Paradox for Windows • SQL Standards Update • IBM's New DB2 Directions

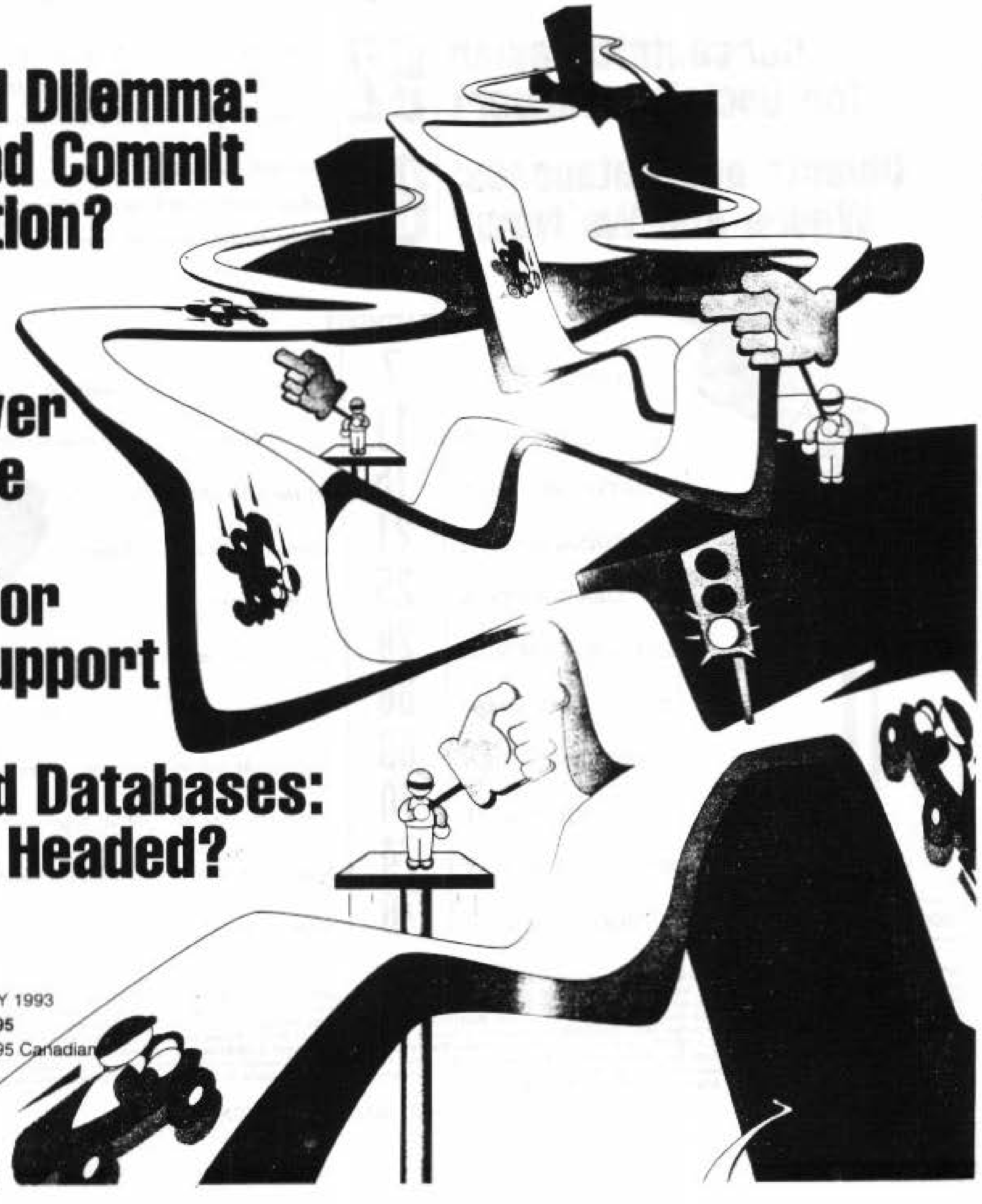
DATABASE Programming & Design

Distributed Dilemma: Two-Phased Commit or Replication?

Client/Server Middleware

Designing for Decision Support

Objects and Databases: Where's It Headed?

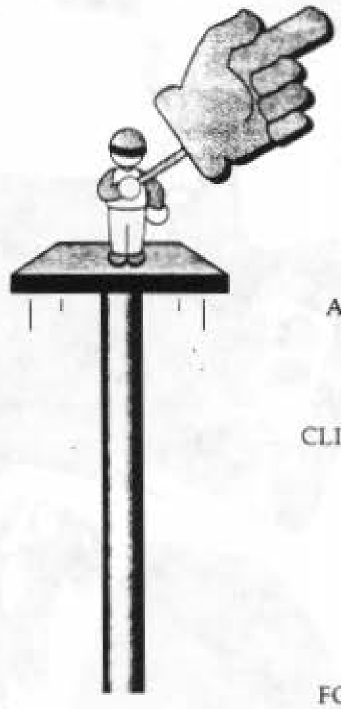


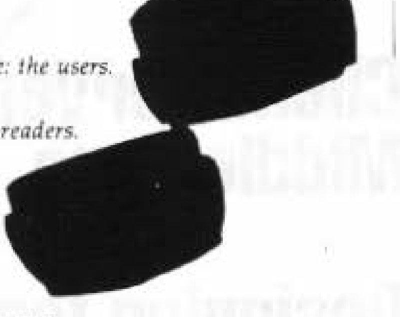
MAY 1993
$3.95
$4.95 Canadian

VOL. 6 NO. 5

DATABASE Programming & Design

MAY 1993





COVER BY: JIM SHINNICK

DATABASE PROGRAMMING & DESIGN (ISSN 0895-4518) is published monthly by Miller Freeman, Inc., 600 Harrison St., San Francisco, CA 94107, (415) 905-2200. Please direct advertising and editorial inquiries to this address. For subscription inquiries, call (800) 289-0169 (outside U.S. (303) 447-9330). **SUBSCRIPTION RATE** for the U.S. is $47 for 12 issues. Canadian/Mexican orders must be prepaid in U.S. funds with additional postage at $6 per year. Canadian GST Permit #124513185. All other countries outside the U.S. must be prepaid in U.S. funds with additional postage at $15 per year for surface mail or $40 per year for air mail. **POSTMASTER:** Send address changes to DATABASE PROGRAMMING & DESIGN, P.O. Box 53481, Boulder, CO 80322-3481. For quickest service, call toll-free (800) 289-0169 (in Colorado or outside the U.S. (303) 447-9330). Please allow six weeks for change of address to take effect. SECOND CLASS POSTAGE paid at San Francisco, CA 94107 and at additional mailing offices. **DATABASE PROGRAMMING & DESIGN** is a registered trademark owned by the parent company, Miller Freeman Inc. All material published in DATABASE PROGRAMMING & DESIGN is copyrighted © 1993 by Miller Freeman Inc. All rights reserved. Reproduction of material appearing in DATABASE PROGRAMMING & DESIGN is forbidden without permission. 16mm microfilm, 35mm microfilm, 105mm microfiche and article and issue photocopies are available from University Microfilms International, 300 N. Zeeb Rd., Ann Arbor, MI 48106. (313) 761-4700.

BY DAVID McGOVERAN

2PC? Replication? Or Both? Here's a primer to help you find your way through the central issues in today's distributed DBMS debate

Two-Phased Commit or Replication?

RECENTLY, A RIVALry has developed between Oracle and Sybase around automatic versus programmatic two-phase commit (2PC) implementations. For example, code for Sybase programmatic 2PC[1] has been contrasted with the Oracle SQL using transparent 2PC in an Oracle Corp. advertisement. Such comparisons are useful for marketing purposes, but do not expose the more important technical issues.

Similarly, when Sybase announced System 10 (the System 10 SQL Server was formerly referred to as Release 5) and Replication Server in November of 1992, the computing industry press began publishing articles that portrayed replication as an alternative to the overhead of 2PC. This inappropriate comparison is technically incorrect and has been widely propagated by the computing industry press. Hopefully, this article will help clarify the situation with respect to both issues.

In this article, I will discuss the key methods of distributing data, focusing on two: replication and distributed transactions with 2PC. I will describe the details of implementation by some commercial relational DBMSs, specifically those of Oracle and Sybase. This article is an attempt to help users analyze the strengths and weaknesses of each implementation, with an emphasis on providing guidelines for using one method versus the other.

DISTRIBUTED FOUNDATIONS

Distributed database technology builds on an idea that is central to the relational model: Users (including application programmers and DBAs) need only know about logical constructs and will be protected from changes to the physical implementation. In particular, the key logical construct is a relational table. Its implementation as a physical construct should be hidden from all users. Other physical constructs include the data's location, its physical storage format, and the methods used to access it, such as index creation and selection.

In principle, the DBMS can manage *all* physical constructs and operations automatically based on declarative (logical) instructions. The exception to this stricture against referencing physical constructs is defining the physical resources available to the DBMS. The frequent mixing of logical and physical constructs in current implementations (via the particular SQL dialect) is an unfortunate, disabling violation of the relational model. Strict separation of logical and physical constructs in a relational DBMS and its applications permits the implementation of powerful features, including tranparent management of distributed data.

In a distributed database, several methods of managing distributed data exist. Among the methods are fragmentation, replication, snapshots, and distributed transactions. A table *fragment* is exactly what it says: a fragment of a table. Note that the fragment is itself a table in keeping with the concept of relational closure. (Throughout this article, I will not distinguish among tables that happen to be "base tables" versus those that are *derived tables* such as views and snapshots, since this differentiation should not be relevant to users.)

A better name for a fragment might be a partition; a fragment can either be a horizontal or a vertical partition of a table, not neces-

ARTWORK BY: JIM SHINNICK

sarily disjointed and each stored at a particular physical location (incidentally, equipartitioning a table—dividing it into some number of disjointed tables of equal size—is an important operation that is missing from current relational implementations). A DBMS that supports fragmentation provides for the physical placement of table fragments independent of and transparent to the table's logical identity. Ideally, even the DBA need not know how a table is fragmented; the division of a table into separate physical fragments and its distribution can be largely automatic and based on access patterns.

A *replicate* is a set of distinct physical copies of a table that is automatically kept in synchrony by the distributed DBMS, regardless of physical location. The process of creating and maintaining replicates is naturally called *replication*. A number of methods exist by which a DBMS can maintain the synchrony of a set of replicates. Properly speaking, an update to any replicate is immediately propagated to all other replicates. In reality, a variety of mechanisms have been proposed that relax this requirement, one of which leads to the notion of a copy of a table at a point-in-time, called a *snapshot*.

Obviously, maintaining database consistency while propagating updates among copies of tables requires some notion of a transaction and, in particular, the property called *atomicity*. That is, either the entire update is propagated to all copies, or none of the update is propagated to any copy (depending on the scheme used, the originating update may fail altogether). Replication schemes often ensure atomicity without the user being aware of the transaction boundaries involved in the particular mechanism; updates are propagated using system-initiated transactions that commit, rollback, and may even retry automatically. Depending on the scheme, the transaction boundaries may or may not coincide in time with the boundaries of the update transaction initiated by the user.

MANAGING TRANSACTIONS

In addition to replication and fragmentation support, another important distributed database feature is support for *distributed transactions*. These features are not exactly alternative mechanisms; replication can involve distributed transactions, which can involve replication. Suppose a particular DBMS implements replication, immediately propagating all updates to all replicates. If these replicates are distributed across multiple databases, a transaction that updates a replicated table is a distributed transaction. Distributed transactions are implicit in a variety of other situations. For example, if a local table is updated but a referential integrity constraint involving remote tables exists, an implicit distributed transaction is required once again.

Several methods of managing distributed data exist

As noted earlier, one property of database transactions is atomicity—either every action in the transaction completes successfully or none of them do. In a nondistributed database, atomicity is typically ensured by some form of journaling, which permits the database to be restored to its original state in the event of an error. In a distributed database, each site must have its own journaling mechanism if it is to be autonomous and robust. In order to maintain consistency, each database executes its portion of a distributed transaction in cooperation with all others; if one fails, they must all fail. Any time a user references local and remote tables in a single transaction, either directly or indirectly, the DBMS should automatically enforce transaction atomicity. The user should not have to be concerned with the fact that tables are distributed. Unfortunately, few (if any) commercial attempts to implement a distributed RDBMS have made the location of tables entirely transparent to users, and distributed transaction atomicity is costly to enforce using current methods.

The principal method of ensuring distributed transaction atomicity is *two-phase commit*, meaning that the coordination among databases that participate in a transaction go through two distinct phases in attempting to commit a transaction. (Actually, if many phases exist, the more general concept of a multiphase commit is used. As more phases are involved, the distributed transaction's atomicity is more reliable and more overhead exists.) The two phases do not begin until all requests that make up the transaction have been processed. At this point, the user issues a commit request and the first phase (prepare) begins. The prepare phase determines the ability of each participant to commit its portion of the transaction, which we refer to as a *subtransaction*. The second phase (commit) informs all participants either to go ahead and commit (if all participants were prepared to commit during the prepare phase) or rollback (if even one participant was not prepared to commit during the prepare phase).

A 2PC protocol involves a designated participant, called the *coordinator*, to coordinate the decision during the commit phase. While 2PC protocol can be made robust to failures of most participants, special problems occur if the coordinator fails after sending a "commit" decision to some but not all participants. In particular, some method of coordinating the recovery of all participants after a failure of the coordinator must be implemented. This aspect of the 2PC mechanism differs most radically among commercial implementations, and greatly determines relative efficiency and ease of distributed database administration.

2PC: PLUSES AND MINUSES

The implementation of 2PC protocols is, ideally, robust, efficient, flexible, and fully transparent. In practice, however, each commercial implementation has strengths and weaknesses that determine whether it is adequate to a particular organization's needs.

Automatic 2PC support reduces the amount of application code the user must write, which is certainly true if the assumptions in the design of the automatic 2PC implementation are compatible with the user's needs, but may not be otherwise. In particular, should the coordinator request that a participant retry if an error is reported by that participant during the prepare phase? If so, for what class of errors? Alternatively, should all errors be considered catastrophic, thereby forcing the entire distributed transaction to be rolled back? (Note that many of these issues are important whether the transaction is distributed or not.)

Another way in which a particular implementation of automatic 2PC may not work well is in the context of more complex applications. Keeping in mind that a database transaction is intended to transform the database from one consistent state to another, the question as to what constitutes a consistent state, as well as how strictly this consistency is to be enforced must be addressed. For example, it is not difficult to find business rules that are "conditional"; a particular integrity rule is applied, and, if it fails, an alternative rule is put in effect.

Similarly, business rules are sometimes expected to be enforced within some period of time, but not necessarily immediately. For example, one generally cannot sell what one does not own. However, in stock trading the concept of "selling short" defers the point in time at which ownership must exist even though that ownership must eventually be manifest. Such complex business rules are difficult to characterize with today's RDBMS implementations and, unfortunately, require application code with complex logic and robust error handling.

Another class of circumstances that is not generally handled by transparent 2PC implementations involves application requirements for concurrent transactions and parallelism. These situations require support for generalized nested transactions (that is, transactions that contain other transactions and can be concurrently and independently executed, committed, or rolled back). It is easy to show that not all concurrent transactions can be replaced by a single flat transaction, even with emulation of nested transactions by savepoints. In theory, if nested transactions were supported by an RDBMS, they could be distributed transactions with transparent 2PC. Without this support, such complex transactions require special application logic and commit processing.

2PC is usually an inefficient process involving many costs

Distributed transaction implementations (regardless of whether transparent 2PC is supported or not) should offer the user control over transaction isolation levels or degree of consistency enforcement. When evaluating an implementation, the user should be careful to make certain that the required degree of consistency enforcement has not been forfeited. This information can be difficult to ascertain since it depends on the locking mechanism's technical details, deadlock detection and recovery, timeout mechanisms, potential failure modes, and methods of manual or automatic recovery (none of which the vendor may be willing to disclose).

In applications in which a high degree of concurrency is required, a lower degree of consistency enforcement permitting certain update anomalies may be desirable. This approach is acceptable as long as the update anomalies that would cause a loss of consistency cannot occur given the transaction mix. Furthermore, control over the degree of consistency may be required, and this degree may differ from subtransaction to subtransaction.

2PC is usually an inefficient process involving delay, message, and write costs. Each participant must first receive its portion of the work and prepare to commit. It must then inform the 2PC coordinator that it is ready. If the commit coordinator determines that all participants are ready to commit, it instructs them to go ahead and commit. If any participant informs the commit coordinator that it is not prepared to commit, the commit coordinator must inform all participants to abort. Therefore, the transaction cannot proceed any faster than the slowest participant. The slowest participant must be slower than if it had operated alone because each participant must communicate with the commit coordinator.

2PC IMPROVEMENTS

For this reason, a number of optimizations that improve 2PC performance have been developed, three of which I will mention here. The *read-only commit optimization* recognizes that read-only subtransactions need not participate in callback portions of the prepare, commit, or abort phases. (Of course, if the all subtransactions are read-only, 2PC is not needed at all.) The *lazy commit optimization* is essentially a distributed group commit, in which messages and disk writes are piggy-backed. The *linear commit optimization* arranges subtransactions in a linear order so that prepare and commit propagate down and back up a chain of participants.

The 2PC protocol for guaranteeing atomicity of distributed transactions has a number of variations to handle catastrophic failures. To begin with, the methods and degree of recovery depend on whether the failure occurs during the prepare or the commit phase. For example, the commit coordinator can fail, leaving in-doubt transactions (sometimes called limbo transactions). Typically, either a polling or a time-out mechanism is used to determine whether or not the participant is still "alive" and in communication. On the one hand, a time-out mechanism cannot distinguish between a busy and a "dead" participant. On the other hand, a polling mechanism is expensive. One method of handling failures is called a "presume" protocol of which two basic types exist: *presume abort* and *presume commit*. With presume abort, if a participant requests information about the state of a transaction from the coordinator (typically during recovery), it is presumed to have been aborted if no record of the transaction is found. A corresponding definition exists for presume commit.

An alternative approach is to have the participants poll the commit coordinator when communications are reestablished. Of course, this approach can result in resources being held and may prevent access to data. However, it eliminates the need for the DBA to resolve in-doubt transactions manually, which is a tedious process.

ORACLE'S IMPLEMENTATION

The Oracle Version 7.0 Distributed Database Extension facility for 2PC is transparent in virtually all circumstances. If a remote object is referenced within a transaction, two-phase commit is used. All integrity constraints, remote procedures, and triggers are protected by 2PC. However, declarative referential integrity constraints cannot span databases. Distributed referential integrity constraints can be implemented via triggers. However, on errors or system failures, both the parent and the child tables are locked until they are released by their respective local DBAs or until the transaction is successfully completed.

The originator of a distributed transaction is known as the *global coordinator*. Any instance (Oracle's database unit of start/stop) that must reference other databases is known as the *local coordinator*. One coordinator is designated as a *commit point site*; it is used to determine the outcome of a 2PC after the PREPARE phase. Ideally, the commit point site will be the instance that stores the most critical data for the transaction. In practice, the commit point site is that instance having the highest *commit point strength*, a factor the DBA assigns to an instance at startup. The factor cannot be changed dynamically and it is not adjusted automatically.

Read-only subtransactions do not participate in the COMMIT portion of a 2PC, a partial implementation of the *read-only commit optimization*. The read-only condition is detected dynamically; users do not have to declare a read-only transaction. The state of distributed transactions is maintained in a "pending table." This table is used by the Oracle background recovery process to recover in-doubt transactions, or by the DBA to identify and recover them manually. In-doubt transactions hold exclusive locks until the state of the transactions are resolved, although the local DBA can force them to be released.

No concept of distributed statement atomicity exists

Transactions can be annotated with a comment at commit time (which is distinct from the COMMIT phase of the 2PC). These comments are useful in identifying transactions during manual recovery operations. In addition, certain special comments can be used to force a failure at a selected point in the 2PC process. These comments are useful for testing a distributed database configuration.

Several difficulties introduced by distributed transactions must be carefully managed. Between the PREPARE and COMMIT phases of a 2PC, queries cannot access locked data. For consistency, these locks are guaranteed to survive an instance failure. Unfortunately, a failed distributed transaction may hold locks indefinitely until access to the coordinator and commit point site has been reestablished for all participants. It is up to the local DBA to free them up. Similarly, a database link that is involved in an in-doubt transaction cannot be dropped; unfortunately, no way exists to discover which links are involved in such transactions.

A time-out is used to avoid distributed deadlock conditions rather than distributed deadlock detection and recovery. Unfortunately, a time-out appropriate to the avoidance of deadlocks may be too short for long-running, distributed queries. Any error condition in a distributed transaction, including deadlock, requires that the entire transaction be rolled back. Of more practical concern, no concept of distributed statement atomicity exists, only transaction atomicity. This approach assumes that statement failures (due to violations of resource limit, authorization, constraints, and so on) must be detectable by the global coordinator and that the amount of completed work that must be aborted and redone is not extreme.

SYBASE'S IMPLEMENTATION

In the current release, and in System 10, the Sybase implementation of 2PC for distributed transactions accessing multiple SQL Servers is programmatic. A set of routines are supplied as a part of the Open Client API and are used to obtain a *commit service* (distributed transaction logging and recovery) from an SQL Server, obtain a distributed transaction identifier, send subtransactions (not distributed queries) to SQL Server participants, and then step through the prepare and commit phases with each participant. The current implementation relies on the application for this logic. In effect, the application and the commit service form the coordinator.

If any participant fails prior to the commit phase, the application code requests a rollback from each participant. If it fails during the commit phase, the appropriate recovery is automatic. The failed participant will automatically interrogate the commit service during its recovery and perform the appropriate commit or abort automatically. If the commit service fails prior to the commit phase, the application code must acquire a new commit service and start over. If the coordinator fails during the commit phase, no solution handles all the possible failure modes automatically, and manual or user-written programmatic intervention is required to restore the participants to a consistent state.

While I believe that programmatic 2PC should generally be second choice compared to automatic 2PC, circumstances exist in which the degree of flexibility it offers for error recovery and transaction management outweigh the costs of developing and maintaining code. Of course, this approach assumes the user has the good sense to develop a library of general-purpose, 2PC service functions that ensure a uniform response, and prevent programmers from rewriting this code for each distributed transaction.

It is a little-known fact that SQL Server implements automatic 2PC (and distributed queries, joins, and so on). Unfortunately, SQL

Server restricts its use to multiple databases managed by a single SQL Server. This restriction makes the functionality of little or no use for physically distributed databases.

REPLICATION

Propagation of updates from a primary to a set of replicas can be characterized, for lack of better terms, as either transactional or nontransactional. By transactional, we mean that changes that are propagated as a unit all correspond to some transaction. By contrast, nontransactional *replication* propagates updates without respect for the original transaction boundaries, typically as soon as each individual row is updated or based on the current state of an individual table at some point in time. Nontransactional replication can introduce certain kinds of integrity problems, especially with respect to recovery from errors. In general, some notion of global time must be maintained within the entire distributed system to avoid these errors.

Typical replication mechanisms include utility-based (which may or may not be a separate server process), trigger-based, or programmatic. Declarative definition of replicates can use either a utility-based or trigger-based mechanism. A procedural definition can, of course, use any mechanism. Replication can be real-time, time-based, or store-and-forward. Real-time replication generally uses 2PC to ensure that all replicas are updated synchronously. Time-based replication normally uses some sort of utility and is often used for snapshot support. Store-and-forward techniques are used to handle network or site failures.

In extreme cases, the requirement for synchrony is relaxed to the degree that synchrony is required only at a particular point-in-time or perhaps periodically. In this case, it is important to distinguish between the data's primary copy and all other copies, called *snapshots*.

In the simplest form, both replicates and snapshots are copies of entire tables. However, it is also possible for replicates and snapshots to be table fragments. Between the extremes of table copies

Both replicates and snapshots are copies of entire tables

that meet the formal definition of replicates and those that are deferred snapshots, a range of possibilities exists. For example, assuming it is still possible to guarantee consistency, the propagation of updates can be deferred in time. The guarantee of consistency should be automatic, which raises the question of what constitutes consistency and what does not.

A DBMS that enforces absolute consistency is said to enforce the *serializability* of transactions; regardless of the mix of concurrent transactions, the result is guaranteed to be as though some particular serial (that is, sequentially in time) execution of those transactions had been run. Of course, DBMSs frequently offer enforcement of lesser degrees of consistency. In particular, they permit inconsistent results when certain types of transaction mixes are run. It is then the job of the DBA to ensure that these particular mixes of database transactions do not in fact occur and thereby avoid loss of database integrity.

REPLICATION IN ORACLE

The Oracle Version 7.0 (Oracle7) Distributed Database Extension facility for replication includes several variations, which Oracle refers to as either replicas or snapshots. The mechanisms used depend on whether you are copying data from a primary, or identifying specific rows to be copied vis-a-vis a *snapshot log*. Snapshot logs contain the ROWIDs of changed rows in the primary replica (Oracle refers to a primary as a master table) along with a timestamp. The snapshot log is maintained via an "after row" trigger.

Oracle defines a snapshot as a copy of a table at a point-in-time. Snapshots are defined using a SELECT statement and are classified as simple or complex. A simple snapshot, in contrast to a complex snapshot, has no GROUP BY, CONNECT BY, join, subquery, or set operation in its defining SELECT. Simple snapshots can be refreshed from a snapshot log; complex snapshots are refreshed directly from the primary table and require a complete refresh of the entire table. Snapshots can also be either synchronous or asynchronous. Asynchronous snapshots are read-only.

Two ways to refresh a snapshot table from a snapshot log exist. In one technique, a refresh utility is used to read the snapshot log and refresh the snapshot on a refresh interval. The second technique is to force the refresh manually using Oracle-supplied stored procedures. Refresh using a snapshot log is called fast refresh. Multiple snapshots can use the same snapshot log. While updates to the snapshot log are transactional, the actual refresh is not.

Oracle reserves the term *replicas* to refer to synchronous snapshots. Replicas are implemented by user-written triggers and may be either read-only or read-write. Updates to any replica are intended to be propagated by triggers to all other replicas. Each replica must have two triggers defined on it (one for update and insert operations and one for delete operations), and a special flag column. A projection view is defined on each replica to prevent users from seeing the flag column. The flag column is used by the implementor-defined trigger code to prevent endless cascades among replicas. To add a new replica, the triggers in all other replicas must be modified manually. In contrast with asynchronous snapshots, note that the updates propagated to replicas are protected by 2PC and are transactional.

Several points should be made regarding Oracle snapshots. First, a table with a self-referential constraint cannot be automatically refreshed. Second, neither a replica nor the snapshot log sees deletes of the primary by the TRUNCATE command (a TRUNCATE command does not cause triggers to fire). Third, dropping a primary leaves the snapshot tables intact.

SYBASE REPLICATION

It is possible to use a user-written trigger mechanism to implement replica and snapshot updates in SQL Server. This approach is similar to the user-written trigger mechanism Oracle offers (using TRUNCATE or dropping a primary has the same effect), but SQL Server uses a different mechanism to prevent endless cascades. Unfortunately, the remote procedures you can use to update a remote replica are not a part of the transaction in which the triggering update occurs. If a loss of integrity can result in the given application, the developer must provide both a means of ensuring that the updates are not visible unless the triggering transaction is committed, and that a compensating transaction is run if the triggering transaction is aborted. Such a mechanism is roughly equivalent in complexity to using programmatic 2PC.

The Sybase solution to this complexity, Replication Server, is expected to be in beta release in the early second quarter of 1993. The goal of Sybase Replication Server is to move data among SQL Servers, thereby making the overhead of distributed transactions within an application unnecessary since all accessed data can be local. The mechanism is server-based, partially in an effort to minimize interference by the replication process with local operations.

With Replication Server, SQL Servers "subscribe" to a primary copy maintained by another, typically remote, SQL Server. The declarative definition of a subscription is similar to that used for a view definition (an SQL SELECT is used to define which table fragment is to be replicated) and a similar authorization mechanism is used (the definer must have the appropriate permissions for the data at the remote SQL Server). The primary does not need to be constrained to Sybase data. The concept of "virtual" tables and columns is used to provide a pseudo-relational view of nonrelational data. In general, the developer must write a set of routines for the log transfer manager that provides this view, although Sybase has stated an intent to build the log transfer manager for some foreign data sources such as DB2.

Any changes to the secondary replicas are automatically propagated back to the primary. The definer can decide the order in which updates are to be propagated—whether to update the primary first and then propagate to copies, or to the local copy first and then the primary. Replication Server is designed so that subscriptions can specify that timestamps are to be used for point-in-time snapshots or for automatic conflict resolution, or can prevent conflicts by requesting 2PC on updates; the latter is not likely to be available in the product's first release.

2PC and replication have costs as well as benefits

The data changes corresponding to a transaction are moved among SQL Servers, rather than copies of the data or the statements (for example, SQL) that would make the changes. The Replication Server scans the after-image log at a remote SQL Server, detecting appropriate committed update transactions, and sending these portions of the after-image log to subscribers, where they are used to "rollforward" the replicas.

Users of Replication Server should be aware of potential problems. For example, in the event of a network or SQL Server failure, Replication Server automatically propagates updates to replicas as soon as the remote SQL Server is accessible. Care must be taken to ensure that no conflicts result from updates that take place to isolated subnetworks while replicas are "disconnected." Also, the required time to propagate replicas is on the order of 10 seconds so that not all replicas will be identical simultaneously unless 2PC is requested. While a delay of this magnitude will often be unimportant, users must determine whether or not it constitutes an integrity exposure for their application.

Replication Server error handling differs from that of Oracle7 since it does not make replica update a part of the original transaction. If an update causes an error at some SQL Server, the originating SQL Server gets an entry in a special table that Sybase refers to as a "queue" table. Entries contain a transaction identifier, the reason the update was rejected, and optionally the time or other identifying information. These entries are intended to be handled by either a user-written utility or by user-written triggers on the queue table (again, this approach provides maximum flexibility at the expense of some additional development and maintenance).

FALSE COMPARISONS

The industry's trade journals' discussion regarding the merits of automatic 2PC versus replication is based on a serious misunderstanding of these database features. The comparison is inappropriate. As we have seen, both have their appropriate uses, benefits, and costs.

Any particular implementation of two-phase commit (whether automatic or programmatic) or replication has its costs and benefits, and strengths and weaknesses. For example, neither Oracle nor Sybase uses replicas to optimize performance automatically. Also, neither can ensure that updates to two or more replicas will not conflict. In principle, the mechanisms used could result in distributed deadlocks and errors, as well as locks being held by in-doubt transactions. Although we have looked at the implementations of Sybase and Oracle, these are not the only companies with RDBMS products that provide some support for two-phase commit and replication. The reader is encouraged to examine the implementations of such products as DEC's Rdb/VMS, Tandem's Non-Stop SQL, and Cincom's Supra Server in the light of this article.

The difficulties regarding how to advise users on when to use replication and when to use two-phase commit remain. I will conclude with a few tips that address this issue:

☐ Do not arbitrarily mix distributed transactions and replication. This approach can lead to integrity problems since updates in a distributed transaction are propagated immediately and the replication mechanism may involve a delay across sites.

☐ Use distributed transactions protected by two-phase commit if the potential for data integrity loss is not low given the transaction mix.

☐ If relationships must be maintained "to the second" and you can tolerate the corresponding loss in throughput, use distributed transactions.

☐ If the likelihood of data integrity loss is low and conflicts can be resolved, use replication.

☐ Consider using replication if concurrency needs outweigh data integrity requirements. In particular, if few integrity constraints exist between the rows and tables that would be involved in transactions accessing the replicated data, replication may work fine.

☐ If you cannot define a single path of update propagation, given a particular state of the network, make certain that the replication mechanism will not introduce update sequence errors before using it. This step is especially important if a site can receive the same replica update many times. Otherwise, you should use distributed transactions.

☐ If replication is utility-based, make certain the utility is robust. Weigh the possible costs and benefits (points of failure, interference, performance, and administration) before using replication.

☐ If possible, ensure that the entire mix of transactions that access replicas form a commuting set (in other words, order of execution should not change the final database state), and that each such transaction has a compensating transaction.

Finally, examine any feature offered by a database vendor carefully. Those consumers who fail to heed this advice or fail to question the vendor's use of terminology and their depth of understanding of the requirements are certain to be (unpleasantly) surprised. Caveat lector—let the reader beware. ■

REFERENCES

1. McGoveran, D., with C. J. Date. *A Guide to Sybase and SQL Server*, Addison-Wesley Publishing Co., 1992. The example of programmatic 2PC that appeared in Oracle's advertisement may be found on p. 496, Figure 27.1.

2. Ceri, S., and G. Pelagatti. *Distributed Databases: Principles and Systems*, McGraw-Hill Inc., 1984.

3. Date, C. J. *An Introduction to Database Systems: Volume 1*, Fifth Edition, Addison-Wesley Publishing Co., 1990.

4. Date, C. J. *An Introduction to Database Systems: Volume 2*, Addison-Wesley Publishing Co., 1985.

5. Date, C. J. *Relational Database Writings, 1986-1989*, Addison-Wesley Publishing Co., 1990.

6. Gray, J., and A. Reuter. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, 1993.

7. McGoveran, D. "Oracle 7 Evaluation Report: Database Product Evaluation Report Series," Alternative Technologies, Boulder Creek, CA, 1993.

8. McGoveran, D. "Sybase Evaluation Report: Database Product Evaluation Report Series," Alternative Technologies, Boulder Creek, CA, 1993.

David McGoveran is president of Alternative Technologies (Boulder Creek, California), a relational database consulting firm founded in 1976. He has authored numerous technical articles that have appeared in *Database Programming & Design* and other leading industry journals. He is also the publisher of the "Database Product Evaluation Report Series."

Fortunately, there's only *one* direction for Client/Server *training*. . .BENCHMARK

BENCHMARK's courses on server and application development products are available from coast-to-coast. Join the major corporations that rely on training from BENCHMARK.

Call 1-800-347-4661 for a *FREE* Education Catalog

BENCHMARK
TECHNICAL SERVICES, INC.
386 Main Street, Middletown, CT 06457 (203) 346-4661

SQL-Server is a registered trademark of Microsoft Corp. and SYBASE, Inc. ORACLE is a registered trademark of Oracle Corp. DB2 is a registered trademark of the IBM Corp. XDB is a registered trademark of XDB Systems, Inc. Micro Focus Workbench is a registered trademark of Micro Focus, Inc.

CIRCLE 15 ON READER SERVICE CARD