UC Irvine
ICS Technical Reports

Title
AGM, a dataflow database machine

Permalink
https://escholarship.org/uc/item/831110t6

Authors
Bic, Lubomir
Hartmann, Robert L.

Publication Date
1984

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Submitted for publication

Notice: This Material may be protected by Copyright Law (Title 17 U.S.C.)

AGM: A Dataflow Database Machine[1]

by

Lubomir Bic

Robert L. Hartmann

Department of Information and Computer Science

University of California, Irvine

85-24

October 1984

Technical Report No. 85-24

[1] *This work was supported by the NSF Grant MCS-8117516: The UCI Dataflow Databases Project.*

ABSTRACT

In recent years, a number of database machines consisting of large numbers of parallel processing elements have been proposed. Unfortunately, one of the main limitations to parallelism in database processing is the I/O bandwidth of the underlying storage devices. One way to solve this problem is to use multiple parallel disk units. The main problem with this approach, however, is the lack of a computational model capable of utilizing the potential of any significant number of such devices.

This paper presents a database model which is based on the principles of data-driven computation. According to this model, the database is represented as a network in which each node is conceptually an independent processing element, capable of communicating with other nodes by exchanging messages along the network arcs. To answer a query, one or more such messages, called tokens, are created and injected into the network. These then propagate asynchronously through the network in the search of results satisfying the given query.

To investigate the performance of the proposed system, we have implemented the model on a simulated computer architecture. The results of the simulation experiments indicate that the model is capable of exploiting the potential I/O bandwidth of a large number of disk units as well as the computational power of the associated processing elements.

1. Introduction

A typical database is an organized collection of data kept on secondary storage devices such as magnetic discs. To increase efficiency during processing, a number of multiprocessor database machines have been proposed. Unfortunately, since all data to be processed must first be transferred into primary memory, the available I/O bandwidth provided by the disk drives is the main limitation in the design of database machines. For example, Agrawal and DeWitt /AgDeW84/ have shown that for 2 disk drives (IBM 3350) not even 10 query processors are adequately utilized.

To alleviate the problem of I/O bandwidth, several approaches can be taken:

1. *Head-per-Track devices.* A read/write head is associated with each track of the disk thus virtually eliminating the seek time. Furthermore, some amount of processing may be performed directly by marking data on the disk, without having to move it to primary memory. An example of such a system is the Relational Associative Processor (RAP) /Sch78/. The main problem with this approach is a relatively high cost of implementation and a number of technological difficulties in supporting the special-purpose circuitry of the read/write mechanisms. Furthermore, the processors associated with each track must be very simple due to space and power supply limitations, thus a large amount of processing must be done outside of the database machine by a host computer.

2. *Parallel Readout Disks.* Disks may relatively easily be extended to read all tracks of a given cylinder in parallel. While the potential I/O bandwidth is increased by an order of magnitude, it is rarely the case that a query will be able to utilize the content of an entire cylinder. Hence the actual bandwidth is reduced to the useful portion of the data actually retrieved with each operation.

3. *The Use of Multiple Disks.* While the potential I/O bandwidth increases with the number of independent disk drives, the main problem is that of utilizing this potential, i.e., the choice of an adequate database model. Network-based (DBTG) models /TaFr76/ are not suitable to parallel processing due to their low level of interaction with the database. The user is actually seen as a 'navigator', who guides a sequential thread of computation through the database.

The Relational Model, on the other hand, permits queries to be expressed in a non-procedural manner. Unfortunately, other features of the model make an efficient parallel implementation difficult. In particular, binary relational operators such as join, set intersection, or set difference, require that the involved relations be sorted or otherwise preprocessed in order to avoid comparing each element of one relation against all elements of the other. Hence one of the most difficult problems is to decide how to distribute the different relations over the available disks and how to select the site at which a particular operation should be performed.

The above discussion suggests that, while increasing the I/O bandwidths is a *necessary* precondition, it is not *sufficient* to guarantee better performance, regardless of the number of processors provided by the architecture. Rather a different model of computation must be devised, that would be capable of exploiting the potential parallelism resulting from a large number of independent disk units. Such a model must satisfy the following requirements:

- It must be possible to process many requests concurrently. Without this condition a large portion of the total I/O bandwidth would be wasted since each request will, in general, involve several phases not all of which require disk access.
- There must be no centralized control to distribute computation to and to supervise the progress of individual processing elements. The elimination of the

control bottleneck must not, however, prevent data integrity and back-up policies from being enforced.

- The model must be able to tolerate the long latencies in accessing data, resulting from the relatively slow speed of secondary storage devices. That is, while data is being transferred from a disk, the processing element must be able to work on some other task. In addition, it must be able to tolerate the fact that data will not necessarily be arriving in the order in which requests were issued.

In this paper we present a model for data representation and manipulation that satisfies the above requirements. This model, referred to as the *Active Graph Model*[1] (AG-Model, for short), is based on the principles *dataflow systems* /Com82, TBH82/ which depart from the sequential, one-instruction-at-a-time concept of Von Neumann computers by enforcing data-driven functional computation. This model has been implemented on a simulated computer architecture consisting of a large number of disk units, each equipped with a separate processing element. The model together with the underlying architecture will be referred to as the *Active Graph Machine* (AGM). Using a series of simulation experiments, we will demonstrate that the system is capable of exploiting the potential I/O bandwidth of a large number of disk units as well as the computational power of the associated processors.

2. The Model

2.1 Data Representation

To represent the database, we adopt the basic ideas of the Entity-Relationship model /Che76/ which perceives information as collections of *entities* and *relationships.* An entity-relationship diagram is used to describe a particular database. In

[1] The justification for selecting this name will be given in Section 2.2.

this representation, entity sets and relationship sets are shown as rectangular and oval shaped boxes, respectively; arcs are used to indicate the participation of entities in relationships. *Attributes* may be associated with both entities and relationships. They are defined as mappings between the entity or relationship sets and value sets.

Internally, entity and relationship sets are represented as follows:

- Each element n of an entity or relationship set is represented by a *member node.* It consists of a key value k_n and a set of attribute values. Key values are unique within each set, thus any member node is uniquely identified by the pair (S, k_n).
- For each entity or relationship set S there exists a unique node called *master node.* All elements of the set S are connected to their corresponding master node via arcs.
- All arcs in the system are represented as bi-directional pointers.

Figure 1(a) shows a sample database comprising three entity sets, PROFessors, COURSEs, and STUDents, interconnected via the corresponding relationships TEACH and ENROLLment. (Heavy lines indicate the flow of tokens as will be explained in Section 2.4.)

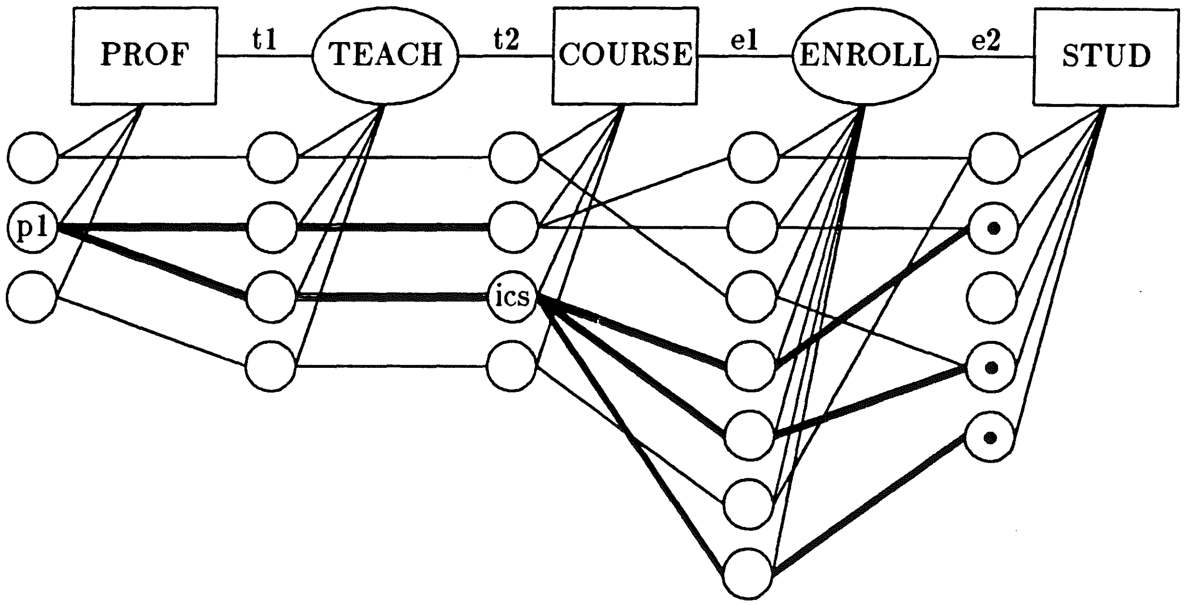

Figure 1(a)

2.2 A Dataflow View of Processing

The assumption implicit to most database systems is the existence of an outside agent – a processor – which accesses and manipulates the data stored on secondary memory. Under this Von Neumann model of computation it is very difficult to exploit parallelism, primarily due to problems of synchronizing any significant number of processing elements. In order to overcome these difficulties, inherent to conventional models of computation, we adopt the dataflow point of view which eliminates the need for any centralized control. At the model level, we do not view the database graph as a passive representation of entities and relationships. Rather, each node is viewed as an *active* element capable of receiving, processing, and emitting value tokens traveling asynchronously along the graph arcs. Similar to general dataflow systems, the operation of each node is triggered solely by the arrival of tokens. The terms 'Active Graph Model' and 'Active Graph Machine' thus derive from the fundamental assumption that each node of the graph is, logically, an autonomous 'processing element'.

2.3 The Data-Manipulation Language

One of the main requirements of the model is that it provides a high-level non-procedural language to specify all requests – queries and updates – to be performed against the database. The language developed for the AG-model /Har84/ has its roots in the query language CABLE (ChAin-Based LanguagE), proposed for the Entity-Relationship model /Sho78/. Each request consists of two parts, referred to as *selection* and *operation*. The selection part is a collection of 'beads' interconnected into a *tree* structure. Each bead names one of the entity or relationship sets and some restriction to be applied to elements of that set. These restrictions may be based on attribute values of the set itself or they may be based on the existence of arcs between nodes of various sets. Hence the tree of beads forming a request may be viewed as a pattern to be superimposed onto the database graph for the purposes of selecting nodes that satisfy the given restrictions.

The processing of the tree starts from its leaves and converges onto its root. At each set the corresponding restrictions are applied and, if necessary, specified attribute values are extracted and carried along by tokens. The root node represents the final target set at which the actual operation (e.g. data retrieval, update, etc.) is performed. As will be discussed in Section 2.4, the entire selection process is carried out by tokens propagating asynchronously through the database graph. Before discussing the details of token propagation we present the data manipulation language more formally.

A request is a collection of beads B_S followed by an operation O, i.e.,

$$B_{S1}, ..., B_{Sn} : O$$

Each bead B_S has the form

$$\langle in\_arc_1, ..., in\_arc_n\rangle S[\rho; \textbf{export } \varepsilon]\langle out\_arc\rangle$$

where the individual components have the following meaning:

- $in\_arc_1, ..., in\_arc_n$ and out_arc are names of arcs occurring in the entity-relationship diagram. The collection of beads are required to form a tree[2] as follows. There must be exactly one bead with no *out_arc*; this becomes the *root* of the tree. For each in_arc_i of this and of all other beads there must exist another bead whose *out_arc* matches in_arc_i. Beads with no *in_arc*s then become the leaves of the tree. We will refer to this tree as the *request tree* and

[2] By permitting more than one *out_arc* with each bead, a DAG (directed acyclic graph) could be formed, rather than a tree. To simplify the subsequent discussion, we have restricted ourselves to using only one *out_arc*.

to its nodes as *set nodes*, to distinguish them from nodes (master or member) constituting the database graph.

- S is the name of an entity or a relationship set occurring in the entity-relationship diagram.
- ε is an algebraic formula which specifies the attribute values to be carried from nodes of the set S to nodes of the next set, say S' of the tree, along the corresponding *out_arc*; these are said to be *exported* to S'. The keyword **export** is used to visually separate the two clauses ρ and ε, each of which could be a complex expression.

  ε is composed of *selectors* and the set-combining operators union ($\cup$), intersection ($\cap$), difference ($\setminus$), and Cartesian product ($\times$).

  Selectors are lists of the form $(x_1, ..., x_n)$, where each x_j is either an *attribute designator*, an *aggregate function*, or a constant c.
  An attribute designator can be one of the following:

  - $*.i$, which specifies the value of the i-th attribute of the set S
  - $in\_arc_i.j$, which refers to the value of the j-th attribute exported by the set connected to S via in_arc_i.

An aggregate function may be one of the following:

- COUNT(in_arc_i), AVERAGE(in_arc_i), MIN(in_arc_i), or MAX(in_arc_i), where in_arc_i is one of the input arcs of the set S. Each of these functions is applied to the results exported by the previous set along in_arc_i.

ε is then defined recursively as a formula, where:

- Every selector is a formula.
- If x and y are formulas then (x), $x \cup y$, $x \cap y$, $x \setminus y$, and $x \times y$ are formulas.

- ρ is a restriction expression to be applied to all nodes s of the set S; a node s is called ***selected*** if it satisfies the restriction expression ρ. Each ρ is a Boolean formula composed of ***elementary restrictions*** and logical operators (not, and, or).

  Elementary restrictions have the form: $op_1\ \theta\ op_2$, where

  ◇ θ is one of the relational operators $=$, $\neq$, $<$, $>$, $\leq$, $\geq$, and

  ◇ each op_i is one of the following:

    ∘ an attribute designator of the same form as defined above, i.e., $*.i$ or $in\_arc_i.j$,

    ∘ one of the aggregate functions defined above, i.e., COUNT(in_arc_i), AVERAGE(in_arc_i), MIN(in_arc_i), or MAX(in_arc_i), as defined above,

    ∘ or a constant c,

    ∘ the function $ARC\_COUNT(in\_arc_i)$, which, for a given node s, returns the number of arcs of type in_arc_i, connected to that node.

  Each ρ is then defined recursively as a formula, where:

  ◇ Every elementary restriction is a formula.

  ◇ If x and y are formulas then (x), $\neg x$, $x \wedge y$, and $x \vee y$ are formulas.

The following examples illustrate the purpose of the individual component and demonstrate the expressive power of the selection process:

Assume that the sets of Figure 1(a) have he following attributes:

PROF:	KEY, NAME, RANK
TEACH:	KEY
COURSE:	KEY, NAME, SUBJECT

ENROLL: KEY

STUD: KEY, NAME, STAT

Example 1. The query 'find all ics courses with more than 20 graduate students; output the course name' is expressed as follows:

STUD[$*$.3='GRAD']⟨e1⟩

⟨e1⟩ENROLL[ ]⟨e2⟩

⟨e2⟩COURSE[$*$.3='ICS' $\wedge$ COUNT(e2)>20, **export** $*$.2]: OUTPUT

This query involves three sets connected via the roles e1 and e2. Proceeding from the set STUD to the set COURSE, the restrictions are applied as follows. First, all elements of STUD satisfying the restriction $*$.3='GRAD' (i.e. STATUS='GRAD') are selected. As a next step, elements of the set ENROLL are selected; since no explicit restriction is specified, all elements connected via an arc e1 to at least one of the selected nodes of STUD are selected. Finally, a selection is performed on the set COURSEs. Each element in that set must meet the following requirement to be selected: it must satisfy the restriction $*$.3='ICS' (i.e., SUBJECT='ICS') and it must be connected via an arc e2 to at least twenty of the selected elements of the previous set ENROLL. The **export** clause then specifies that the value of the second attribute (NAME) is to be extracted from each of the selected element and output, as specified by the operation part of the query.

Example 2. The query 'find all ics courses with only graduate students; output the course name' is expressed as follows:

STUD[$*$.3='GRAD']⟨e1⟩

⟨e1⟩ENROLL[ ]⟨e2⟩

⟨e2⟩COURSE[$*$.3='ICS' $\wedge$ COUNT(e2)=ARC_COUNT(e2), **export** $*$.2]: OUTPUT

This query is very similar to the one of Example 1; the only distinction is the restriction COUNT(e2)=ARC_COUNT(e2), which states that an element *s* of the set COURSE is selected only when all elements of the previous set to which *s* is connected via an e2 arc have been selected. Courses in which any undergraduate students are enrolled do not satisfy this restriction and are therefore not selected.

Example 3. The query 'find all ics courses with more than 20 graduate students and taught by an associate professor; output the course number and the course name' is expressed as follows:

PROF[*.3='ASSOC']⟨t1⟩
⟨t1⟩TEACH[ ]⟨t2⟩
STUD[*.3='GRAD']⟨e1⟩
⟨e1⟩ENROLL[ ]⟨e2⟩
⟨t2,e2⟩COURSE[*.3='ICS' $\wedge$ COUNT(e2)>20; **export** (*.1, *.2)]: OUTPUT

This query forms a tree where PROF and STUD are the leaves and COURSE is the root. The selection proceeds independently along the two branches PROF-TEACH-COURSE and STUD-ENROLL-COURSE. At each set, elements satisfying their corresponding restriction are selected. In the final set, COURSE, an element *s* must satisfy the following criteria in order to be selected: its SUBJECT attribute must be 'ICS', it must be connected to at least 20 (graduate) students (via elements of the set ENROLL), and it must be connected to an associate professor (via an element of the set TEACH). The course numbers and names of the selected nodes are then output.

The operation O constituting each request may specify the following basic operations:

• *OUTPUT*(*opt*). This operations causes the attribute values exported by the elements of the root set S to be output. The three queries discussed above were examples of using this operation. The optional parameter *opt* may specify one of the functions COUNT, AVERAGE, MIN, or MAX, in which case the appropriate aggregate value is calculated and output, or it may specify the function SORT(i), in which case the results are output sorted by the attribute value i. For example, to determine the number of graduate students, rather than to retrieve any of their attribute values, the following query would be used:

STUD[*.3='GRAD']: OUTPUT(COUNT)

• $UPDATE(a_1, ..., a_n)$. This operation causes each node of the root set S selected by the preceding selection operation to modify itself as follows. Each a_i is an assignment of the form $*.i \leftarrow$ ***arithm_exp***, indicating that the attribute $*.i$ is to be replaced by the value of the arithmetic expression ***arithm_exp***, defined recursively as follows:

◇ an attribute designator, which can have the form $*.i$ or $in\_arc_i.j$ as defined earlier, is an arithmetic expression,

◇ if x and y are formulas then (x) and $x \vartheta y$ are arithmetic expressions, where ϑ is an arithmetic operator $(+, -, *$, etc.).

For example, to increase the salary of all associate professors by 3%, the following query could be used:

PROF[*.3='ASSOC']: UPDATE($*.4 \leftarrow (*.4) * 1.03$)

• $INSERT\_NODE(a_1, ..., a_n)$. This operation causes one new node to be inserted into the root set S of the request. This node is automatically connected to its master node via an arc. Each a_i has the same form as in the case of the *update* operation – it specifies the new attribute values of the inserted node. In the simplest case the selection information constituting the request will consists of only one bead – the root set S itself. For example, to insert a new student, the following query could be used:

STUD[ ]: INSERT($*.1 \leftarrow$ '999', $*.2 \leftarrow$ 'JANE JONES', $*.3 \leftarrow$ 'GRAD')

If a tree consisting of more than one set node is specified then new arcs between the inserted node and nodes of other sets are established as well. If S' is a set node connected to the root set S via its *out_arc* then a new arc is established between the inserted node and each selected node of the set S'.

• *DELETE_NODE.* This operation causes each node of the root set S selected by the preceding selection operations to delete itself. All arcs connecting such nodes to any other node are removed as well. For example, to delete all undergraduate students enrolled in a course C20, the following query could be used:

COURSE[*.2='C20']⟨e2⟩
⟨e2⟩ENROLL[ ]⟨e1⟩
⟨e1⟩STUD[*.3='UNDERGRAD']: DELETE

• *INSERT_ARC.* This operation causes the insertion of new arcs between existing nodes. If S' is a set node connected to the root node S via its *out_arc* then a new arc is established between each selected node of the set S and each selected node of the set S'.

• *DELETE_ARC.* This operation causes existing arcs to be deleted. If S' is a set node connected to the root node S via its *out_arc* then all arcs between the selected nodes of the set S and the selected nodes of the set S' are deleted.

2.4 Execution of Requests

As described in Section 2.3, each request is a tree structure consisting of beads. To process a request, the system performs the following tasks:

• For each bead B_S, find the elements of the set S that satisfy the corresponding restriction ρ. As discussed in Section 2.3, this restriction may be based on attribute values of the set S itself, or it may be based on some relationship between attributes of the set S and attributes of some other set S', preceding S in the request tree.

- Perform the operation on the selected elements of the set corresponding to the root of the tree.

To accomplish these tasks, the request is translated into a collection of ***tokens*** which are injected into master nodes of the database graph; recall that these are active elements, capable of receiving and processing tokens. We employ two types of tokens as follows:

Each bead B_S, i.e., $\langle in\_arc_1, ..., in\_arc_n \rangle S[\rho; \textbf{export } \varepsilon] \langle out\_arc \rangle$, is placed on a separate token, called ***restriction*** token, and is injected into the corresponding master node of the set S. From there it is replicated along existing arcs to all member nodes of that set. The second type of token, called ***sweep*** token, is used to transmit information between elements of two sets. Initially, one sweep token is created by each node of a leaf set of the given request tree; from there the sweep tokens are propagated through the tree toward the root. Each sweep token has the form:

$$\bar{\rho}, \bar{\varepsilon}$$

where

- $\bar{\rho}$ denotes the value (True/False) of the restriction ρ, indicating whether the node emitting that sweep token has been selected by the restriction ρ.

- $\bar{\varepsilon}$ denotes the value of the expression ε; this represents the collection of attribute values exported by the node emitting that sweep token.

The processing of each request is completely data-driven. That is, once the restriction tokens are injected into the database graph, their propagation as well as the creation and propagation of sweep tokens is governed by the following procedures performed by individual nodes receiving tokens:

Each node s of a set S involved in a request will receive exactly one restriction token from its master node. If no *in_arcs* are specified, the node is, by definition, a leaf of the request tree, implying that both the restriction ρ and the export expression ε may be based only on attribute values internal to the node s itself. The node determines the corresponding values $\bar{\rho}$ and $\bar{\varepsilon}$ and constructs a sweep token consisting of these two values. It sends a copy of this token along all arcs matching the *out_arc* specified in the restriction token.

All nodes receiving that token will perform a similar step: the values of the corresponding ρ and ε expressions are determined and a sweep token is constructed and forwarded along the appropriate *out_arcs* to the next set. The evaluation of ρ and ε is, however, more complicated than in nodes of a leaf set since both may be based on internal attribute values as well as on values carried by the received sweep tokens. We can distinguish the following four cases according to the possible atoms constituting ρ and ε, as defined in Section 2.3:

- $*.i$ refers to attribute values of the node s itself; these are kept with the node and thus are readily available.
- in_arc_i refers to attributes exported by sets preceding S in the tree; these are carried by sweep tokens arriving along in_arc_i.
- COUNT(in_arc_i), AVERAGE(in_arc_i), MIN(in_arc_i), and MAX(in_arc_i) also refer to attributes exported by sets preceding S; the node s must apply the given function to all sweep tokens arriving along in_arc_i.
- a constant value c; this is supplied as part of the restriction token itself.

The above steps, i.e. the evaluation of ρ and ε and the forwarding of sweep tokens, are repeated by all nodes along the request tree until the root set is reached. Each node of that set, in addition to determining the values of the corresponding expressions ρ and ε, performs the operation O specified by the request.

In the case of *OUTPUT*, all data from the selected nodes is sent to the corresponding master node which, at the model level, may be viewed as performing the necessary processing such as sorting, or computing of aggregate values. As will be discussed in Section 3.2, the actual implementation permits many processing elements to be involved in each of these operations.

The *UPDATE* operation is performed by each selected node of the root set. Each of these simply applies the operations specified as the parameters of the *UPDATE* operation to its own attribute values, thus modifying itself.

The *INSERT_NODE* operation is more complex in that is requires the cooperation of several nodes. First, the new node is created by and connected to the master node of the root set S. As a next step, an arc between the newly created node and each selected node of a set S', connected to S via its *out_arc*, must be established. This is accomplished according to a protocol followed by the master nodes of the sets S and S'; these two nodes exchange the necessary information to establish the arcs.

The *INSERT_ARC* operation also requires the cooperation of selected nodes of the root set S and some other set S', connected to S via its *out_arc*. The necessary information to be exchanged among each two nodes in order to establish an arc is transferred via the corresponding master nodes of the sets S and S' as in the case of inserting a node.

The *DELETE_NODE* and *DELETE_ARC* operations, on the other hand, are quite simple. In the first case, each node selected for deletion removes all arcs connecting it to other nodes. It then informs its respective master node that it wishes to be deleted. In the second case, each selected node removes all those arcs along which tokens were received from the previous set.

2.5 Concurrent Processing and Data Integrity

One of the main requirement of the model was to permit concurrent execution of requests. The model, as described so far, is completely asynchronous in that each node performs an operation whenever it receives a token. There are two main problems to be solved:

- Each node will, in general, receive one restriction token and zero or more sweep tokens. Since communication is asynchronous, tokens will be arriving with arbitrary delays, thus each node must be able to distinguish tokens belonging to different requests and combine these accordingly.
- In the case of requests involving modifications (update, insert, or delete) of any part of the database, data integrity must be preserved in that requests must be serializable. That is, concurrent execution of two or more requests must always yield the same result as if they had been executed in sequence. We employ the following scheme to solve both problems:

Each token carries, in addition to its content as described in Section 2.4, a unique identifier called ***activity name***. For restriction tokens the activity name has the form

$$[req\_id, r/w, (i, j)]$$

while sweep tokens carry only the first component

$$[req\_id]$$

The meaning of the individual components is as follows:

- *req_id* is a unique *request identifier* generated for each new request submitted to the system. This permits each node to distinguish tokens belonging to the same request.

- r/w is a Boolean *flag* which specifies whether the request is a retrieval or a modification of the database.

- (i, j) is a pair of integers generated as follows:
  For each set S_i represented in the database the system maintains two counters R_i and W_i. When a request is submitted, the system performs the following operations:

  ◇ For each bead B_{Sk} constituting the request, the counter R_k, corresponding to the set S_k, is incremented by one.

  ◇ If the request is a modification operation, the corresponding counter W_k is incremented as well.

  ◇ The component (i, j) of the restriction token to be injected into the master node of the set S_k is then composed of the current values of the counters R_k and W_k. (The two integers i and j are similar in nature to the 'use' bit and the 'dirty' bit, used in paged memory systems.)

To maintain integrity of the data, each node of the database graph is then required to obey the following rules:

- A modification request with the activity name $[req\_id, w, (i, j)]$ may be processed only when all requests with activity names $[req\_id, r/w, (i', j')]$, where $i' < i$, have been completed.

- A retrieval request with the activity name $[req\_id, r, (i, j)]$ may be processed only when all requests with the activity name $[req\_id, r/w, (i', j')]$, where $i' < j$, have been completed.

The first rule states that a ***modification*** request must await the completion of ***all*** previous requests, regardless of their type, while the second rule states that ***retrieval*** requests have to await the completion of only the ***last modification*** request. The latter implies that retrieval requests between any two consecutive modifications may be interleaved arbitrarily. Furthermore, since the sequencing is enforced at the node level, a given node does not have to wait for other nodes to complete their respective operation.

3. Implementation of the Model

In the previous section, we have presented a rather abstract model of data processing in which all operators are injected into the dataflow database graph and propagate asynchronously by being replicated and forwarded by individual nodes. This section presents one possible implementation of such a model.

3.1 Economics of Disk Storage

Assume that we wish to implement a database comprising on the order of 10^{10} bytes of data. Table 1 below contrasts the use of large disk units, such as the IBM 3380, with medium size units, such as a high capacity 5 1/4 inch Winchester drive. The line labeled 'no. of units for 10^{10} B' indicates that 4 of the large units versus 133 of the smaller ones would be required. Assuming that each of the large units has 4 parallel actuators, the total number of accesses per millisecond is $16/24 = 0.66$; in the case of the smaller disks this number is $133/43 = 3.09$. Thus, while the cost per byte of storage increases only by approximately 10%, the maximum I/O bandwidth increases by a factor of 5.

	large	medium
cost/unit	\$70K	\$2.2K
capacity	$2.5 * 10^9$	$7.5 * 10^7$
cost/byte	.0026¢	.0029¢
access time	24 msec	43 msec
no. of actuators	4	1
no. of units for 10^{10}B	4	133
no. accesses/msec	16/24 = 0.66	133/43 = 3.09

As will be shown in Section 4, the AG-model is capable of exploiting several hundreds of disk units. Thus to increase the database capacity from 10^{10} to 10^{11}, 40 of the large disk units could be used, resulting in a maximum total bandwidth of 6.6 random accesses per millisecond.

3.2 The Simulated Architecture

Based on the above observations we have implemented the AG-Model on a simulated computer architecture with the following characteristics:

A collection of n processing elements (PEs) are arranged into a k-dimensional array. Both the number of PEs as well as the number of dimensions were varied during the simulation runs to determine their effect. To simplify subsequent discussion, we shall assume k to be 2, i.e., the architecture is a square array, where each PE is connected to its four nearest neighbors, as shown in Figure 1(b). Each PE is in control of a separate disk unit and is equipped with local primary memory, used for program storage, token buffers, and disk cache. The amount of memory available for disk cache is assumed to be very small (about 3%), relative to the disk space allotted for node storage. Thus disk performance is the major factor determining the access time to nodes.

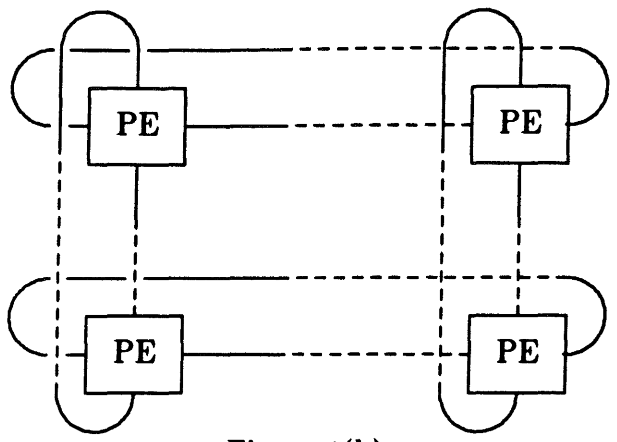

Figure 1(b)

For reasons of reliability, the space on each disk is divided in half; one half contains data belonging to the owner PE while the other contains a copy of the data belonging to its *buddy* (left hand neighbor). Thus, in the case of a PE/disk failure, the neighboring PE may resume the work of the failed component.

To communicate with users, the database machine has one or more I/O processors attached to selected PEs. The optimal number of such processors and their connection to different PEs is independent of the model. Conceptually, one I/O processor is sufficient, which is the assumption made in this section.

The database graph is mapped onto the collection of individual memories as follows. Each node n of the graph is uniquely identified by a pair (S, k_n), where S is the set to which the node belongs and k_n is its key value within S, as discussed in Section 2.1. Each of the PEs is identified by a unique number from 1 to p, where p is the number of existing PEs. To map a node n onto a PE, a system-wide hashing function f is applied to the corresponding pair (S, k), which always yields a number between 1 and p. This number, $f(S, k)$, is then used as the PE number whose disk unit will hold the node n.

Recall that, according to the model, each node must be an active entity. In the implementation, each PE may be viewed as the incarnation of all nodes mapped onto that PE; for all of these nodes the PE must receive, process, and emit tokens traveling along the graph arcs.

An arc between two nodes n1 and n2, belonging to the sets $S1$ and $S2$, respectively, is represented by recording the corresponding key value $(S1, k_{n1})$ with the node n2 and the key value $(S2, k_{n2})$ with the node n1.

Propagation of Tokens

As mentioned above, arcs of the graph are represented as lists of set and key value pairs, kept with each node. According to the model, each node must be able to send tokens along any of its arcs to other nodes. Since for any given query the number of tokens to be exchanged among nodes could be very large, we must try to find ways of minimizing the number of tokens actually transmitted. As described in Section 2.4, there are two types of tokens – restriction and sweep tokens. Let us first consider a scheme to reduce the number of actually transmitted sweep tokens.

Each sweep token carries the value $\bar{\rho}$ (True or False), which indicates whether the node emitting that sweep token has been selected by its corresponding restriction ρ. Typically, the selection rate is quite small (less than 10%) and hence the number of sweep tokens carrying the value False will exceed those with the value True by an order of magnitude. To reduce the token traffic, only True tokens will actually be transmitted while the absence of a token on a particular arcs will be interpreted as the arrival of a False token.

The main problem with this scheme lies in the asynchronous nature of the model. Since tokens may be arriving with arbitrary delays, each node must be able to determine whether the absence of a True tokens is to be interpreted as the arrival of a False token or whether a True token is still in transit. To solve this problem, we implement the following protocol:

Assume that the selected nodes of a set $S1$ are to transmit sweep tokens to nodes of another set $S2$ along existing arcs. Each node of the set $S1$ will report to its master node the number of tokens actually sent. The master node accumulates the total number of tokens sent and reports it to the master node of the receiving set $S2$. In the meantime, nodes of the set $S2$ report the numbers of tokens received from nodes of the set $S1$ to their master. When the total number of tokens received equals the total number of tokens sent, the master node of the set $S2$ notifies each of its member nodes that no more tokens will be arriving. At this point, each arc from which no token has been received may be interpreted as having delivered a token with the value False.

Assuming that sweep tokens will be transmitted according to the scheme described above, we can distinguish the following three situations; for each of these, efficient token propagation mechanisms must be provided:

1. A node is sending a copy of a token to all elements of a given set. Typically, this will be the case when a master node is replicating a restriction token to all its member nodes, or when it is informing its member nodes that no more sweep tokens are in transit.

2. A node is receiving a large number of tokens, each arriving from a different node. This will occur when the master node is waiting for all its member nodes to report the number of sweep tokens sent or received, or when the results of a query are being collected.

3. A node is sending a copy of a token to a selected subset of nodes, and, conversely, a node is awaiting a number of tokens to arrive from different nodes. This situation represents the flow of True sweep tokens exchanged among individual member nodes.

For each of these three situations we will employ a different scheme for transporting tokens to minimize the communication overhead. In the following discussion we will refer to a given PE by a pair of coordinates (i, j), where i and j designate the corresponding row and column within the two-dimensional array of PEs:

1. To replicate a token to all elements of a set S a scheme called *flooding* is used which replicates the token as follows. From the PE holding the sending node the token is replicated first in only one direction, say horizontally, along the coordinate i. Each PE in the row i then replicates the token vertically along the corresponding column j of PEs thus 'flooding' the entire PE array. Each PE then treats the received token as if a separate copy had arrived for each node of the set S. Note that the number of transmissions is proportional to the number of PEs rather than the number of nodes comprising the set S.

2. To return a large number of tokens to a single node a scheme called *draining* is used, which accomplishes the reverse function of flooding. The last row of PEs receiving the flooding tokens will return the results along the same columns j until the originally first row i is reached in which the tokens are propagated horizontally toward the PE containing the receiving node. Thus the tokens being drained follow the reverse paths of the flooding tokens. Since each PE combines its own tokens with those received from its immediate neighbors, the total number of transmissions is again proportional to the number of PEs.

All aggregate operations as well as sorting of values are performed during the draining process. Consider for example the situation where each member node of a given set must report to its master node the number of tokens sent to member nodes of some other set as described above. During the draining process, each PE will count the number of tokens emitted by its own nodes and add to it the number of tokens reported to it by its immediate neighbors before forwarding the result to the next PE.

Other operations such as calculating the values of the functions COUNT, AVERAGE, MAX, MIN, or the sorting of values can be performed in an analogous way in that each PE performs the necessary operations locally on all nodes mapped onto that PE and merges the results with those received from its neighbors.

3. Sending and receiving sweep tokens along arcs between selected subsets of nodes must be performed on an individual basis. The PE holding the sending node determines the i and j coordinates of the PE holding the receiving node. Based on that information and its own position within the array it determines which of its four neighbors has the shortest geometric distance to the destination PE, and sends the token into that direction. This operation is repeated by each PE receiving the token until the final destination is reached.

The initial injection of restriction tokens into master nodes is implemented in an analogous way. The token is injected into one of the PEs connected to an I/O processor. Based on the token's destination, the injection PE determines the i and j coordinates of the PE holding the destination node and send the token into the appropriate direction.

4. Simulation Results

To test architectural ideas and to evaluate the performance of the the proposed database system, we have implemented the AG-Model on a simulated architecture. The complete software package, henceforth referred to as the *AG-Simulator*, consists of approximately 8000 lines of SIMULA code executing on a VAX-11/780.[2] The following sections describe the results of the simulation experiments.

[2] VAX is a registered trademark of Digital Equipment Corporation

4.1 Performance Evaluation Methodology

We have followed the methodology for evaluating database systems proposed by Boral and DeWitt /BoDeW84/. The basic structure of the synthetic database as well as the proposed four query types and the query mix were adopted from this paper:

- Query type I is a direct access of a single node using a key. In the implementation of the AG-Model, hashing is used instead of indexing.
- Query type II selects 1% of a given set.
- Query type III selects 10% of one set and joins the resulting subset with another set. In our case, no indices are used; rather, a join is functionally equivalent to sending sweep tokens from selected elements of one set to another.
- Query type IV is the same as query type II except the selection rate is 10%. In addition, this query performs some aggregate function; we have chosen to perform a sorting on the final results.
- Finally our query mix is the one suggested in /BoDeW84/: 70% of type I, 10% of type II, 10% of type III, and 10% of type IV queries.

4.2 Parametric Variation Experiments

We have carried out the following four series of experiments to test various aspects of the proposed system.

4.2.1 Architecture Topology Variation

As mentioned in Section 3.2, the architecture assumed for the proposed system is a k-dimensional array of processing elements. The first set of experiments was intended to investigate the effects of varying the number of dimensions, k. The primary objective was to confirm our intuitive assumption that, once a 'reasonable' number of physical links are established among PEs, adding new connections has little impact on improvement in performance.

We can distinguish the following three major phases of each query: (1) flooding of the array, which sends restriction tokens to all PEs, (2) exchange of sweep tokens among nodes of the involved sets, and (3) draining of the array, which collects the results. Let us consider these in turn:

1. *Flooding*: Figure 2(a) shows the correlation between the number of dimensions and the flooding time: the improvement is dramatic when increasing the number of dimensions from 1 to 2, as represented by the distance between the solid and the dashed curves; it becomes less important when a third dimension is added. (Note that the time to flood the array is completely independent of the database size, the query type, or the disk performance.)

2. *Propagation of sweep tokens*: Each sweep token is propagated along the shortest geometric distance from the sending to the receiving node. Since the distribution of nodes over PEs is random, each sweep token will travel a distance corresponding to the average path length within the array. This distance is plotted in Figure 2(b) for the three different dimensions. The resulting curves are similar to those for the array flood times (Figure 2(a)): the improvement between 1 and 2 dimensions is dramatic but diminishes when a third dimension is added.

3. *Draining*: The time to drain the array obviously depends on the number of results to be returned. If this number is very small, the time to drain the array is essential the same as the array flood time (Figure 2(a)). If the number of results is large, the time to complete the query will be limited by the speed of the IO device designated to (sequentially) output all results. (This, of course, does not prevent

other queries to proceed in parallel, thus utilizing the available resources.)

In summary, we observe that in all three cases the effect of increasing the connectivity of the PE array diminishes rapidly. Considering the fact that each new dimension requires two communication links to be added to every PE, the improvement from two to three dimensions appears already quite marginal. Based on this observation, we have restricted all subsequent experiments to only two-dimensional arrays of PEs.

4.2.2 Problem Size Variation

The next set of experiments is intended to study the effects of varying the amount of work handled by each PE, on the request processing time. For that purpose, we consider an array of 9 PEs (3 $\times$ 3) and vary the set size from 10 to 1024. Figure 3(a) shows the mean processing time for three different types of queries.[3] While a very slow increase is observed for queries of Type I, it becomes almost linear for queries of Type II and III; that is, the mean processing time for the latter types is directly proportional to the problem size.

At a first glance, this result does not seem to represent any major breakthrough in performance, since a conventional database machine displays a similar degradation in response time. We must, however, consider the amount of resources actually utilized to process each query. This is shown in Figures 3(b) and 3(c) for secondary memory and for the PEs, respectively. As expected, for queries of Type I, most (90%) of the available I/O bandwidth as well as the processing time is unused. For queries of Type II and III, disk utilization rises to a maximum of approximately 75% and then decreases slightly; this is due to a decrease in average seek time as

[3] These experiment were carried out with a multiprogramming level of two, i.e., two queries were executing simultaneously.

the density of nodes on the disks increases. The available processing power is even less utilized; with 100 nodes per PE, over 60% is still unused.

In summary, we observe that, for queries of Type I, the mean processing time remains nearly constant; that is, all disks except one and almost all PEs are unused. For other query types the mean processing time increases linearly with the problem size, however, even in the small array of only 9 PEs, much of the available I/O bandwidth (> 35%) as well as the computing potential (> 60%) is still unused. (This unused capacity may be exploited by increasing the number of simultaneous queries, as will be discussed in Section 4.2.4.)

4.2.3 Array Size Variation

The purpose of this series of experiments is to investigate the effects of increasing the array size, i.e., the number of PEs. Ideally, the mean processing time for a query should increase only slightly (due to longer communication paths within the array), while the unused I/O bandwidth and the PE idle time should increase in proportion with the array size. To confirm this assumption, we have varied the array size from 4 to 1024 PEs, while keeping the set size and the queries constant. The resulting mean processing time for a query is plotted in Figure 4(a). We observe that by increasing the number of PEs from 9, (which was the size assumed in the previous experiment), to 1024, i.e., by two orders of magnitude, the mean processing time for a query does not show any dramatic changes; it decreases first as more disk units and PEs are added and then rises again due to longer communication paths within the array. (Note that a logarithmic scale is used in Figure 4(a).)

While the above changes in query processing time are rather insignificant, the increase in unused I/O bandwidth and PE time is dramatic, as shown in Figure 4(b) and 4(c), respectively; with 300 PEs, both values are nearly zero.

4.2.4 Multiprogramming Level Variation

The previous experiments have shown that increasing the number of PEs does not have any significant adverse effect on the mean query processing time. Our objective now is to show that the unused I/O bandwidth and the computational power may usefully be exploited for simultaneous processing of other queries. For that purpose, we return to the original array of nine PEs, and vary the number of simultaneous queries (selected from the mix suggested in /BoDeW84/) from 1 to 16. Figure 5 shows that, even in the case of nine PEs, where resource utilization is relatively high (Figures 3(b) and 3(c)), the system throughput increases from 1.3 to approximately 3 queries per second.

5. Conclusions

The objective of this paper was to demonstrate that the use of hundreds of processors in a database machine is feasible, provided the I/O bandwidth of the secondary storage medium is increased accordingly, as pointed out in /AgDeW84/. To accomplish the latter, we proposed to replace each large disk with a number of smaller units, each connected to an independent processor. By employing a database model (the AG-Model) suitable to parallel processing, we have shown that the potential I/O bandwidth and the associated computational power of the PE array may usefully be exploited.

We have implemented the AG-Model on a simulated architecture. Due to limitations imposed by the simulator (a typical simulation run producing one data point for the plotted curves consumed between 2 and 10 hours of VAX-11/780 CPU time) we were forced to accept a number of restrictions. In particular, (1) the size of the array had to be kept very small; for example, to place any significant load on individual PEs, only 9 were used for the problem size variation experiments (Figure 3(a)-(c)); (2) the distribution of data nodes over the disks was assumed to be random; a better memory management scheme would significantly improve the utilization of the available I/O bandwidth; (3) in an actual implementation, queries referring to the same sets could reduce the number of disk accesses significantly by using a cache, as discussed in /BeDeW84/; we did not exploit this potential of data sharing in the simulator.

Despite the above adverse assumptions, the obtained results are quite encouraging — the proposed system is capable of utilizing the available I/O bandwidth and the computational power of a significant number of asynchronously operating processing elements.

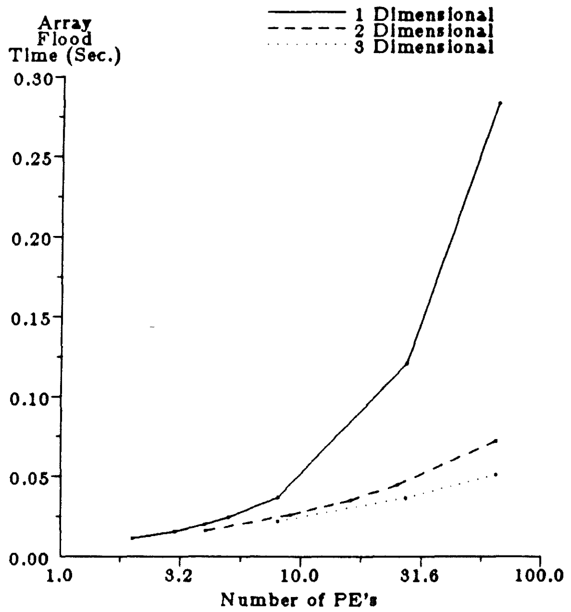

Fig. 2(a). Dimensionality Effect on Flood Time

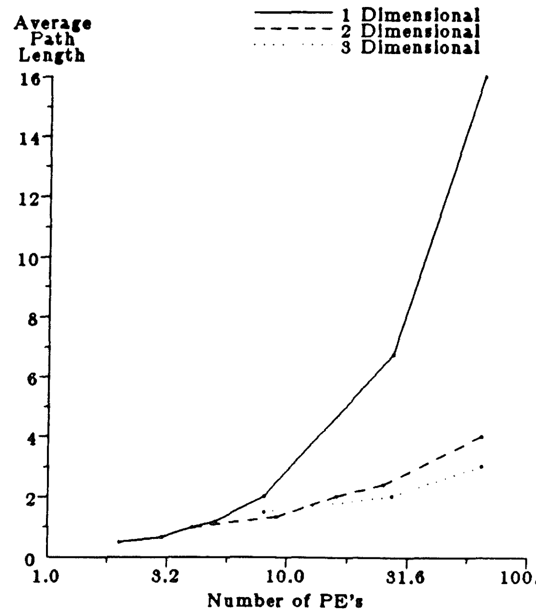

Fig. 2(b). Dimensionality Effect on Path Length

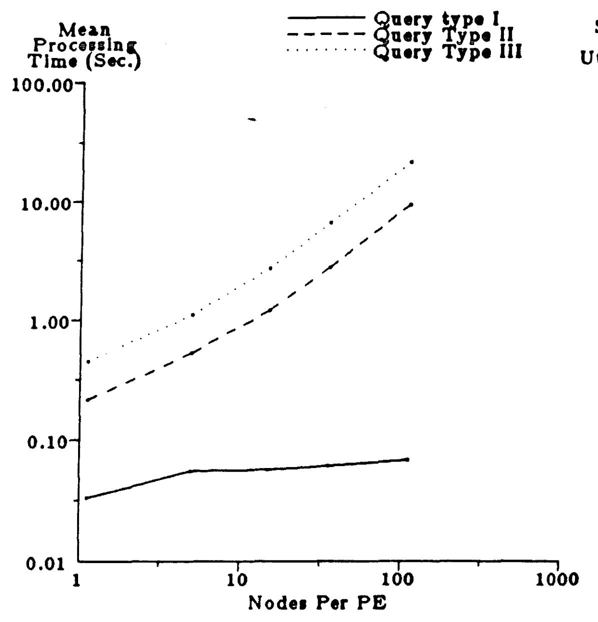

Fig. 3(a). Problem Size Effect on Processing Time

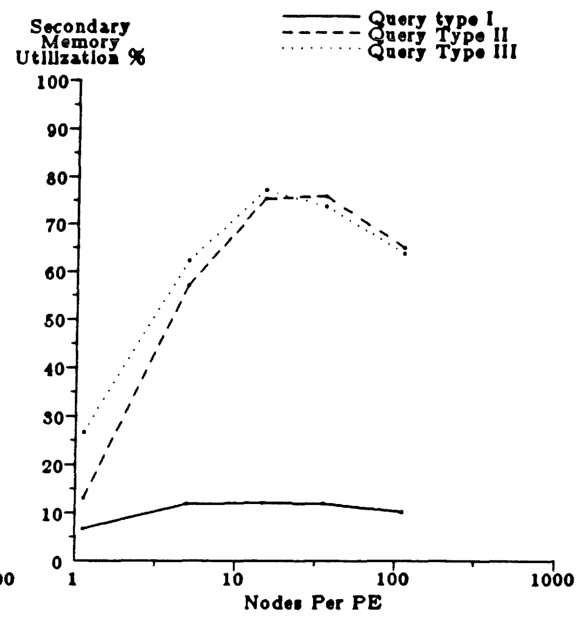

Fig. 3(b). Problem Size Effect on Disk Utilization

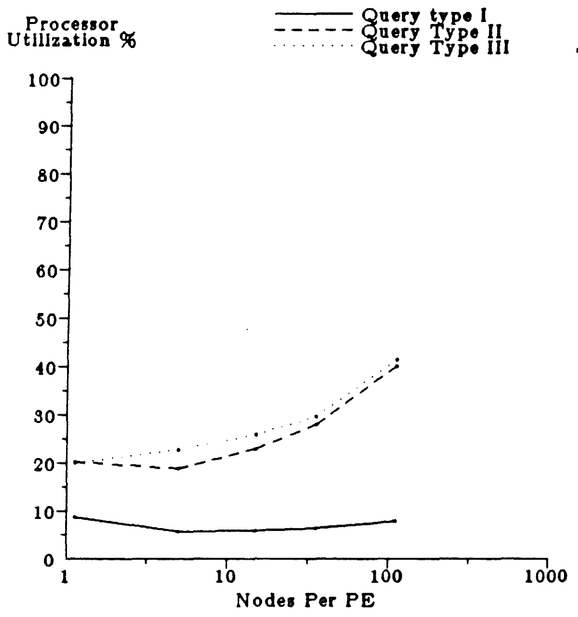

Fig. 3(c). Problem Size Effect on Processor Utilization

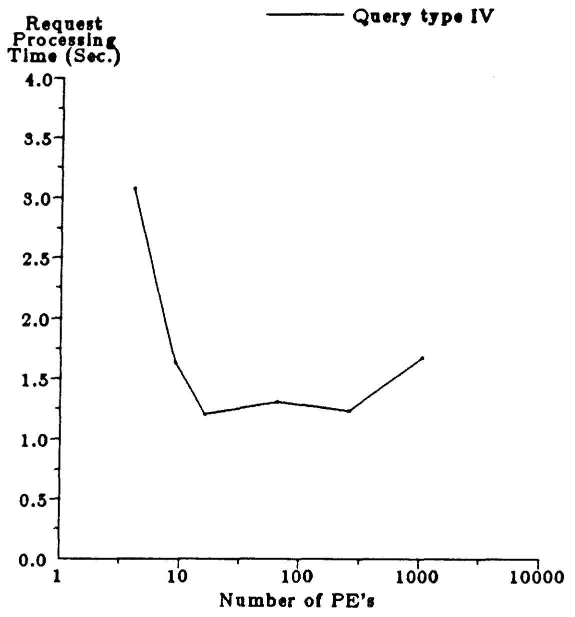

Fig. 4(a). Array Size Effect on Processing Time

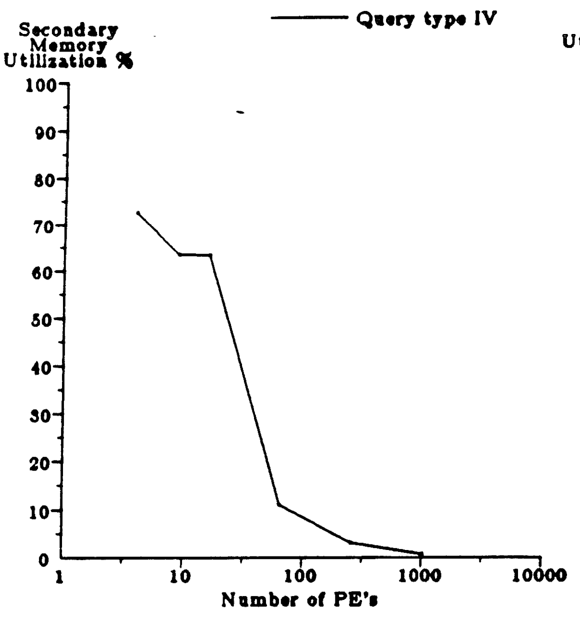

Fig. 4(b). Array Size Effect on Disk Utilization

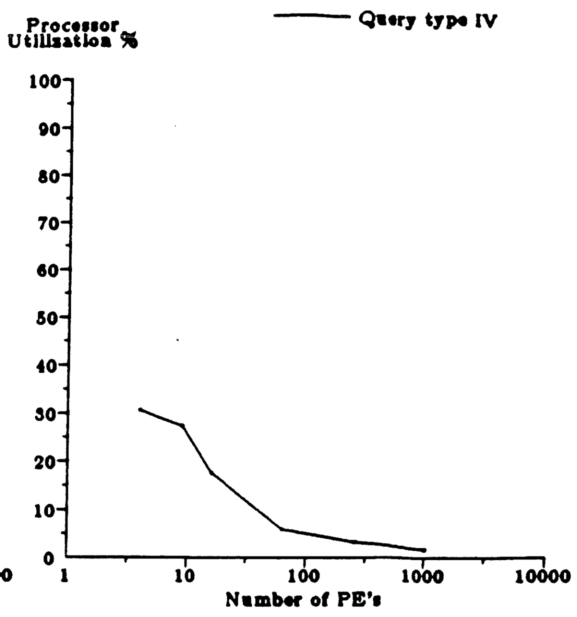

Fig. 4(c). Array Size Effect on Processor Utilization

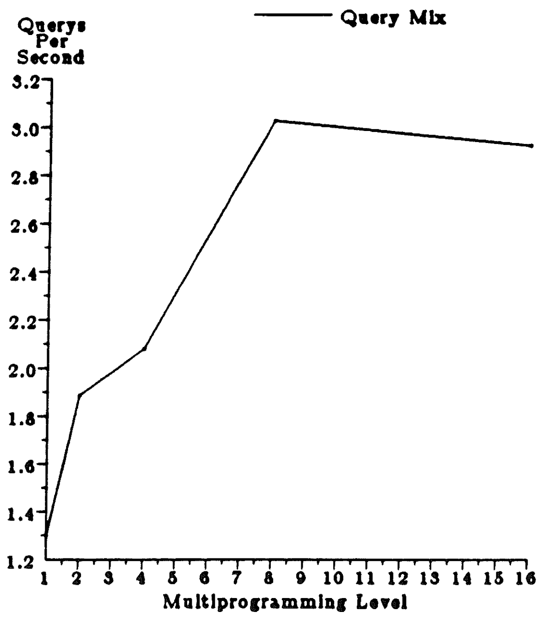

Fig. 5. System Loading Effect on Throughput

References

/AgDeW84/ Agrawal, R. and DeWitt, D. J.: "Whither Hundreds of Processors in a

Database Machine?," Int'l Workshop on High-Level Architecture, Los Angeles, California, 1984.

/BoDeW84/ Boral, H. and DeWitt, D. J.: "A Methodology for Database Performance Evaluation," ACM SIGMOD, Proc. of Annual Meeting, Vol.14, No.2, 1984.

/Che76/ Chen, P.: "The Entity - Relationship Model: Toward a Unified View of Data," ACM TODS, 1,1, March 1976

/COM82/ COMPUTER, Special Issue on Dataflow Systems, 15,2, Feb. 1982

/Har84/ Hartmann, R.: "The Active-Graph Database Machine," PhD Thesis (to be completed Dec. 1984), Dept of ICS, University of California, Irvine, 92717

/Sch78/ Schuster, S. A., at. al.: "RAP.2 – An Associative Processor for Data Bases," Proc. Fifth Annual IEEE Symp. for Computer Architecture, 1978

/Sho78/ Shoshani, A.: "CABLE: A Language Based on the Entity-Relationship Model," Technical Report, Lawrence Berkeley Lab., 1978

/TBH82/ Treleaven, P. C., Brownbridge, T. R.,. Hopkins, R. C.: "Data-Driven and Demand-Driven Computer Architecture," ACM Computing Surveys, 14,1, March 1982