R*: AN OVERVIEW OF THE ARCHITECTURE

R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng,
R. Obermarck, P. Selinger, A. Walker, P. Wilms and R. Yost

IBM Research, Almaden Research Center, Ca. USA.

Abstract

R* is an experimental distributed database system being developed at IBM Research to study the issues and problems of distributed data management. R* consists of a confederation of voluntarily co-operating sites, each supporting the relational model of data and communicating via IBM's CICS. A key feature of the architecture is to maintain the autonomy of each site. To achieve maximum site autonomy SQL statements are compiled, data objects are named and catalogued, and deadlock detection and recovery are all handled in a distributed manner. The R* architecture, including transaction management, commit processing, deadlock detection, system recovery, object naming, catalog management, and authorization checking is described. Some examples of the additions and changes to the SQL language needed to support distributed function are given.

1. TRENDS IN DATABASE AND DISTRIBUTED PROCESSING SYSTEMS

There are several factors leading to the development of distributed database management systems. People and organizations share data because of its intrinsic value; indeed corporations regard their data as a major asset. The number of computer installations is increasing due to declining costs of hardware. Users of these installations need to share data to do their work, and now that computers can easily be interconnected by electronic networks, distributed systems are evolving for data exchange. Database systems provide consistent views of data, concurrency control for multiple users and recovery in case of failures by using the notion of transaction processing. A database management system can therefore be expanded into a distributed database management system, DDBMS, to supply the same application features to a network of users potentially able to share all the data at all the sites.

Each site should retain local privacy and control of its own data. It is also very desirable to present data to programs and users at a site in the network as if that site were the only one. Furtermore to achieve best performance programs should run locally if all the data for the program is local to the site. Specifically there should be no central dependencies in the network; no central catalog, no central scheduler, no central deadlock detector or breaker, etc. We call this concept of maximum independence "site autonomy"[21].

With these goals in mind, we have designed a DDBMS, called R*, using the relational database technology. It is a follow-on project from System R [4], [5]. The SQL language[6] has been extended where necessary to allow for new functions but existing SQL programs that ran in System R should also run in R*.

The paper describes the environment in which R* runs and the data forms to be supported by R*. The skeletal architecture of R* is described by following the processing of a query entered at one site; query processing in R* is similar to the query processing in System R. The major issues and processes are:

- Environment and Data Definitions
- Object Naming
- Distributed Catalogs
- Transaction Management and Commit Protocols
  - Transaction Number
- Query Preparation
  - Name Resolution
  - Authorization Checking
  - Access Path Selection and Optimization
  - Views
- Query Execution
  - Concurrency
  - Deadlock Detection and Resolution
  - Logging and Recovery
- SQL Additions and Changes

R* is partially implemented (see Status section 9).

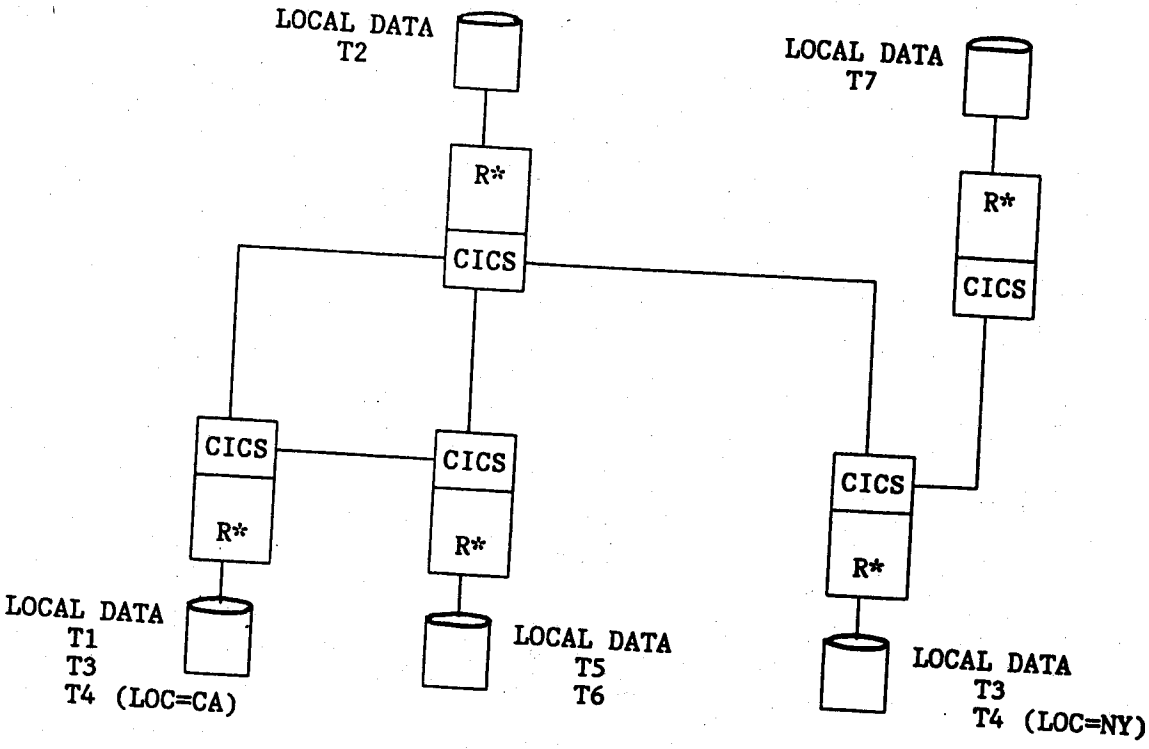
2. ENVIRONMENT & DATA DEFINITIONS FOR R*

R* consists of several database sites that communicate via CICS, an IBM software product[7], as shown in Figure 1. CICS was chosen as the communication medium to minimize our prototyping efforts so that we could concentrate on distributed database, DDB, issues. Any network configuration and interconnection topology allowed by CICS can be used in R*; CICS communications is used merely as a transport medium. R* also considers the communications medium to be unreliable, i.e., message delivery is not guaranteed, however when messages are delivered to the database software they are assumed to be delivered intact in order and without duplication. R* runs in a CICS address space, and CICS handles terminal I/O, program and task management. Note that sites in R* would typically be physically separate computer systems, but sites need to be only logically distinct, not physically distinct, especially for initial development purposes (there are already enough problems using one machine for debugging complex software)!

Data in R* is stored in tables (relations) that may be dispersed, replicated or partitioned. Combinations of these distributions are supported also; for example partitioned data may be replicated. Dispersed data means that tables T1,....Tn are uniquely stored at the various sites S1,.....Sk; e.g., T1,T2 at S1; T3 at S2 etc. Replicated data means that copies of a table exist and are guaranteed to be identical at all times. All copies are updated synchronously, using a two-phase commit protocol. Partitioned data is logically one table, part of which is stored at one site, and another part(s) at another site(s). In horizontal partitioning some rows (tuples) are stored at one site, some at another site according to some disjoint separation criteria based on column values (e.g. store values 1 to 100 in site 1 and all the rest in site 2). In vertical partitioning, some columns are stored at one site, some at another, according to some column separation criteria. The vertical fragmentation must be lossless so that the original relation can be created from the vertical partitions[3] and therefore all partitions must have a set of common columns that determine the values of all columns in the complete relation, or a virtual column created artificially by the DDBMS to enable a one-to-one match of the fragmented tuples during reconstruction. Partitioned data can also be replicated. The end-user of the database need not be aware of the data distribution for query execution or for application programming. Data consistency, reassembly of partitions and access of remote data is performed by the DDBMS itself and is transparent to the end-user.

The SIRIUS/DELTA DDBMS also allows partitioned and replicated data forms[19]. The Polypheme prototype went further and allowed for heterogeneous database systems in its design. The technique was for all systems to employ a standard relational form for communications to other sites, and a mapping to the internal form at each site [1] CICS supports distributed data processing through function shipping using the Inter-Systems Communications feature[7]. Also using function shipping, TANDEM systems support distributed data, with an emphasis on availability, but few technical papers exist on their systems[34].

In R*, some tables can be snapshots of other tables[2]. Snapshot data is a copy of a relation(s) in which the data is consistent but not necessarily up-to-date. Snapshots are read-only and are intended to provide a static copy of a database (e.g. Friday's sales figures). They are not updated when the base relation is updated. They may however be periodically refreshed by recopying the data from the base relations (e.g. every Friday at 6 p.m.). Programs that only need snapshot data might run more efficiently from snapshots than from current relations. For example relations that are locally stored snapshots derived from a remote operational database would allow local programs to run much faster from the snapshots than from the operational data. Availability is increased also for transactions that use snapshot data.



T1, T2, T5, T6, T7 : Dispersed Data
T3 : Replicated Data
T4 : Partitioned Data (by tuples with LOC=CA or LOC=NY)

Figure 1. R* Distributed Database Configuration.

3. OBJECT NAMING

The naming problem in distributed systems is to allow data sharing but without undue restrictions on an end-user's choice of names. For autonomy reasons we don't want to have a global naming system, nor do we want to force users to choose unique names. Furthermore adding a new site with a previously defined database to a previously established network would lead to terrible renaming problems.

Names used in SQL statements are names chosen by end-users writing ad-hoc queries or application programs. Network details must be transparent to such users, so that programming is as simple as possible and also so that the same programs will work correctly when entered at any site.

We solve the problem by mapping end-user names, which we call "print names", to internal System Wide Names, "SWN". An SWN has the form:

```
USER @ USER_SITE.OBJECT_NAME @ BIRTH_SITE
```

The BIRTH_SITE is the site in which the object was first created. Because site names are chosen to be unique outside of the system, an SWN is unique. Name completion rules for adding default parts to print names and synonym mapping tables for each user are used to convert a print name to an SWN. For example if BRUCE logs on in SAN_JOSE and accesses a table he calls T, which was locally created and stored in San Jose, then the SWN:

```
BRUCE @ SAN_JOSE.T @ SAN_JOSE
```

is generated.

This mapping mechanism allows different end-users to reference either the same object with different print names or different objects with the same print names. Objects may be stored and moved without impacting user code (see catalogs in section 4) and this location transparency mechanism permits site autonomy. A more complete discussion can be found in[22].

4. DISTRIBUTED CATALOGS

In the SDD-1 system[28],[28], the catalog is logically a single table, which can be fragmented and replicated. This allows catalog entries to be replicated and distributed among the data module sites. However, this also implies that local objects may have their catalog entries at a remote site and that data definition operations may not be totally local. SDD-1 does cache catalog entries to aid performance but this also adds overhead for updating purposes. A distributed version of INGRES[33] distinguishes between local

relations (accessible from a single site) and global relations (accessible from all sites). The name of every global relation is stored at every site. Creation of a global relation involves broadcasting its name (and location) to all sites of the network, but cached catalog entries are supported. Thus both SDD-1 and INGRES implement a global catalog and therefore restrict site autonomy and complicate system growth. For a large network any operation requiring unanimous participation will create difficulties, and may require complex recovery mechanisms.

R* uses a distributed catalog architecture. Catalogs at each site keep and maintain information about objects in the database, including replicas or fragments, stored at that site. In addition the catalog at the birth site of an object keeps information indicating where the object is currently stored and this entry is updated if an object is moved. Cataloging of objects is done in this totally distributed manner to preserve site autonomy. An object can be located by the system from its SWN and no centralization is necessary.

For performance reasons, a catalog entry can be cached at another site so that a reference to it can be as efficient as a local reference e.g., for compiling (see section 6). A cached entry may become out-of-date if another transaction has changed the structure of or the access paths to the object after the cached entry is made; this fact is discovered during a later processing step and then the cached entry is updated from the correct catalog entry and the initial processing must be restarted. This discovery is made because entries have version numbers which are checked during subsequent processing against version numbers store in the real catalog entries to determine if the cached entry used at an earlier stage was valid. Restarting in this way is not expected to occur very often because catalog entries are relatively static.

The catalog entry for an object includes the object SWN, type and format, the access paths available, a mapping in the case of a view to lower level objects, and various statistics that assist query optimization. Each entry is identified by the SWN of an object. To find an object's catalog entry, first the local catalog, (plus the local cache), then the birth-site catalog and then the site indicated by the birth-site catalog are checked in that order, stopping when the catalog entry is found. This gives the best efficiency together with site autonomy.

5. TRANSACTION MANAGEMENT AND COMMIT PROTOCOL

5.1. Transaction Number

The DDBMS must support the notion of a **transaction.** A transaction is a recoverable sequence of database actions that either commits or aborts. If the transaction commits, all of its changes to the database take effect, but if the transaction aborts, none of its actions have any effect upon the database state. In SDD-1 and in INGRES each statement is a transaction whereas in System R one can define a transaction to be any number of SQL statements.

A transaction starts at the site where it is entered. Subsequently agents may be created at other sites to do work on behalf of the transaction. Both synchronous and asynchronous execution can be performed to take advantage of parallelism or pipelining during the compilation and execution of the transaction.

Any request to the DDBMS is given a transaction number that is made up from the site name and a sequence number (local time of day may be better) Each site is unique and the sequence is increased for each new transaction. Therefore the transaction number is both unique and ordered in the R* network. Uniqueness is necessary for identification purposes, for acquiring resources, breaking deadlocks etc. For example if a transaction starts at site A, sends work to site B, which in turn sends work to site A then it is necessary for A to know that both pieces of work are on behalf of the same transaction so that locks on data objects can be shared.If such locks could not be shared a deadlock would occur and make processing the query impossible. Ordering is used to provide a means of knowing which transaction to abort in the case of a deadlock between different transactions. R* aborts the youngest, largest numbered, transaction.

5.2. *Transaction Commit Protocol*

Whenever a transaction's actions involve more that one database site, the DDBMS must take special care in order to insure that the transaction termination is **uniform:** either all of the sites commit or all sites abort the effects of the transaction.

The so called "two phase" commit protocol[7],[13],[20], [24] is used in order to insure uniform transaction commitment or abortion. The two phase commit protocol allows multiple sites to coordinate transaction commit in such a way that all participating sites come to the same conclusion despite site and communication failures. There are many variations of the two phase commit protocol. In all variations there is one site, called the **coordinator**, which makes the commit or abort decision **after** all the other sites involved in the transaction are known to be recoverably **prepared** to commit or abort, and all the other sites are **awaiting** the coordinator's decision.

When the non-coordinator sites are prepared to commit and awaiting the coordinator's decision, they are not allowed to unilaterally abandon or commit the transaction. This has the effect of **sequestering** the transaction's resources, making them unavailable until the coordinator's decision is received. Before entering the prepared state, however, any site can unilaterally abort its portion of a transaction. The rest of the sites will also abort eventually. While a site is prepared to commit, local control (autonomy) over the resources held by the transaction is surrendered to the commit coordinator.

Some variations of the two phase commit protocol sequester resources longer than other variations. Rosenkrantz, Stearns, and Lewis[30] require all sites other than the single active site of the transaction to be prepared at all times. The linear commit protocols described in[13] and[20] have a commit phase with duration proportional to the number of sites involved.

R* actually uses a presumed-to-commit protocol. The number of messages required in the usual two phase commit protocol is 4(N-1), where N is the number of sites involved in the transaction, but by assuming the commit succeeds the number of messages can be reduced to 3(N-1). If a failure requiring transaction abort occurs, then all 4(N-1) messages are needed. The improvement is obtained by removing

the need for acknowledging the commit message. The coordinator logs the start of the transaction commit processing and then sends messages to the other sites (called apprentices) involved in the transaction. An apprentice is a site that does work at the request of another site, which then is called the master. Each apprentice still has to log its decision and reply to the coordinator who logs the resulting decision. Therefore the apprentice has to await commit/abort instructions from the coordinator, during which time its resources are tied up. Lost commit messages are detected by a time-out, but to avoid the very long outages that could occur if a network breaks down, operator intervention must be permitted.

This presumed-to-commit protocol minimizes the duration of the commit protocol. Other variations, notably those proposed for SDD-1[15], provide mechanisms for circumventing the delay caused by coordinator failure, by sending extra messages (to nominate a backup coordinator) which prolong the commit phase in the normal case. It does not appear possible to completely eliminate the temporary loss of site autonomy during the commit procedure[18].

6. QUERY PREPARATION

6.1. Name Resolution

When an SQL statement is first seen by R*, it is parsed and then undergoes name resolution in which all SQL print names are resolved into SWNs. Then it is possible to determine if catalog entries for each database object are available locally. If any entries are missing because they are stored at remote sites and not in the local cache then a message has to be sent to the remote site to fetch catalog information for the remote objects from the birth site as described in section 4.

6.2. Authorization Checking

After name resolution the authorization of the user to perform operations indicated by the SQL statement on **local** data is checked. Because all sites are cooperating in R* voluntarily, no site wishes to trust other sites with respect to authorization. Therefore authorization checking for a remote access request must be done at the site that stores the data and all controls for accessing data must be stored at the same site as the data being controlled. It is possible to control all access to local data locally without the need to contact another site, thus preserving site autonomy. Each site is responsible for maintaining its own site authorization using passwords etc. Authorization for data access is checked for each user, but remote sites authenticate on a site-to-site basis when responding to a request for data. A remote site is trusted to have validated its own users. If this breaks down the damage is limited to the aggregate of privileges held by the users at that site.

Thus the authorization entity in R* is a user at a site, and user level authorization semantics are enforced using site level authentication[35]. For example, PAT @ SAN_JOSE is different from PAT @ YORKTOWN. An object owner, initially the creator, can grant access rights to any other user, local or

remote, and that person can pass on access rights to other users if the original grant permitted subsequent grants (included the grant option). This is the same as in System R [14]. For site to site authentication in networks see encryption techniques[25],[16].

6.3. Compilation and Plan Generation

Just as with programming languages, it is possible to compile rather than interpret the database language. Compilation offloads from execution time to compile time much of the overhead of operations needed to set up the data request and thus improves the performance of repetitively executed data access requests.

For conventional programming languages, compilation is a binding process in which high level constructs are mapped to a low level instruction set, which is fixed by the machine on which the compiled code is to be run. Analogously, in a database system access requests expressed in a very high-level database language, such as SQL, can also be compiled into an access program which uses low level objects [6]. This compilation includes a binding process in which the requests are bound to required authorizations, data objects, and the paths to access them. However, one of the primary differences between the compilation of programs written in conventional programming languages and the compilation of programs written in database languages lies in the fact that the latter depends on objects that are subject to change. Between compile time and execution time, a relation may be deleted or moved, an access path may be dropped, or a required privilege may be revoked. Recompilation or invalidation is necessary when such items change.

In a DDBMS data objects accessed and access paths used may reside at a remote site and the question as to where binding should be done arises.

The approaches can be grouped into three classes:

- All binding for every request can be done at a chosen site;
- All binding can be done at the site where the request originates;
- Binding can be done in a distributed way,

at the sites where data objects are accessed.

The first approach would not function well because it is a centralized approach and suffers from poor efficiency and lack of resiliency to failure. It would require a centralized catalog and therefore an excessive amount of communications in the network. The advantages offered by a DDBMS would be mostly lost if a centralized compiler had to be used for all compilations; it would become a system bottleneck. Site autonomy would be lost in this approach of course.

The second approach is not good either. First, to preserve each site's autonomy, it should not be necessary to get agreement from all sites at which requests have been compiled before another site can change an access path for its locally stored relation. Secondly, the compiling site should not need to remember and record the physical details of data access paths at other sites since individual databases may be changed, for example by adding a new access path. Thus if a program depends on a relation that has been changed at a remote site we do not want to do a global recompilation for the whole program if we can avoid it by doing a local recompilation at the remote site for part of the program. Also, to protect data in a high level DDBMS, a user at a certain site may choose only to grant access to a view which is an abstraction

of underlying physical data objects, rather than granting access to the objects themselves, which means that the entire compilation and binding cannot be done at the originating site.

The third approach overcomes the drawbacks mentioned above and offers additional advantages. The master site can decide inter-site issues and perform high-level binding and the local sites can decide local issues and do a lower-level binding (e.g. for access path selection). When compilation is distributed and the portion of program to access and manipulate a site's data objects is generated at the same site, it follows naturally that if recompilation has to be done due to changes of local objects, it can be done on a local basis. Thus distributed compilation allows for local control in apprentice sites, which preserves site autonomy. However, global optimality becomes more difficult to obtain after local changes have occurred and it is desirable to be able to do a complete global recompilation, optionally, to improve execution efficiency in some cases. Other advantages of distributed compilation are that failure resiliency is also improved by limiting the scope of actions to a local site when possible. Also different versions or releases of system code could exist at different sites and it would still work correctly.

The INGRES DDBMS [33] seems to be moving towards a compilation approach of preparing queries for execution but most distributed systems still use interpretive methods. R*, using the third approach above, performs distributed compilation [9],[26]. In R* the site where the SQL query enters becomes the master site and to compile a request for data at mutiple sites, the overall global plan for executing the program has to be created at the master and then communicated to the apprentices. The difficulty in the design of distributed compilation arises because we want to compile programs to achieve global optimality for execution but retain local site autonomy. The master site may not have complete and up-to-date knowledge of the data objects and access paths available at the apprentice sites, therefore it may make poor decisions and generate very inefficient code if it did the complete compilation. Incorrect decisions can be detected by an apprentice site by checking the version number of the information on which the decision was based, but the extra overhead of correct but distributed compilation is the cost incurred for site autonomy and data protection. The overhead in this case is to redo the entire compilation. The solution is that the master chooses the execution plan, join order, which sites do work etc. and apprentices choose how to access local data.

The global plan is a structural skeleton of the **access strategies** and is generated at the master site using the information available at the master site. If the master has insufficient catalog information about data objects at other sites, it can request the necessary information and cache it for later use. The global plan specifies the invocation sequence of the participating sites and the order of parameters. If the SQL statement requires a join, the global plan would also specify the join order and join methods.

The global plan is a high level representation of the decisions made by the master site with regard to the execution of the SQL statement. The optimal choice of access paths is discussed later in section 6.4. A global plan should be globally optimal if the information used in access path selection is correct. In addition to the global plan, the compiler at the master site also generates a set of local execution strategies for the local data objects accessed. This includes, for example, which index to use and whether sorting is done. The selection of the global plan and the local execution strategies is termed the "path selection" phase.

The global plan together with the SQL statement is sent to those remote databases that contain the data needed for this SQL statement. This processing phase is termed the "plan distribution" phase. In the global plan, references to data objects are made in terms of their relative positions in the SQL statement. The use of the SQL statement for the expression of the action needed, together with the fact that the global plan is a high level representation, solves the problem of version incompatibility of system code among DBMS's at different sites.

At the remote apprentice sites the first task performed is to check the validity of the catalog information that the master used. If an out-dated version was used, an error message is returned to the master site and the global plan is re-generated by the master using updated information. Compilation in an apprentice site follows the same pattern as at a master site except that no name resolution is needed. The decisions made by the master site concerning the interfaces among sites to execute this SQL statement will be followed. However, the apprentice is free to change the sequence of the local operations. For example, if the SQL statement requires a join, then the apprentice site can change join orders and join methods for local relations as long as the result tuples are presented in the order prescribed, if any, in the global plan. The apprentice may also use access paths unknown to the master. The access path selector in the apprentice site also generates a set of local execution strategies, which, in turn, undergo the code generation phase to produce local subsections. Figure 2 shows the compilation process.

Just as in a programming language compiler, there is a code generation step. Code is generated at the master site and at each apprentice site involved in the distributed compilation; each piece is called a subsection. Each subsection contains code both for calling its local Research Storage System,RSS, which performs local data management, and for passing data and control to other sites according to the overall plan.

The whole compilation for a SQL program or query is processed itself as a transaction. After all the subsections have been generated in the involved sites for every SQL statement in the program being prepared and no errors are detected, the master commits the compilation transaction using the two phase commit protocol. Upon receiving the "prepare" command, each apprentice stores the subsections into an access module. Besides the access module, the SQL statements and the global plans are also stored for recompilation purposes. During execution, subsections call one another as subroutines or coroutines, or they may be executed in parallel.

6.4. Access Path Selection and Optimization

To execute a query efficiently it is necessary to select access paths to data that minimize the total processing time of the query. Epstein studied this problem for distributed INGRES[7] and Selinger for SYSTEM R [31]. The SYSTEM R work has been extended for R*.

During compilation the access paths to data objects are selected and the access path selector in R* tries to minimize total predicted execution time of a SQL statement by exploring a search tree of alternatives and estimating the cost of each[32]. Three components are included to model the execution costs of SQL statements for different access paths in R*: I/O cost , CPU cost and message cost. The cost formulae have the form:

```
TOTAL_COST = I/O_COST + CPU_COST + MESSAGE_COST

I/O_COST =  I/O_WEIGHT * NUMBER_OF_PAGES_FETCHED

CPU_COST =  CPU_WEIGHT * NUMBER_OF_CALLS_TO_RSS

MESSAGE_COST = MESSAGE_COST * NUMBER_OF_MESSAGES_SENT +
          BYTE_COST * NUMBER_OF_BYTES_SENT
```

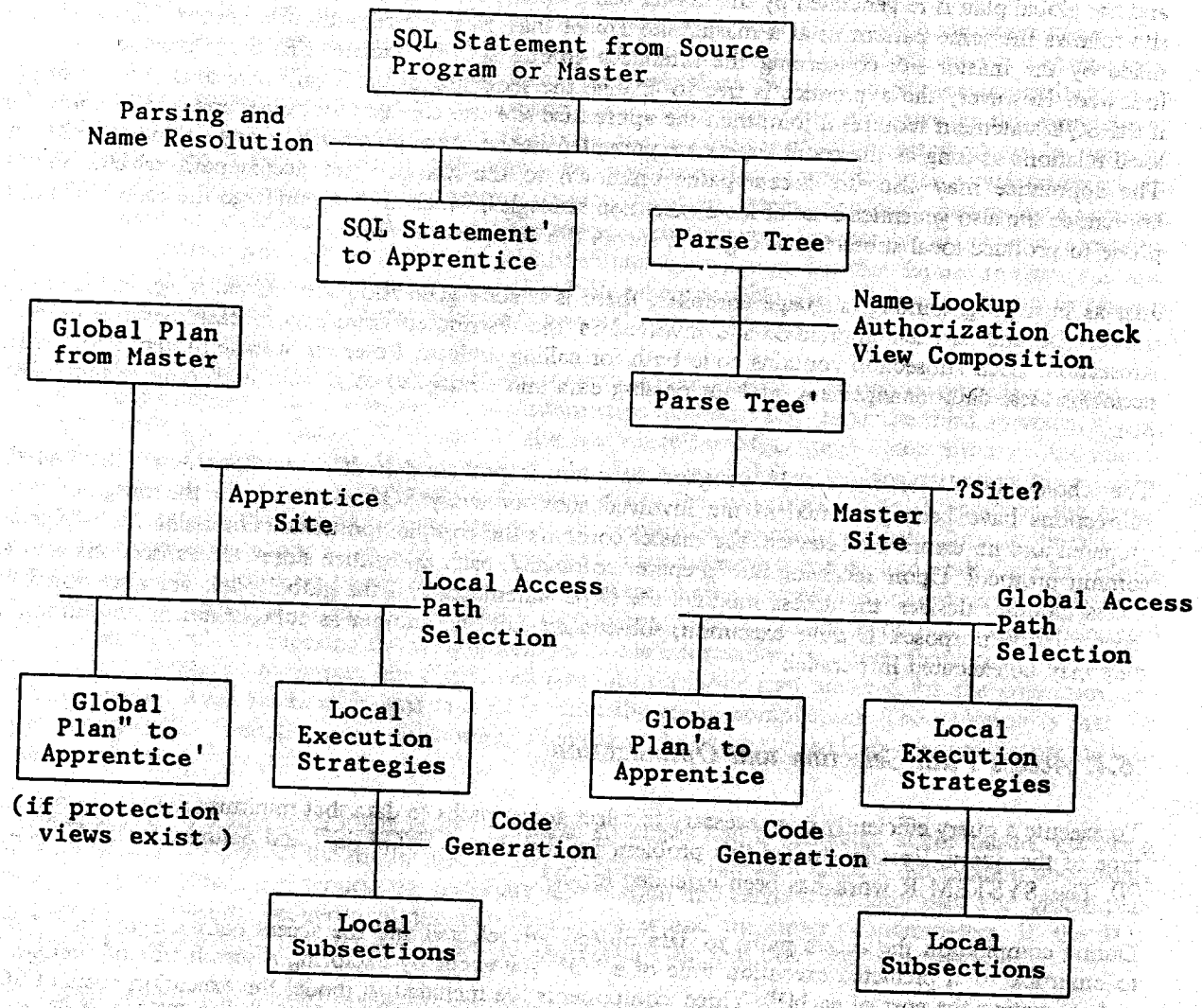


Figure 2. Skeleton of Distributed Compilation.

As in System R, the cost of an access plan is calculated as the weighted sum of the components. NUMBER_OF_CALLS_TO_RSS represents the estimated number of tuples retrieved; and MESSAGE_COST consists of a per message cost and a per byte cost. Both the number of messages and the amount of data moved are minimized together.

In order to take into account the added cost component and the variety of data forms (partitoned and replicated data) that can be created, the access path selector in R* is considerably more complex than that in System R. The cost of accessing a horizontally partitioned relation is the sum of the costs of accessing its components. Note that not all of the components need to be accessed if the path selector can exclude some components by examining the partitioning criteria. The cost of accessing a vertically partitioned relation is the cost of joining the components. Again, the partitioning criteria may reduce the cost. The cost of accessing a replicated relation for read is the minimum of the costs of accessing the replicas. For updates, it is the sum [32].

The situation is more complex in choosing an optimal path for a join of relations which reside at different databases. In addition to the join order, output tuple order, and join method, the join result location becomes another parameter, thereby increasing the branching factor of the search tree. To join the inner relation B residing at site N to the outer relation A residing or produced at site M, R* considers five possibilities:

- All the qualified tuples of the inner relation B are sent to site M and stored in a temporary relation. The join is performed at site M.
- Qualified tuples of the outer relation A are sent to site N.one at a time. Matching qualified inner relation tuples are retrieved and joined to the outer relation tuple, at site N.
- Outer relation tuples are retrieved. For each qualified tuple, a request containing the values of the outer relation's join column(s) is sent to site N. Then matching qualified inner tuples are retrieved and sent back to site M, where the join is performed. This way of obtaining inner relation tuples is termed "fetching as needed."
- All the qualified inner relation tuples are sent to a third site, site P, and stored in a temporary relation. Then (matching) qualified outer relation tuples are sent to site P to perform the join using the temporary relation. This is a combination of the first and second approaches above.
- Outer relation tuples are sent to a third site, site P, and for each of these outer relation tuples, a request is sent to the inner relation site to retrieve the matching tuples. The join is performed at site P. This is a combination of the second and third approaches above.

In executing a chosen join method, advantage is taken of the parallelism and pipelining available in processing an SQL statement. For example, fetching tuples from site A and other tuples from site B can be done in parallel. As another example while inner tuples from site A that match an outer tuple from site B are joined at site A, the next outer tuple can be fetched from site B.

6.5. *Views*

A view in R*, as in System R, is a non-materialized virtual relation defined by an SQL statement. It is defined in terms of one or more tables or previously defined views and during processing a view is materialized from its component objects. Views can be used as shorthand notation for reducing the amount of typing required when frequently executing complex queries, or they can be used as a protection mechanism for hiding rows or columns in underlying tables from the user of the view.

Unlike in System R, in R* view component objects may be at different sites. Therefore to provide data protection between sites a protection view is materialized only at the site owning the view. This scheme prevents sending sensitive data to a node where no user is authorized to see it. Therefore during compilation the master site generates a plan for processing the view as if it were a physical table and sends the plan and SQL statement to the apprentices where the view will be processed (and where view records will eventually materialize.) An apprentice site may itself decompose a view component in terms of other views on tables at yet other sites. In that case the apprentice acts as a master to other apprentices and must generate a plan and send subplans to its apprentice sites. The plan distribution progresses in this way until all views are resolved and may even loop back to a site that has already participated in the compilation.

7. QUERY EXECUTION

Queries are executed by running the compiled code generated during query preparation. The local subsection is loaded and executed and it calls remote subsections as needed. Messages are sent to execute remote-procedure-like calls. Local and remote sections of code call the Research Storage System RSS, which is the same as in System R. The RSS returns one record at a time when it is called; joins and other multiple record handling operations are carried out at a level above the RSS. Therefore the only new features required in the RSS are begin and end transaction functions associated with each unique transaction number. Transaction management and commit protocols to synchronize database changes at the end of a transaction were covered in section 5.

7.1. Concurrency

Distributed transaction processing requires database concurrency control mechanisms[12],[13] in order to avoid interference among concurrently executing transactions. The concurrency control mechanism should not require centralized services for resource allocation or deadlock detection. Distributed concurrency control algorithms, including distributed deadlock detection, which do not impact site autonomy have been developed,[27] and[23,7]. The R* techniques used for deadlock detection and resolution are different from those in System R (section 7.2), but the concurrency control mechanisms used in R* are the same as those used in System R.

7.2. Deadlock Detection and Resolution

The SDD-1 system has an interesting technique for reducing deadlock occurrences. It attempts to analyze the read and write sets of queries and group them into classes that require disjoint sets of resources. Then one query in each class can be run without interference and so no deadlock should occur. However, for unpredictable data references during query execution this technique would not work.

The distributed deadlock detection algorithm in R* attempts to maximize site autonomy, minimize messages and minimize unnecessary processing or other bottlenecks [27]. The basic idea is that each site does periodic deadlock detection using transaction wait-for information gathered locally or received from other sites. Real deadlock cycles are resolved and potential cycles are converted to transaction wait-for 'strings'.

Each 'string' is sent to the next site along the path of the (potential) multi-site deadlock cycle only if the first transaction number in the string is less than the last transaction number in the string. This is an optimization to reduce the number of messages sent. Note also that other orderings could be chosen. This process continues until a cycle is found. The cycle will be found at one site only (due to the transaction ordering) and only sites involved in the transaction will be involved in finding it. This is usually 2 or 3 sites only in what is potentially a very large network. When a deadlock cycle is discovered a standard deadlock cycle breaker program is run and the deadlock is broken by aborting one of the transactions (such as the one that has done the least work so far). The other sites involved in the chosen transaction will be told subsequently to abort the transaction.

An example distributed deadlock is shown in Figure 3. There are three sites 1,2,3 and three transactions X,Y,Z each of which has an agent waiting for resources at a remote site. The algorithm for breaking the deadlock can be understood by following the messages sent from site to site until the cycle is found at site 3.

7.3. Logging and Recovery

The mechanisms used for logging are the same as in System R. As previously discussed each site involved in a transaction has to log data changes and commit decisions during two phase commit. If a site or communications link fails during query execution **before** two phase commit, time-outs will occur at the calling site and the called sites and the transaction will be aborted at all sites. No resources will be sequestered after time-outs have occurred. If a failure occurs **after** a site has entered phase one of the two phase commit then its resources are held by that transaction until communications are re-established and the in-doubt transaction status is resolved and the database made consistent.

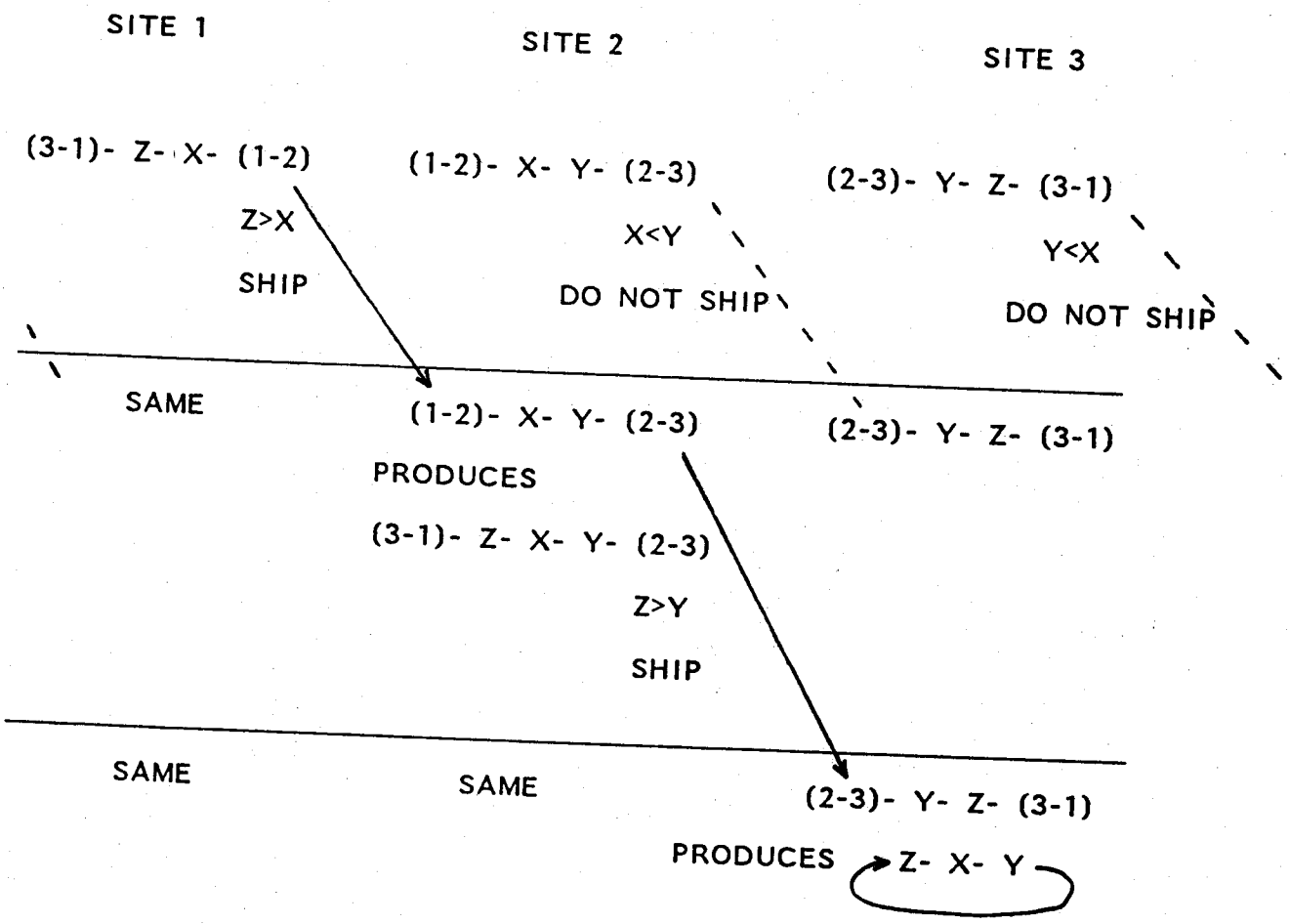


Fig. 3. Global Deadlock Detection.

8. SQL Additions and Changes

The SQL database language (6) developed for System R has been extended for R* and some example extensions are given below. These extensions are not the only ones needed, nor because of space limitations, can each be fully explained. However it is hoped that they give the reader an idea of the kind of high-level non-procedural language statements needed in a distributed database system like R*.

```
DEFINE SYNONYM <relation-name> AS <System-Wide-Name>

DISTRIBUTE TABLE <table-name> HORIZONTALLY INTO
   <name> WHERE <predicate> IN SEGMENT <segment-name@site>
    .
    .
   <name> WHERE <predicate> IN SEGMENT <segment-name@site>

DISTRIBUTE TABLE <table-name> VERTICALLY INTO
   <name> <column-name-list> IN SEGMENT <segment-name@site>
    .
    .
   <name> <column-name-list> IN SEGMENT <segment-name@site>

DISTRIBUTE TABLE <table-name> REPLICATED INTO
   <name> IN SEGMENT <segment-name@site>
    .
    .
   <name> IN SEGMENT <segment-name@site>

DEFINE SNAPSHOT <snapshot-name> (<attribute-list>)
       AS <query>
       REFRESHED EVERY <period>;

REFRESH SNAPSHOT <snapshot-name>

CREATE INDEX <name> ON <table-name>
DROP INDEX <name> FOR <table-name>

MIGRATE TABLE <table-name> TO <segment-name@site>
```

9. STATUS OF R* AND FUTURE PLANS

As of November 1981, when this paper was written, large sections of R* have been coded and tested as an experimental prototype system. The transaction management, communications environment and system interfaces for CICS running under VM or MVS have been coded and run in a single processor. The compiler, optimizer and access path selector have also been written. Code generation from the R* compiler is just beginning. R* has just started sending the first messages for remote catalog look-up and for distributed compilation. The RSS data management, deadlock detection, commit/abort processing, logging and recovery have been tested.

Future work includes linking the coded subsystems together once they are all written, the generation of code from the compiler and the execution and testing of "distributed queries" running, at first, in a single machine environment. Then we will bring up physically separate machines and test actual distributed processing with real message traffic using the CICS/ISC (Inter-Systems-Communications) facility. Next different kinds of data distributions will be examined. By that time we should be able to examine the overall behaviour of R* and to make improvements to the optimizer and to the system code to improve the performance and possibly the design too. R* is a large experimental project and we are under no illusions that we have got it all just right!

10. CONCLUSIONS

We have presented the overall architecture of R*, emphasizing those issues that affect the autonomy of the sites participating in the distributed database. A key ingredient of site autonomy is careful distribution of function and transaction management responsibility among the participating sites. Avoiding global and centralized data and control structures is required, not only to enhance local autonomy, but also to facilitate graceful system growth, data protection and failure resiliency.

Data access authorization is managed by the site holding the data and remote access requests are authenticated. Local control and stand alone processing capabilities require that all relevant catalog structures be locally stored and managed. Distributed query compilation was developed to support local site autonomy. Local representation of compiled query fragments allows local invalidation of the query if local objects or authorizations are modified. Although local control must be surrendered to the coordinator site during the transaction commit protocol, careful selection of a distributed commit protocol can minimize the duration of the loss of local control.

The R* architecture supports several kinds of data distribution. Attention has been given to efficient execution of programs by the compilation and optimization of users queries and SQL programs.

Unlike other DDBMS (e.g. INGRES [33] and SDD-1[29]), R*'s emphasis on site autonomy has led us away from shared control and globally managed or centralized catalog and name resolution structures. At the same time, R* provides transparent remote data definition and manipulation facilities, distributed transaction management, and distributed concurrency control which should simplify data sharing for both ad hoc query users and application programmers. Existing single site SQL programs and queries can be run against distributed data without modification in an R* environment and programmers can continue to develop programs without having to worry about network issues.

11. ACKNOWLEDGMENTS

The following people have contributed to the R* work also, and we would like to acknowledge their main contributions: M. Adiba who developed snapshots, J. Gray who worked on the architecture, recovery and commit, F. Putzolu who developed the RSS data management, and I. Traiger who worked on several distributed systems issues and the RSS data management.