OOHYDRO: A Heterogeneous Distributed Database System for Object Base and Database Interoperability

W. Perrizo[1], J. Gustafson, H. Hakimzadeh, R. Haraty, P. Kodali, B. Panda
and K. Scott
Computer Science, Box 5075,
North Dakota State University,
Fargo, ND 58105

ABSTRACT

At North Dakota State University and IBM Rochester, MN, a development project is underway for a Heterogeneous Distributed Database System which includes object oriented data stores. The system is called the Heterogeneously Distributed Request Orderer (HYDRO). We view the object model as an extension of the relational model. When HYDRO connects autonomous local database systems as well as object oriented data stores into one unified system, we refer to it as OOHYDRO. In this paper we show how global serializability and atomic commitment can be attained in this setting. Each local system is assumed to be off-the-shelf and binary-licensed. Global serializability is achieved using a set of system objects based on the ROLLs (Request Order Linked Lists) ([PER89], [PER91a]). ROLL is a Serialization Graph Test concurrency control methodology ([BER87]) which provides freedom from idle-wait, deadlock, livelock and restart. Atomic commitment is provided through a variation of the Two-Phase Commit protocol.

1. INTRODUCTION

Ideally, a heterogeneous distributed database system should provide transparent, efficient , correct, and durable access to standard relational data and to non-standard object-oriented data as well. In this paper we describe the Heterogeneously Distributed Request Ordering (HYDRO) system, which is being developed for that purpose at North Dakota State University with the support of IBM ABS in Rochester, MN. HYDRO is intended for distributed database management in a networked heterogeneous environment combining a variety of local database systems and object stores, including the IBM AS/400 with its native database system and its fully object oriented Machine Interface. For transaction correctness, global serializability and atomic commitment are used. Local autonomy is provided at each site. For all purposes local autonomy means only that each local DBMS is an off-the-shelf, binary-licensed product. These local systems may be standard transaction processing DBMS's, decision support systems, knowledge bases, or object management systems. We view the object model as an extension of the relational model.

[1]Partially supported by IBM AS/400 Research Grants 3341-IN-R-09233 and 632207.

At each site, OOHYDRO has a local server module (called a LHYDRO) which is customized to take advantage of the local interface provided by the local system at that site. At some sites, complete transaction management will be required. Local server options are provided for sites where a visible PREPARE statement is provided and at sites where PREPARE is not available.

There are many models and points of view for developing heterogeneous distributed database systems. Our point of view is that the management of a large organization, with many independent data management systems, mandates organization-wide data integration. Strong incentives are given to individual departments to apply for membership in the organization-wide database federation. To do this, local data management units must fit their data (or that portion of it to be federated) into the federated schema and provide access to it. The LHYDRO objects provide the "connection" to the federation. Our model takes the point of view that local autonomy is a temporary system liability and not a virtue. The local sites export their local schema for the data they wish to make available to the federation. The site at which the global concurrency control software resides (GHYDRO software), hereafter called the "global site", maintains the federated schemata as a set of translation tables. These tables associate a surrogate value (object identifier or oid) with each logical item or object. Local sites submit global requests to the global site in surrogated form (bit vectors specifying requirements). The global site fully reduces the request and submits sub-transactions to the local sites as surrogate structures which are used to optimize request execution. This process is explained in section 5.

In section 2, we describe attribute level transaction management using the Request Order Linked List object (ROLL) [PER91a]. Section 3 provides several examples of object classes and describe how they are accommodated in our system. Section 4 describes how we achieve atomic commitment. In section 5, we discuss two optimization techniques by which relation oriented queries can be optimized at the surrogate level. This optimization is very important due to the fact that global serializability requirements can cause unusually long execution durations for standard relational data accesses. In section 6, we conclude and discuss what is to come in future.

2. REVIEW OF ROLL CONCURRENCY CONTROL FOR ATTRIBUTE LEVEL ACCESS

To achieve serializable concurrency control, both waiting and restarting are used. Neither is ever desirable. To avoid restarting and minimize waiting we use the ROLL concurrency control method. The ROLL method is based on serialization graph testing (SGT) [BER87]. The SGT approach is optimal in the sense that no serializable execution is rejected. SGT attains serializable executions by ensuring the serialization graph always remains acyclic. We give an overview of the ROLL method here and refer the reader to other published accounts for more detail. ([PER91a], [PER91b]).

In the ROLL object model, data item requests are indicated using a bit-vector, the RV, in which each bit position corresponds to a different logical data item in the system (1 means request and 0 means no request), n bits are assigned to each data item if n access modes are offered (i.e. 2 bits if read and write access are to be distinguished). The ROLL object is a linked list of these bit vectors and is the only data structure accessed by transactions for concurrency control. Three operations or methods are available to transactions, namely, POST, CHECK and RELEASE. The main advantage of ROLL is that transaction managers can act entirely in parallel independent of any bottleneck scheduler module by invoking POST, CHECK and RELEASE operations on ROLL objects.

POST is an atomic enqueue operation which establishes the serialization partial order for issuing transactions. Having created a RV, a transaction POSTs its RV to the ROLL. This POST operation is the only operation that must be atomic.

The CHECK operation allows a transaction to determine availability for its entire request set in one operation. CHECK operations can be done in parallel by any number of transactions. A transaction's CHECK returns the accumulated logical OR of all RVs POSTed ahead of its own. The accumulated OR represents an "access filter" for the transaction (a one means the item corresponding to that position is as yet unavailable and a zero means it is available). CHECK can be repeated at any time to determine which needed items have become available since the last CHECK operation. Thus, transactions are never forced to wait idly for a response from the system scheduler.

To RELEASE a data item once finished with it, the transaction simply flips the corresponding bit from a one to a zero (exclusive OR with a 1). The next transaction POSTed for that item will then find it available upon performing the CHECK operation. If strictness is used to provide recoverability, all bits are RELEASEd together at the commit point.

Some care must be taken to keep the ROLL relatively short (free from vectors that correspond to terminated transactions). Transactions Managers could be made responsible for removing their own RVs, however we favor an approach in which a background system process (which we will refer to as CLEAN) REMOVEs zeroed elements by periodically searching for the last (top most) element which is not yet all zeros and changing its up pointer to null. For a more detailed description of these operations the reader is referred to [PER91a].

It is shown in [PER91a] that ROLL concurrency control is correct, (ie. produces only conflict serializable executions) and is deadlock, livelock and restart free. The main cause of delay in waiting policies, such as Two-Phase Locking (2PL), is the single system scheduler. The ROLL method does not use a scheduler, instead, transaction managers monitor the availability of items in parallel using the CHECK operation.

As stated previously, we assume the HDDBMSs is being developed bottom-up to accommodate currently existing local DBMSs (LDBMSs). Thus, we will assume each LDBMS is an off-the-shelf, binary-licensed commercial product which supports local serializability, recoverability and atomic commitment. Our model includes local autonomy as defined by Elmagarmid [ELM88]. Unlike Elmagarmid however, we route local transactions through the local LHYDRO module prior to being submitted to the LDBMS.

2.1 Attribute Level ROLL or ALROLL

An object can be very large, complex, and may have many attributes and methods. It would be a tedious task to try to apply the traditional concurrency control protocols to manage requests for complex objects, due to the fact that objects are stratified. Getting exclusive access at the object level might result in a bottleneck and therefore idle-waiting. For this reason we have developed a method for finer granularity access called Attribute Level ROLL (ALROLL). In ALROLL, there are two types of ROLL objects: a Meta ROLL (MROLL) and Object ROLLs (OROLLs).

The sole purpose of the MROLL is to allow as much parallelism in the atomic POST process as possible. Without a MROLL, POSTing would be governed by a semaphore and therefore be entirely serial in nature. We find it useful to view the MROLL as a multiple-queued-semaphore which provides parallelism in POST operations. Each bit position in the MROLL corresponds to an object. A transaction creates a "Meta Vector" or MV with a 1-bit set for each object to which access is needed (regardless of the mode of access needed). The MV is then enqueued to the MROLL (using an MPOST operation). A CHECK operation on the MROLL (MCHECK), determines which of the needed objects can be POSTed to (OPOST). For those objects, an Object Request Vector (ORV) is POSTed (in its Object ROLL or OROLL). An ORV has a 1 bit set for each needed attribute and access mode.

We pause here to give examples at this point. Further detail on these and other examples is provided in the next section. Consider a stack object with methods push, pop and top. Stacks are treated as atomic data items for concurrency control purposes. This is reasonable since access to a stack is seldom made to individual stack elements, except the head (push, pop and top). An Object Request Vector for such a stack would have three bits, one for each method. An Object CHECK (OCHECK) operation is done to determine accessibility of the stack. The OCHECK operation returns an Object Access Vector (OAV) of the same form as the ORV. Knowing that push and pop combinations do not commute nor do they commute with top, and that two top operations do commute, analysis of the OAV and the transactions ORV determines the availability.

A second example is a that of a relational tuple as an object. We assume two methods, read and write, and n attributes. The ORV would have 2n bits position, one for each method-attribute combination. An OCHECK operation provides an OAV which specifies which attributes are available and in which mode of access they are available. Again, comparing the OAV and ORV, the Transaction Manager can completely determine availability. Note that much higher throughput can be expected than with standard "tuple granularity" approaches since conflict occurs only if non-commutative operations are requested on the same attribute.

Each OROLL corresponds to one active encapsulated object. Thus there are as many OROLLs as there are objects in current use . An OROLL can be deactivated (and its space released) as soon as it holds only 0-bits.

Figure 1. below illustrates ALROLL.

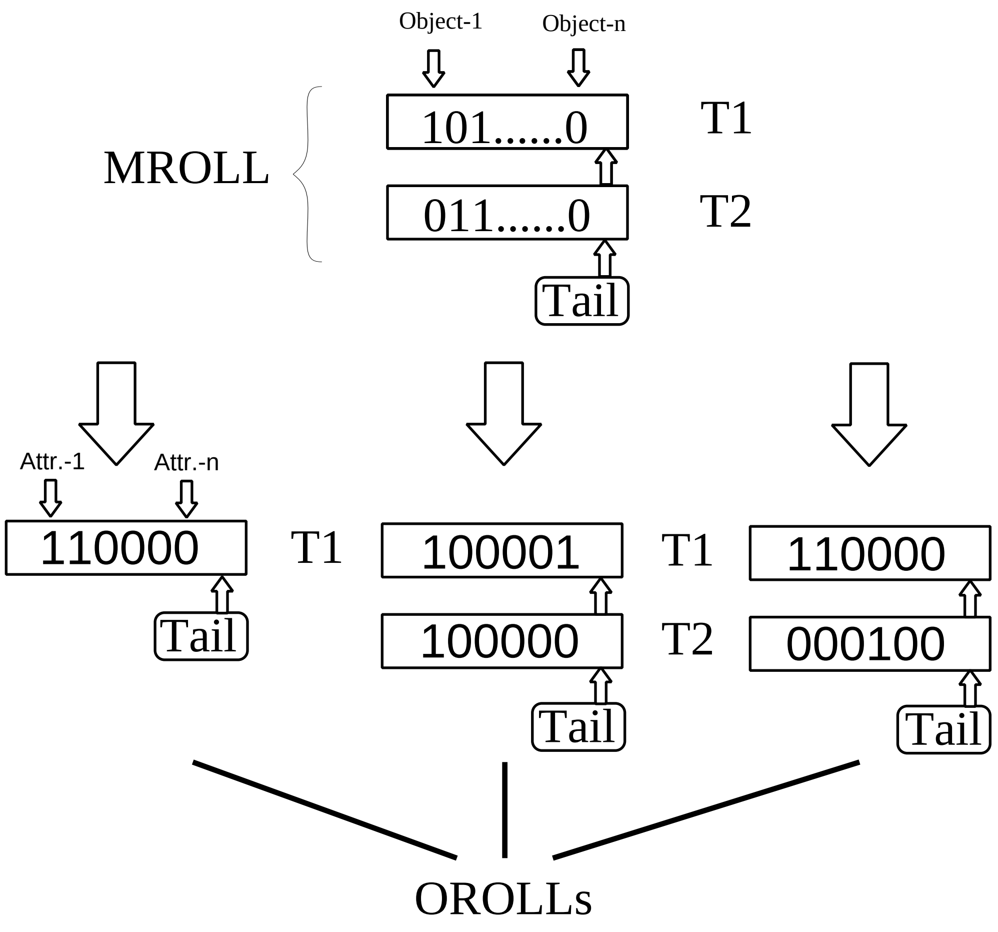

Figure 1. ALROLL

POSTing to the MROLL establishes the serialization partial order of transactions. It ensures that all local OROLL POSTings are serializable across objects. The OROLL will provide for more concurrency by handling requests at the attribute level rather than the object or the class levels and by encapsulating the concurrency control mechanisms for its attributes within the object itself.

We note that the MV vectors might be very wide. The MROLL can be replaced by a hierarchical scheme of MROLLs for each Class and Subclass or the position mapping can be a hash function (grouping objects together in "buckets" so that the hot objects are spread out). In so doing, we reduce the size of the MVs significantly. Also, it is to be noted that the MROLL is only accessed once, at the beginning of the transaction and serves only to insure that a transaction's ORVs get OPOSTed in a serializable order. The brute force way to insure this would be to provide a single "POST semaphore". Then each transaction would seize the semaphore, do all its local OPOSTing, and then releases the semaphore. This would require much more idle waiting in the semaphore queue than our scheme. For long-duration transactions, such as are common in any object oriented database setting, ALROLL allows much higher processing speeds than conventional methods, since the large MROLL is accessed only once, at initiation, the MPOST process is allowed to be done in parallel, and subsequent OCHECK operations are very fast.

Several performance enhancements to this general approach are discussed in other papers ([PER89], [PER91a], [PER91b], [MCC92]) which will dramatically decrease the width of the ORV's in situations where those vectors would be expected to be excessively wide (too wide to process efficiently). We also note that the internal structure of the encapsulated ROLL object need be a linked list of RV's, but alternatively a linked list of access vectors (AVs), so that OCHECK is a very fast copy operator. We call this approach AVROLL. In AVROLL, a background process updates a list of AV's much more efficiently. Preliminary analysis shows this approach to be very advantageous.

3. OBJECT CLASS EXAMPLES

In this section we will examine a number of classes of abstract data types such as queues, stacks, tuples, sets and other user defined types such as those used in the CAD environment [ATK90]. We will show how the synergy between ROLL concurrency control and operation commutativity can be used to achieve more concurrency.

We will describe the class, its methods, commutativity of its methods and the data granularity level used for that class. Following that analysis, the form of the Object Request Vector (ORV) is specified for the class. In each case, one can see how throughput can be enhanced by using either finer data granularity or by "finer" operation semantics.

In general, methods or operations fall into 4 categories, creators, mutators, observers and state-changers. We note that mutators for a class are the creators for the subclasses (e.g., creating a tuple corresponds to mutating the relation to which it belongs). Thus, we restrict our attention to mutators, observers (reads) and state-changers (writes).

Queue Object

A queue is an abstract data type consisting of a linked list with a head and tail pointers. The operations on a queue are ENQueue (ENQ), DEQueue (DEQ) and MATERIALIZE (MAT). The commutativity matrix for a standard queue is as follows:

	ENQ	DEQ	MAT
ENQ	n	n	n
DEQ	n	n	n
MAT	n	n	y

With this standard model of a queue, the ORV would have 3 bits, one for each method.

However, it should be noted that if the queue is not empty more concurrency can be realized. If we distinguish between ENQ, DEQ in general and ENQ', DEQ' (which only apply to non-empty queues) the primed methods clearly commute with each other. (Additional method semantics would be possible for the MATERIALIZE as well.) With this enhanced model, the ORV for a queue object would have 5 bits, one for each of the methods, and considerable concurrency would be introduced to the system. If we further extend the queue model to include a MATERIALIZE for each element of the queue, these new MATERIALIZEs will not only commute with each other, but most of them will commute with the (ENQ, DEQ), (ENQ', DEQ') operations. The ORV would have a separate bit for each method (4 + max-queue-length in all).

Note that the object oriented ROLL shifts the concurrency control to the object itself (except for the establishment of global serialization partial order, which is done at the Meta ROLL level).

Stack Object

The stack object is similar to the queue, with the exception of its semantic behavior and its operations. The operations on a stack are PUSH, POP and TOP and the commutativity matrix for a standard stack is as follows:

	PUSH	POP	TOP
PUSH	n	n	n
POP	n	n	n
TOP	n	n	y

In the case of the stack object the ROLL vector will be 3 bits, however only the first 2 bits conflict. Additional method semantics can be introduced for stack objects in a fashion similar to the above discussion for queues.

CAD Object

In a user defined CAD application the operations may be create, delete, edit_text, edit_graphics, show, show_recursive, draw, draw_recursive, attach, detach, copy, etc. The description of each method is as follows:

Create - create an object
Delete - delete the object
Edit_text - edit the text attributes of the object
Edit_graphics - edit the graphics attributes of the object
Show - show the text attributes of the object
Show_recursive - same as show, but recursively (include sub-objects)
Draw - draw the graphics attributes of the object
Draw_recursive - same as draw, but recursively (include sub-objects)
Connect - attach two object together
Disconnect - detach an object from another
Copy - copy the contents of one object to another (get a snap shot of this object.

The commutativity matrix for this object is as follows:

	CRET	DEL	ED_TXT	ED_GRF	SHOW	SHOW_REC	DRAW	DRAW_REC	CONN	DISCON	COPY
CRET	n	n	n	n	n	n	n	n	n	n	n
DEL	n	n	n	n	n	n	n	n	n	n	n
ED_TXT	n	n	n	y	n	n	y	y	n	n	n
ED_GRF	n	n	y	n	y	y	n	n	n	n	n
SHOW	n	n	n	y	y	y	y	y	n	n	y
SHOW_REC	n	n	n	y	y	y	y	y	n	n	y
DRAW	n	n	y	n	y	y	y	y	n	n	y
DRAW_REC	n	n	y	n	y	y	y	y	n	n	y
CONN	n	n	n	n	n	n	n	n	n	n	n
DISCON	n	n	n	n	n	n	n	n	n	n	n
COPY	n	n	n	n	y	y	y	y	n	n	y

Assuming the CAD object has n-atomic components, an ORV would have 11n-bit positions, 1 for each method-component pair.

Tuple Object

In an object oriented database setting, a relation can be defined as a class and a relational tuple can be considered as an object. If there are m methods (read, write, increment, decrement, ...) and n attributes, the ORV would be m*n bits wide, one for each method-attribute combination. The commutativity matrix would be as follows:

	READ	WRITE	INC	DEC
READ	y	n	n	n
WRITE	n	n	n	n
INC	n	n	y	y
DEC	n	n	y	y

Set Object

A relation is a typical set object. The methods are (read, write, insert, and delete). The elements or "atomic data items" of a set object are the individual tuples (1 bit per method). The commutativity matrix looks as follows:

	READ	WRITE	INSERT	DELETE
READ	y	n	n	n
WRITE	n	n	n	n
INSERT	n	n	y	n
DELETE	n	n	n	y

It should be noted that if the set is not empty, INSERT and DELETE methods will commute.

More complex examples such as trees, quadtrees, octrees, MD-tree etc can be analyzed similarly.

The above examples on queues, stacks, CAD objects, tuples, and set objects clearly demonstrate the subtle interaction between the ROLL concurrency control and operations commutativity. Recall that in the ROLL concurrency control model when an operation is invoked on behalf of a transaction, the operation creates an Object ROLL (OROLL) vector placing a 1 bit for each attribute that it needs and a 0 bit for those that it does not. This vector is then posted in the OROLL structure. However, by consulting the semantic information in the commutativity matrix the CHECK may realize that transactions which appear to be in conflict on the surface can actually be executed concurrently. This is the advantage of Attribute Level ROLL (ALROLL).

4. ATOMIC COMMIT PROTOCOLS

To achieve atomic commitment for global transactions, we define three classes of LDBMSs. For each we use a variation of the standard Two-Phase Commitment (2PC). Two-Phase Commitment protocols involve a voting phase in which all Global Sub-Transaction Managers (GSTMs) vote yes or no in response to a PREPARE message from the Global Transaction Manager (GTM), acting as the coordinator. Before voting yes, the GSTM must guarantee that it can go either way (Commit or Abort) at that site. In the presence of local autonomy this state is difficult to achieve. The second phase of a 2PC protocol is the decision phase in which the GTM makes a decision and broadcasts it to the GSTMs (all GSTMs must have been READY in order for the GTM to come to a Commit decision). Based on the commitment features provided, we classify LDBMSs into three classes, Class-P, Class-R and Class-O. Each of these three classes vary with regard to their voting phase.

The following theorem is given to motivate our approach to atomic commitment.

THEOREM 1: For global atomic commitment, the level at which the local transactions enter the system must guarantee rigorousness.

PROOF OF THEOREM 1: Let us assume that rigorousness is not provided at the local transaction level. To appear in the same serialization partial order at all involved sites a global sub-transaction G1 should not conflict with another global sub-transaction G2. If a direct conflict occurs, the LDBMS will resolve it. But if an indirect conflict ($r_{G2}(y)\ w_L(y)\ r_L(x)\ w_{G1}(x)$) occurs the LDBMS cannot detect it. However, if rigorousness is guaranteed, the local transaction which caused the indirect conflict be forced to be wait until the earlier global sub-transaction has committed. Hence by rigorousness being guaranteed, indirect conflict occurrences are excluded and atomic commitment can be achieved.

Class-P

Class-P LDBMSs are those which externalize a PREPARE command and, hence, offer a visible "READY" state. The GSTM issues a "PREPARE" and waits for acknowledgment. Upon receiving acknowledgment, the GSTM votes "yes" and moves to the READY state.

For Class-P LDBMSs using concurrency control units in which the GSTM delays all RELEASE operations until the global decision has been received and implemented. Implementation of the global decision is straight forward. If the decision is to commit, a COMMIT operation is issued locally. If the global decision is to abort, then an ABORT operation is issued locally. Since the LDBMS is assumed to provide local recoverability, no conflicting transaction would have committed out of serialization partial order during the uncertainty period.

For Class-P LDBMSs we use concurrency control units in which the GSTM need not delay other transactions at all. If the global decision is to commit, a COMMIT operation is issued locally. If the global decision is to abort, an ABORT operation is issued locally.

Class-R

Class-R LDBMSs guarantee rigorous execution of transactions at their site i.e. transactions are not allowed to conflict with uncommitted transactions. Examples of rigorous DBMSs include all systems using strict 2 phase locking or conservative timestamp ordering as their concurrency control method. The GSTM force-writes a READY-record to the DT-Log READY-list upon receiving the acknowledgment of its last read or write from the LDBMS. It then moves to the READY state and votes "yes". It is noted that the global sub-transaction has not yet committed locally and, thus, no conflicting operations will be allowed by the LDBMS during the uncertainty period. If the global decision is to commit, the local commit is issued and when the commit is acknowledged, the READY-record is removed. If the global decision is to abort, an abort is issued locally (resulting in the sub-transactions being rolled back). If however, the site failed during the decision phase and the decision was to commit, all uncommitted global sub-transaction are reissued in DT-log READY-list order upon recovery. We note that no local transaction which is dependent on an uncommitted global transaction could have committed due to local rigorousness. Thus, upon recovery, these local transactions will be rolled back by the LDBMS itself. The re-issue order of these local transactions may not be the same as the original issue order, but that will not affect global serializability.

Class-O

Class-O is the default class in which the LDBMS commitment features are not known.

5. RESIDUAL ATTRIBUTE SURROGATE (RAS) ACCELERATION

In the previous sections, we have shown how ROLL concurrency control methodology can be used to enforce concurrency control in a heterogeneously distributed environment. Apart from providing transaction management, HDDBMSs also need to support techniques for query optimization. In general concurrency control can be integrated with query optimization if these two processes use the same data structures containing the same information. Residual attribute surrogates are used to accelerate joins and other operations used in the processing of queries and transactions (PER90, GUS92, SCO92]). The main data structure is a bit vector (the RAS vector) in which each position represents one of the instances of the objects occurring in a class (or in the simple relational model, one tuple in a relation). The presence or absence of that object is signified by the presence or absence of a 1-bit in the corresponding position in the vector.

For the purposes of illustration, let us assume that only one access mode is supported by our concurrency control methodology and that the objects under consideration are relations. Under these assumptions it is possible to use RAS vectors to form Select-Omit-Vectors (SOV's, explained below) which are interchangeable with the MVs used in the forgoing. There would be two basic steps in the execution process: First, all items covered by the involved RASs would be requested in the read mode only (so that the state doesn't change while the acceleration technique is being applied). When granted, all RAS processing would be done, resulting in fully reduced SOV's. These fully reduced SOV's specify exactly which items need to be accessed and which do not. They would then be logically "ANDed" with the original blanket request vectors giving fully reduced request vectors, and processing could proceed from there.

The following material explains the nature and use of RAS vectors. RAS vectors have three characteristics which distinguish them from other bit filtering schemes. 1) They are persistent: They become a part of the access mechanism for the object class for which they are implemented and maintained regardless of query activity. 2) They are not hashed: Each bit in a RAS vector represents one object in its class. 3) Finally, they are based on the extant domain: The RAS vector represents all existing objects of a particular class. A RAS vector also has room for expansion, but it does not represent all possible instances in the class.

Along with the RAS vectors come three other structures. 1) Each class for which RAS vectors are maintained requires a RAS vector table. This provides the mapping between RAS vector positions and objects. 2) A class for which a RAS vector is maintained may have a RAS vector index. This provides the mapping between RAS vector positions and the addresses of the corresponding objects. 3) Select-Omit-Vectors (SOV's) are temporary bit filters in which the positions represent all object instances in their access address order. For a given set of objects, 1- bits can be set in the SOV using a RAS index, allowing objects to be accessed by occurrence order. For a complete and detailed description of these structures, the reader is referred to [GUS92].

RASs have little if any acceleration potential for unstructured objects. Within the relational model, RASs can greatly accelerate Select-Project-Join queries. Although RAS acceleration is not dependent on any particular joining technique, its most recent and best version utilizes hash joining for the execution of joins. RAS acceleration results from three factors: the speed of operation on bit vectors; the restriction of I/O allowed by full reduction to only those objects participating in the final result; and the elimination of paging through the combined effects of RAS vectors, SOVs, and a hash join algorithm.

RAS vector tables are needed in order to create and maintain RASs and RAS indexes, but they are not used during query processing itself. They are incorporated within the global federation translation tables. RAS tables need not be ordered by object hierarchy in any particular way and could be ordered by arrival sequence.

5.1 HIT-LISTS

In the relational model (most organizations will continue to have local relational data to manage for many years to come), complex queries can involve more than one attribute in the same relation. This section explains how such relational queries can be optimized at the surrogate level using the Hit-List version of of the RAS concept.

Previous research has shown that RAS vectors with their associated tables and indexes are effective in accelerating queries with only binary or star equijoins, but they are not adequate when multi-way joins are present.

There is no correspondence between RAS vectors for different classes. i.e., it makes no sense to logically equate positions of two RAS vectors of different classes, since their positions represent entirely different sets of incompatible objects. This may be referred to as the crossover problem, and hit-lists are introduced to solve it [SCO92].

RAS vectors use object surrogates maintained in the RAS vector tables. Using these same surrogates, hit-lists are data structures which provide the mapping between the surrogates of one class to another class. In a query, hit-lists supply the means for achieving complete reduction, thereby restricting I/O to only those objects participating in the final result for generalized multi- way join queries, just as RAS vectors do for binary and star joins.

A hit-list is essentially an index and contains paired surrogate values for two classes. In general it is possible to have two hit-lists for each pair of classes, providing access on each surrogate. However, only one is needed, and it does not matter which one is chosen. If the paradigm of complete query reduction, followed by data transmission and join processing is adopted ([HEV79], [YU84]), hit-list accomplishes the same.

As with binary joins using RAS vectors, it is possible to restrict retrieval to only those objects in the final result, but it requires slightly more complex SOV processing. The fully reduced join vectors are sufficient for forming SOV's and retrieving objects that participate in a join. The process is exactly the same as outlined in the previous section for RAS vector processing.

However, for a collection of complex objects participating in a multi-way join on two classes (crossover) the fully reduced join vectors are not sufficient without additional processing. Since a one-to-many or many-to-many relationship may be present in the crossover, a selected object from one class might exist with unselected objects from the other class. Thus, a fully reduced join vector on either class alone does not assure that only objects participating in the final result will be retrieved. This problem is overcome by using both fully reduced join vectors on each class with crossover to create two SOV's, one based on each class. These SOV's are on the same underlying collection of objects, so their bitwise logical "and" can be formed, giving a fully reduced SOV.

The two stage nature of the technique, reduction followed by joining, is not changed by refinement. However, in n-way joins on different objects, it becomes clear that the reduction stage itself consists of 2 phases, the attainment of complete reduction on one participating class, followed by its propagation to all other participating classes.

6. CONCLUSIONS AND FUTURE WORK

OOHYDRO is a project undertaken jointly by North Dakota State University and IBM Rochester, MN, to develop and implement a Heterogeneous Distributed Database System which will connect a variety of autonomous local database systems and object stores into one unified system. Each local system is assumed to be an off-the-shelf, binary-licensed system. OOHYDRO achieves global serializability using a set of objects based on the ROLL (Request Order Linked List) object developed in ([PER89], [PER91a]). ROLL is based on the general Serialization Graph Tester methodology [BER87], and provides freedom from idle-wait, deadlock, livelock and restart. Atomic commitment is based on Two-Phase Commit.

Three classes of local DBMSs are defined and atomic commitment protocols are defined for each. In the absence of a visible PREPARED state offered by the local DBMS or rigorous execution guarantees, the PREPARED state is achieved by committing the transaction locally and protecting writes during the uncertainty period.

We have shown how global serializability and atomic commit can be attained in a Heterogeneous Distributed Database and Object store System in which local autonomy is provided to the LDBMSs. OOHYDRO uses global serializability as a correctness criterion. Serializability is achieved using the Request Order Linked List (ROLL) method within a global OOHYDRO object, GHYDRO, and local OOHYDRO objects LHYDROs at each site. ROLL provides freedom from idle-wait, deadlock, livelock and restart. It is based on the general Serialization

Graph Test methodology, which allows all serializable executions. Atomic commitment is achieved using Two-Phase commitment.

Implementations of OOHYDRO are in progress in a HDDBMS setting consisting of a Solbourne Postgres), Sun workstations (Ingres), DECstations (Postgres), NeXTs (Sybase), AT&T 3B2s (Informix) and IBM AS/400s.

The LHYDRO object communicates with an object called the local coupler which is linked to the global coupler. The global coupler sends and receives messages to and from GHYDRO. This is the only mechanism by which a LDBMS communicates with the GHYDRO object. The communication is standardized between the global coupler and GHYDRO using a surrogate language. Implementation of the LHYDRO module can be done using any method and communication standard found advantageous.

For the prototypes currently being implemented, we have made a few assumptions and will be relaxing these assumptions in future versions. The global surrogation tables are maintained by GHYDRO and are centralized. Any changes made at the local sites that will result in changes to the federated schemata are reported to GHYDRO only at quiescent periods. Also federated data can only be updated by global queries. As of now, we assume that all the data at local sites are global data and that all the queries have to go through the GHYDRO.

REFERENCES

[ATK90] D. DeWitt, K. Dittrich, D. Maier and S. Zdonik, "The Object Oriented Database System Manifesto", M. Atkinson, F. Bancilh on, Deductive and Object-OrientedDatabases, W. Kim and S. Nishio editors, Elsevier Science Publishers, North-Holland, 1990.

[BER87] P.A Bernstein, V. Hadzilacos and N. Goodman, Concurrency Controll and Recovery in DBMS, Addison-Wesley, 1987.

[BRE90] Breitbart, Y., Silberschatz, A., and Thompson, G., Reliable Transaction Management in Multidatabase Systems SIGMOD, 1990.

[ELM88] Elmagarmid, A. and Helal, A.A. , Supporting Updates in Heterogeneous DDBMSs, IEEE Data Engineering Conf., 1988.

[GUS92] Gustafson, J.W., Perrizo, W., Scott, K,. Thureen, D,. and Davidson, W,. DVH: A Query Processing Method using Domain Vector and Hashing, IEEE RIDE-TQP, 1992, Mission Palms, AZ, Feb. 2, 1992.

[HEV79] Hevner, A. R., and S. B. Yao, Query Processing in Distributed Database Systems, IEEE Transactions on Software Engineering, Vol. SE-5, No. 3, May 1979, pp. 177-187.

[KOR90] Korth, H.F., Levy, E. and Silberschatz, A., A Formal Approach to Recovery by Compensating Transactions, Proc. of VLDB-90 pp. 95-106, 1990.

[MCC92] Lester I. McCann and William Perrizo, "Improved Supercomputing Database Concurrency Control with ROLL" NDSU-CSOR-TR-9216 NDSU, Fargo, ND.

[PER89] Perrizo, W. and Richter, R., Concurrency Control Using an Extended Query Language, 4th Int'l Conference on Supercomputing, Santa Clara, CA, 1989.

[PER90] Perrizo, W. "DVAs: A Query Accelerator For Relational Operations". NDSU-CS-TR-90-47. NDSU, Fargo, ND.

[PER91a] Perrizo, W., Request Order Linked List (ROLL): A Concurrency Control Object, Proc. of the IEEE Conf. on Data Engineering, April, 1991, Kobe, Japan.

[PER91b] Perrizo, W., Rajkumar, J., Ram, P., HYDRO: A Heterogeneous Distributed Database System, Proc. of the 1991 ACM SIGMOD Int'l Conf. on Management of Data, Denver, CO, May, 1991.

[SCO92] Scott, K,. Multi-Way Equijoin Query Acceleration Using Hit-Lists, Ph.D. Dissertation, NDSU, 1992.

[TUL90] NSF Workshop on Multidatabases and Semantic Interoperability, Tulsa, OK, November 2-4, 1990.

[YU84] Yu, C. T., and C. C. Chang, Distributed Query Processing, ACM Computing Surveys, Vol. 16, No. 4, December 1984, pp. 399-433.