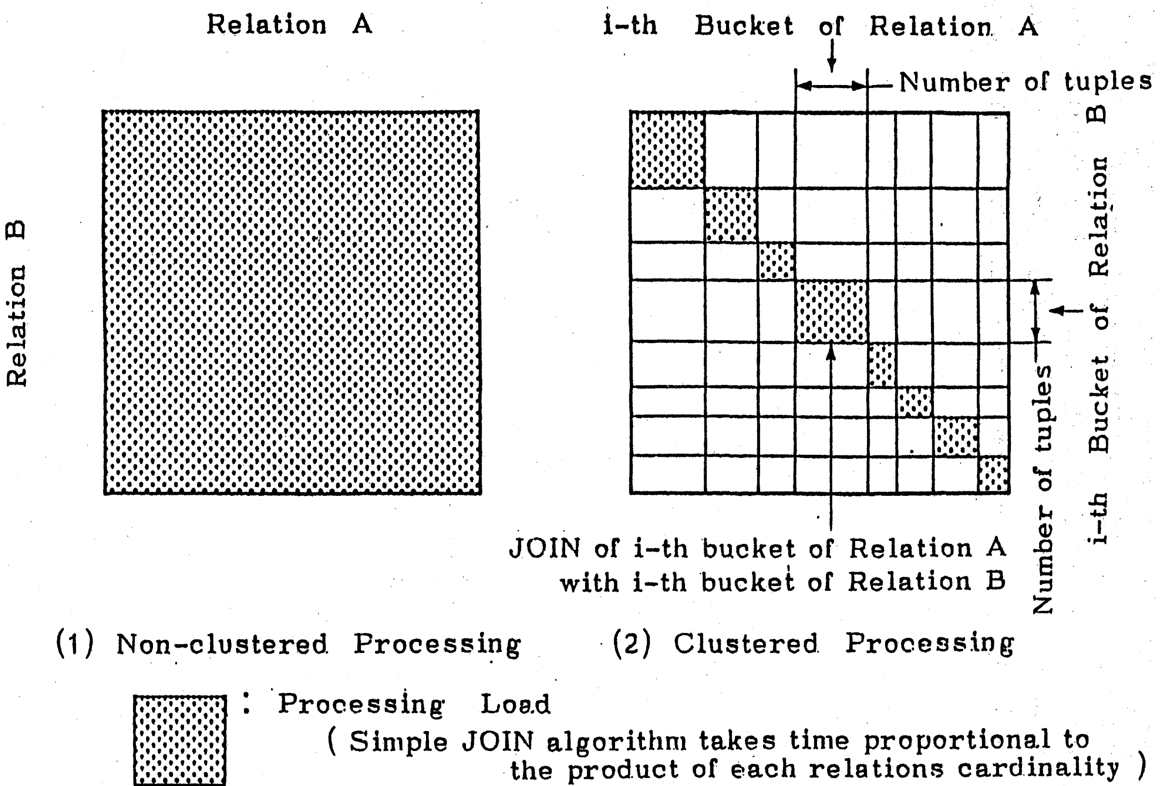

Fig. 1. Clustering Effects By Hashing In JOIN Operation

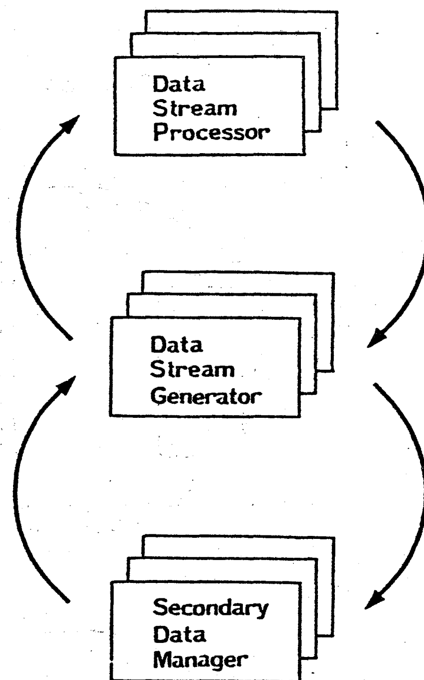

Fig.2.1 Abstract Architecture Of GRACE

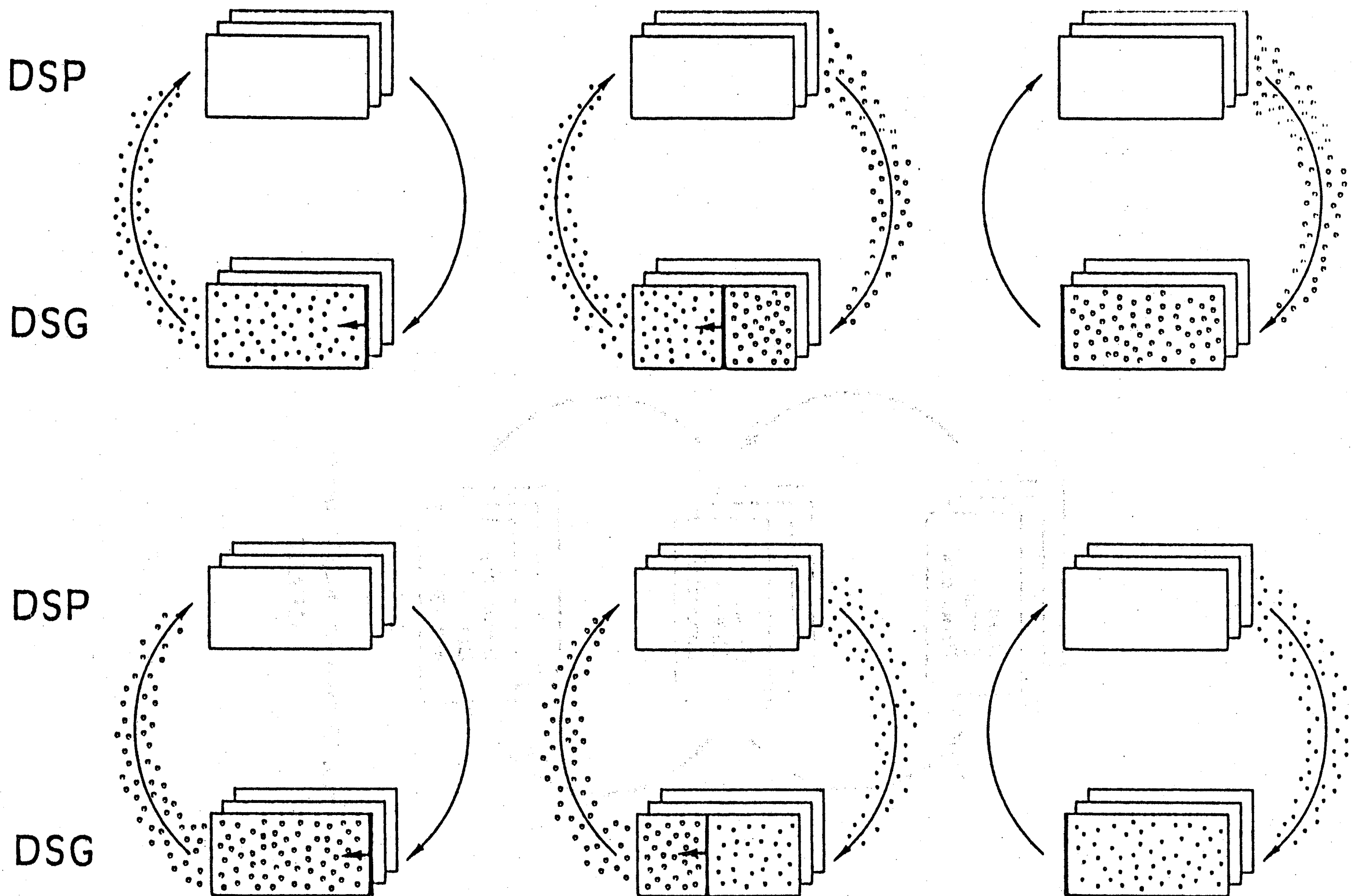

Fig. 2.2 Execution Overview