the Availability Digest

Transaction-Oriented Computing

April 2007

Transaction-oriented computing brings with it a multitude of advantages. Of interest to us is its support of high availability through fast recovery. However, transaction-oriented computing's original task was to ensure the consistency and durability of the application databases. As transactional methodologies evolved, significantly improved performance was added to the list of its advantages as was its support for data replication.

It is for all of these reasons that transaction-oriented computing should be understood by anyone interested in achieving high availability of their data processing services. For many involved in today's computing systems, this may be like preaching to the choir since transactional computing forms the basis for most present-day business applications.

However, the requirement for high availability extends far beyond just business applications to areas in which transaction processing is not commonly used and is not well understood. This includes not only applications that are not business-oriented but also to older business applications that were developed prior to the advent of transactional computing.[1] It is to these audiences that this article is addressed.

What Does Transaction Processing Have To Do With Availability?

Transaction-oriented computing is the foundation of today's recovery mechanisms. As the cost of downtime grows with each new 24x7 application, data center operations are moving from tape backup to maintaining an up-to-date database at a remote site via data replication. Should the primary site fail, restoration of services no longer depends upon fetching the backup tapes, reading the last full backup, and updating it with incremental backups, a process that can take hours or days. Even worse, all of the transactional activity since the last backup is lost – a period of perhaps up to a day's worth or more of valuable data.

The new recovery paradigm sends updates as soon as they happen to a remote site where a remote copy of the database is maintained. These updates immediately update the database so that it is always a consistent copy of the primary database, except for a small (perhaps seconds) delay.

This technique of replicating data from a primary system to a backup system supports cold backups (backup applications must be started), hot backups (backup applications are up and running), and active/active systems (transactions are being processed by all systems). Recovery can be accomplished in seconds to minutes, depending upon the application network configuration, with very little if any loss of data.[2]

---

[1] See Virtual Transactions with NonStop AutoTMF in this issue of the *Availability Digest*.

[2] See our previous articles, Asynchronous Replication Engines (November, 2006), Synchronous Replication (December, 2006), and What is Active/Active (October, 2006).

This is where transaction processing plays such an important role. Data replication depends upon the existence of a change log. The most common form of a change log is the transaction log maintained by a Transaction Manager in a transaction-oriented data processing system.

The Attributes of a Transaction

What is this thing called a transaction? A transaction is a collection of operations on the state of an application. All operations must be executed in order for the application state to remain consistent. If any one of the operations within the scope of a transaction cannot be successfully executed, none of the operations can be executed.

A common example is a consumer banking application. A bank's customer wants to move $500 from his savings account to his checking account. This transaction requires two operations – debit his savings account by $500 and credit his checking account with $500. If only one of these operations is executed, the result is wrong. Either the customer is out $500, or he has an additional $500. The application state is only correct (that is, consistent) if both operations are performed.

In a program, a transaction is identified by preceding all of the operations pertinent to that transaction with a *begin transaction* statement of some sort and by following the set of operations with an *end transaction* statement. A transaction has four important attributes, known collectively as its ACID properties – atomicity, consistency, isolation, and durability.

- *Atomicity* requires that either all of the operations within a transaction are executed or that none are. A transaction is atomic if it appears to a viewer that the application jumps from one correct state to another correct state with no intermediate states.

- *Consistency* means that the results of a transaction satisfy the specified constraints on the application. If a transaction cannot guarantee a consistent state, it is aborted.

- *Isolation* means that the result of a transaction is unaffected by other transactions that may be executing at the same time.

- *Durability* means that the results of a committed transaction are never lost, even following the restoration of the system after a failure.

Atomicity and Consistency

The consistency of an application is guaranteed by the atomicity property of transactions. Assuming that the program is properly written, the execution of any set of operations within the scope of a transaction will leave the application in a consistent state. In the above example, either $500 is debited from the customer's savings account and credited to his checking account, or no changes to his account are made.

An entity called the *Transaction Manager*, described in more detail later, is responsible for enforcing atomicity. When it receives a begin transaction statement, it will ensure that the information required to return the application's database to a consistent state is retained as modifications are made to the database. Should all operations within the scope of the transaction be successful, it will apply those changes (it will *commit* the transaction). Should any one of the operations fail, the database is returned to its original state as if the transaction never happened (the transaction is *aborted*).

The application is informed of the outcome of the transaction so that it can take the appropriate next step (proceed forward on a commit, perhaps retry on an abort).

Isolation

Isolation is provided by locks. Before an operation within a transaction updates a data item, that operation must acquire a *lock* on that item. This prevents any other application from updating that item until the transaction owning the lock releases it.

A transaction can also acquire a lock on a data item that it intends to simply read to use as part of a computation to update another data item. This prevents the value of the read data item from changing until the transaction is complete.

A transaction holds its locks until it is either committed or aborted. At that time, it will release all of its locks.

This locking strategy also plays a role in consistency. Not only does a lock on a data item prevent another application from updating that data item, but it also prevents other applications from reading that data item (except for "dirty reads," which read through locks and which are allowed by some systems). As the transaction progresses and executes one operation at a time, it will take the application through some inconsistent states. No other application can read this state because of the locks. Only when the application state has reached a consistent state and the transaction has committed or aborted is the modified state exposed. Thus, as mentioned earlier, to an outside viewer the application seems to jump from one consistent state to another with no intermediate states.

A problem with locks is deadlocks. A deadlock occurs when two programs attempt to acquire a lock on two data items out of order. Each holds the lock required by the other program, and neither can proceed.

Deadlocks can be avoided by using an *intelligent locking protocol*, which specifies a mandatory order in which locks should be acquired. If deadlocks cannot be avoided, there are several techniques for resolving them, such as having the programs back off and try again after a random time.

Durability

Durability is the attribute in which we are interested from an availability viewpoint. It means that once the results of a set of actions bounded by a transaction have been committed, the new database state survives any subsequent system failure.

Durability is the responsibility of the Transaction Manager, mentioned above. As a transaction progresses, the Transaction Manager retains a record of the before image and the after image of any record or row that is modified by the transaction in a *transaction log*. Transaction logs are described in more detail later. Before any changes are made to the database itself, the before and after images in the transaction log must have been safely stored on a persistent medium such as disk. This means that before a transaction is committed, that portion of the transaction log that contains the transaction's before and after images must be safely stored. Meanwhile, the actual modified data blocks may sit in cache memory for a while and be written to disk at some later time.

Should the system fail, there will be some committed transactions whose changes have not made it to the physical database (they were still in cache memory) and some incomplete transactions which have had some of their changes made to the database (after writing the before and after

images to the transaction log). When the system is returned to service, the database must be *recovered*. This means that it must be returned to its last known consistent state.

This is the responsibility of the Transaction Manager, which uses the transaction log to accomplish this task. The recovery procedure is described later.

The Transaction Processing Architecture

The first well-known transaction processing system was IMS (Information Management System), which was introduced by IBM in 1968. It was followed by IBM's CICS (Customer Information Control System). TP monitors were later introduced by Tandem (Guardian), Digital (ACMS), and Stratus. Open TP monitors include Tuxedo and Encina. Almost all commercial and open databases today incorporate a Transaction Manager.

The generic structure of the facilities to provide transaction-oriented computing is described below.

The TP Monitor

The Transaction Processing (TP) Monitor provides the application environment for transactional computing. Though there is no commonly accepted definition of a TP Monitor, it typically provides the services to manage the following:

- the user terminals connected to the system and which are the source of the transaction
- the presentation services provided to the user terminals
- the communication services that connect the user terminals to the system and that interconnect multiple systems in a transaction network
- transaction context which must be passed from one transaction to subsequent related transactions
- load balancing in multiprocessor systems
- heterogeneity if the application functions across different databases which may, in fact, be from different vendors.
- Start/restart following a failure to bring all databases to a consistent state.

The Transaction Log

The *transaction log* is the heart of the recovery mechanism for transactional computing. It is a persistent log of the before and after images of any changes made to a row or a record in the database. Before the changes to a row or record are physically written to the database, the transaction log recording those changes must be written to persistent storage. The actual database changes can be written at some time later. Until that time, they exist only in non-persistent cache memory.

The transaction log also records begin, commit, and abort operations associated with each transaction. All transaction log records carry a transaction ID which is used to tie together all of the records pertaining to a single transaction.

To be persistent, the transaction log is usually written to a redundant disk system that is different than the database disk system. This allows it to survive should the database system fail, become corrupted, or be destroyed.

The Log Manager

It is important to maintain the transaction log for a long period of time so that a damaged database can be restored. Therefore, a transaction log can grow without bound.

Managing this growth is the responsibility of the Log Manager. The transaction log is typically implemented as a set of files. When one file fills, the Log Manager closes it. The next file is purged and used for the next set of transaction logs. At this time, the oldest transaction log file is rolled off to tape and made ready to become the next transaction log file.

In some systems, multiple transaction logs may be used to improve performance.

Resource Managers

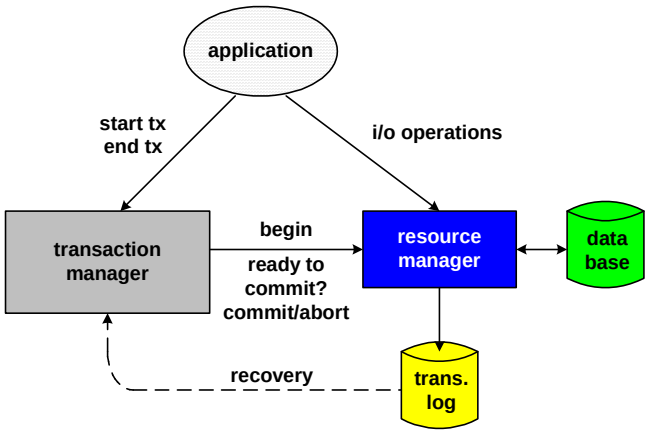
A transaction may span multiple independent databases. The database management system for each database is considered a *Resource Manager*. (There may also be other Resource Managers – any entity that can be affected by a transaction is a Resource Manager.) A database Resource Manager receives begin/end transaction commands from the Transaction Manager, described next, and I/O operations such as reads, writes, and deletes from the application.

The Resource Manager maintains the transaction log for the database's rows or records that are affected by a transaction.

The Transaction Manager

The Transaction Manager provides the "A," "C," and "D" of the ACID properties (that is, the atomicity, consistency, and durability properties). It receives transaction begin and end commands from the application and communicates these to the various Resource Managers in the system. The "I" of ACID is provided by the locking mechanisms.

When the Transaction Manager receives a begin transaction command, it sends that with a transaction ID to all involved Resource Managers. It also returns the transaction ID to the application. Each Resource Manager will optionally write a begin transaction record to the transaction log (some Transaction Managers do not do this and rely instead on the first I/O operation to signal the beginning of a transaction).



As the application issues I/O operations under this transaction, the before and after images for each operation are tagged with the transaction ID before writing them to the transaction log. The application may switch between concurrent transactions by switching transaction IDs, but each operation must be associated with its own transaction.

When the application issues an end transaction command, the Transaction Manager does so with a two-phase commit. During the first phase, it asks all Resource Managers if they are prepared to

commit the transaction. A Resource Manager may not be able to complete the transaction, for instance, if it could not obtain a necessary lock, if it suffered a disk error, or if the changes would violate its referential integrity. If it can commit its part of the transaction, it ensures that its transaction log has been written to disk and replies affirmatively to the prepare-to-commit query. If it cannot commit its part of the transaction, it replies negatively.

The second phase of the protocol depends upon the responses generated by the Resource Managers during the first phase. If all Resource Managers replied affirmatively, the Transaction Manager informs all Resource Managers to commit the transaction. If any of the Resource Managers replied negatively, the Transaction Manager instructs all Resource Managers to abort the transaction. In either event, the Transaction Manager informs the application as to the outcome of the transaction.

Recovery

Should there be a system failure of any sort that would damage the database, the Transaction Manager is responsible for repairing the database. It does this by using the before and after images of the transaction log.

Starting at an appropriate point in the log, the Transaction Manager searches for transactions that have committed and for transactions that have not completed. A transaction may have been committed in the transaction log, but the actual database changes may not have been written to disk if they were still resident in cache memory at the time of the failure. Using the after images in the transaction log, the Transaction Manager ensures that all changes for committed transactions have been made to the physical database.

Conversely, some changes to the physical database may have been made before the transaction could commit or abort. In these cases, the Transaction Manager uses the before images to return the database to its state before the transaction began.

The result of this process is that the database is returned to its last known consistent state.

Another use of the recovery mechanism is to recover accidentally deleted files or tables.

Performance

A side benefit of transactional processing is performance. The transaction log is written serially to disk and is therefore very fast and efficient. Writing the transaction log is much faster than writing random changes to the database. These random writes can be deferred until after the transaction has completed. Therefore, transaction completions are very much faster under a transaction monitor than they are with nontransactional updates and result in faster response times to the users.[3]

In large systems, transaction log flushing is often delayed a few milliseconds to attempt to write to disk as many blocks as possible at one time. A few milliseconds delay may be small compared to the reduction in transaction response time obtained via the use of a log file instead of random updates. This technique also results in a reduced load on the disk subsystem. In fact, with this technique, the busier the system becomes, the more efficient it becomes.

[3] This was not always the case and was one of the impediments to early acceptance of transaction processing. In an early Stratus manual, the user was warned not to use transactions unless absolutely necessary due to performance issues.

Replication

And now the good part. Extreme availabilities are achieved by having a redundant system up and ready to take over processing in an instant's notice. Whether this is done by using a backup system or an active/active configuration, the production database is replicated to a remote system via data replication.

A data replication engine depends upon a queue of changes that it can send to the remote system. The transaction log is an ideal queue of changes for this purpose.

Many data replication engines use the transaction log as the source of changes to be replicated to a remote site.[4]

Distributed Transactions

In many cases, the multiple databases involved in a transaction are not resident on the same system. They may be distributed across many systems using database offerings from many vendors. Transactions will be executed in a highly heterogeneous environment.

To accomplish this, each of the diverse Resource Managers must comply with a specified interface so that the Transaction Manager on any one of the systems can communicate with them. Typically, the Transaction Manager on the system hosting the application which originated the transaction is the *transaction coordinator* for that transaction. Its Transaction Manager communicates with the Resource Managers across the network to begin a transaction.

When the application signals the end of the transaction, the transaction coordinator's Transaction Manager uses the two-phase commit protocol described earlier to complete the transaction. It asks all Resource Managers across the network if they are ready to commit the transaction. If all are ready, the transaction is committed. If one or more Resource Managers cannot commit the transaction, the transaction is aborted.

The standard interface between the Transaction Managers and the Resource Managers is spelled out in the XA specification[5] from The Open Group, an industry consortium. Most commercial and open databases today support the XA standard.

Summary

Transaction-oriented processing is now an established technology that guarantees database consistency and durability. It has progressed to the point that it even has the potential to improve transaction response time.

From an availability viewpoint, the transaction log maintained by the transaction processing system often is used by data replication engines to maintain a remote database in synchronism with the primary active database for fast recovery following a system failure.

---

[4] See Flexible Availability Options with GoldenGate's Transactional Data Management Platform (TDM), *Availability Digest*, February, 2007, and Shadowbase – The Active/Active Solution, *Availability Digest*, March, 2007.

[5] Distributed TP: The XA Specification, The Open Group; 1992 (http://www.opengroup.org/bookstore/catalog/c193.htm).