the Availability Digest

Transaction Processing: Concepts and Techniques

April 2007

Jim Gray is arguably one of the most readable technical authors today. Coauthored with Andreas Reuter, Jim's book, *Transaction Processing: Concepts and Techniques*, is an in-depth (1,070 pages) and easily readable description of transaction-oriented processing. It carries the reader from the basic concepts of transaction processing through a straw man implementation of a Resource Manager to a review of current transaction monitors.

One of the most pleasing aspects of this book is that not only are concepts clearly explained with many examples and without excessive technical jargon, but material presented earlier is reviewed in later chapters when it is once again referenced. For such a complex topic, this book is clearly fireside reading.

Introduction to Transaction Processing Systems

A transaction processing (TP) system is big. It provides the tools to ease or automate application programming, execution and administration. It typically includes application generators, operations tools, one or more database systems, utilities, and networking and operation software. The core services of a transaction processing system are provided by a *transaction monitor*.

A TP system is organized around the concept of a transaction. A transaction is a collection of operations on the application state. A transaction has several important properties, including maintaining the application in a consistent and durable state.

The management of transactions is the responsibility of the *Transaction Manager*. The Transaction Manager coordinates the application of transactions to *Resource Managers*, which hold the data and other objects whose state is to be changed by the transaction.

The bulk of the book is devoted to descriptions of the transaction processing monitor, the Transaction Manager, and Resource Managers. Before immersing the reader in these descriptions, Gray and Reuter devote a chapter to carefully defining a wide range of hardware, software, data, and system terms that they will use in their later material.

Fault Tolerance

The durability property of transactions implies the requirement for a level of fault tolerance in TP systems. Over the years, the satisfaction of this requirement has progressed from tape backup to fully redundant systems. Gray and Reuter report on studies concerning the nature of system failures. The distribution of the causes of failures and failure frequencies are often quoted in the literature.

Hardware and System Software	43%
Communications	12%
Application Software	25%
Operations	9%
Environment	11%

Causes of System Failures

Hardware and System Software	6 months
Communications	2 years
Application Software	8 months
Operations	2 years
Environment	2 years

Mean Time to Failure

The book then describes hardware and software approaches to fault tolerance. Hardware approaches are all based on *failfast* modules which will fail before an error can be propagated. Failfast arbbchitectures include duplexed and triplexed voting systems.

Software fault tolerance is a much more difficult problem than hardware fault tolerance. Approaches to software fault tolerance include process pairs which bring redundancy to the system software.

Transactions bring the ultimate in fault tolerance. If all else fails, they allow the system to crash and restart gracefully in a fully consistent state.

Transaction-Oriented Computing

The remainder of the book is devoted to transaction-oriented computing. As stated earlier, a transaction is a collection of operations on the application state. There are four important properties of a transaction, known as its ACID properties – atomicity, consistency, isolation, and durability:

- *Atomicity* means that either all of the operations in the transaction are executed or that none are. A transaction is atomic if it appears to cause the application to jump from one state to another (or back to the initial state) with no observable intermediate states.

- *Consistency* requires that a new application state fulfills all of the consistency constraints of its specification. A transaction produces only consistent results; otherwise, it aborts.

- *Isolation* means that the processing of a transaction behaves as if the system only had a single user even though several transactions may be executing simultaneously. There is no impact on the results of the transaction by other simultaneously executing transactions except for the case of simultaneous attempts to update the same data item. These attempts will be serialized, and a transaction will await its turn.

- *Durability* requires that the results of a completed transaction are never forgotten. They become part of reality. This means that the system must be able to reestablish the transaction's results after any type of failure.

There exist several models for transactions.

Flat Transactions

A flat transaction is the simplest form of a transaction. It comprises a collection of operations that are bounded by a *begin statement* and an *end statement*. Either all of the operations in the bounded collection are executed (*committed)* or none are (*aborted)*. A transaction may be aborted either by the program executing it or because of an external failure, such as a system crash.

Flat Transactions with Savepoints

There are cases in which flat transactions are too simple a model to be effective. Consider a travel agent attempting to book a complex itinerary. She may have booked airlines and hotels for several legs but then finds that a booking attempt for the next leg fails. She either must abort the entire transaction and try again or put in a compensating transaction to back out the failed operations. Neither is desirable.

It would be much better if she could simply return the current transaction to some previous transaction state that is consistent. This is what savepoints are all about. A *save work* command saves the current state of the transaction. At any later time, the application can roll back the transaction to any of the transaction savepoints that have been established.

Chained Transactions

Chained transactions are a variant of savepoints. However, rather than simply marking a point of consistency to which the transaction can return, the *chain work* command actually commits the work so far. Therefore, there is no opportunity to roll back any of the previous work.

However, the transaction itself survives. Any locks held by the transaction continue to be held as further work is done. Only when the full transaction is committed are its locks released.

Nested Transactions

A nested transaction comprises a tree of transactions. The root transaction can spawn subtransactions which themselves can spawn additional subtransactions. The leaves of the transaction tree are flat transactions.

A parent transaction can pass its locks to a child subtransaction. A subtransaction can commit or abort at any time. In this case, any locks owned by the subtransaction are counter-inherited by the parent. Only when the root transaction commits are all locks released.

Distributed Transactions

Distributed transactions are those that must execute across a network of databases. They are similar to nested transactions. However, the transaction tree for a nested transaction is application-dependent, whereas the tree for a distributed transaction is data-dependent.

Distributed transactions are basically flat transactions. However, if data held by a remote database must be updated, a subtransaction is started on that database. The scope of the subtransaction is the set of operations on the database. At commit time, the parent transaction queries all of its subtransactions to ensure that all of them are prepared to commit before issuing a commit command. If one or more subtransactions cannot commit, the transaction is aborted.

Today's distributed transaction standard for heterogeneous systems is XA from The Open Group (http://www.opengroup.org/bookstore/catalog/c193.htm).

Long-Lived Transactions

Batch transactions are an example of long-lived transactions which can contain millions of updates and last for hours. This can be an intolerable situation.

One solution is to break the batch job into mini-batches operating on data with a common attribute, such as a range of keys. This is not a perfect solution since the atomicity of the entire transaction cannot be maintained.

TP Monitors

There is no commonly accepted definition of what a transaction processing monitor is. Several have evolved over the years to meet specific needs. However, a common thread is that TP monitors control *Resource Managers*. A Resource Manager is a subsystem, such as a database, that provides protected actions on its state to maintain its internal ACID properties.

The common functions provided by a TP monitor include:

- *Manage Heterogeneity* – If the application function requires access to heterogeneous databases such as those from different vendors, each database manages its own transactions. The databases are structured to act as *Resource Managers* that can be coordinated by a remote *transaction coordinator*. The TP monitor must provide the facilities to coordinate these Resource Managers so that the ACID properties of the parent transaction can be ensured.

- *Control Communication* – Communication resources are managed by a Communication Manager. The Communication Manager informs the Transaction Manager of the begin or end of any distributed transaction, whether initiated locally or remotely. To the extent that the Communication Manager is maintaining sessions with and handing messages to the Resource Managers, it is also a Resource Manager.

- *Manage Terminals* – Transactions are typically initiated by users at terminals, and therefore the terminal must be considered part of the transaction. Thus, the TP monitor must be responsible for terminal management. In particular, it must deal with the problem of whether a message was actually delivered to the terminal.

- *Presentation Services* – If the terminal uses sophisticated presentation services, the TP monitor must be responsible for recreating the terminal state following a terminal crash. This includes reestablishing the windows environment, the cursor position, and so forth. In this case, the terminal client software also becomes a Resource Manager.

- *Manage Context* – If context is to be carried between transactions, the TP monitor must maintain this context and make it available to the next transaction.

- *Start/Restart* – Following a failure, the TP monitor must bring up all of the transaction processing components in a state that maintains the ACID properties of all committed transactions.

The authors describe in great detail the structure of a typical TP monitor and provide a strawman implementation.

Concurrency Control

Transaction isolation (the "I" in ACID) is variously called consistency (the static property), concurrency control (the problem), serializability (the theory), and locking (the implementation).

There are two laws of concurrency control:

(1) Concurrent execution should not cause application programs to malfunction.

(2) Concurrent execution should not have lower throughput or much higher response times than serial execution.

Isolation is violated if:

- *Lost update* – Transaction 2 ignores Transaction 1's update and overwrites it.
- *Dirty read* – An inconsistent read through a lock is allowed.
- *Unrepeatable read* – An item read by Transaction 1 is updated by Transaction 2 before Transaction 1 has completed.

Isolation can be defined as a process in which transactions can execute in parallel and:

- a transaction does not overwrite data that has been locked and updated by another transaction.
- transaction writes are neither read nor overwritten by other transactions until that transaction commits.
- a transaction does not read updated and locked data (dirty data) of another transaction.
- Other transactions do not write any data read by another transaction until that transaction completes.

The authors introduce a transactional syntax and proceed to prove several transactional theorems.

Isolation is implemented via locks. Several locking protocols beyond simple semaphores are described.

Predicate Locks

One problem is represented by a SQL statement that reads with lock a set of rows. The locks do not prevent another transaction from inserting another row into this sequence, thus making the read sequence nonrepeatable.

Predicate locks lock a subset of the database defined by a predicate for the particular transaction so that insertions cannot occur. However, predicate locks can cause performance problems because of the time required to compute the predicate so that the particular rows can be locked.

Granular Locks

Granular locks are similar to predicate locks except that the predicate is precomputed according to some application-dependent algorithm. This improves the performance of predicate locks because predicates do not have to be computed on the fly. Several types of granular locks are defined.

Nested Transaction Locks

Locks for nested sequential transactions form a lock tree similar to the transaction tree. Nested locks for parallel nested transactions is a nontrivial problem. Conceptual ideas are discussed.

Deadlocks

Deadlocks can occur when two transactions attempt to lock a pair of data items in opposite order. Deadlock resolution can be implemented by having both transactions abort and retry, or by having them time out. Deadlocks can be avoided by specifying a linear locking order that all transactions follow.

Lock Implementation

Actual implementation algorithms and accompanying code are presented to show the difference in performance of different locking methodologies.

Recovery

A major function of a TP monitor is to recover the system to a consistent state following a failure. This is the responsibility of the *Transaction Manager*. The Transaction Manager depends upon a transaction log to perform this function.

Log Manager

The transaction log is the heart of the recovery process. It is maintained by the Log Manager.

Recorded in the transaction log are the before state and the after state of any resource (typically, a data item in a file or database) affected by a transaction. The transaction log grows without bound over time and can become quite large in a short period of time. Therefore, it must be periodically rolled off of the system onto secondary storage.

The Log Manager is responsible for managing the archiving of the log. Typically, there will be several rotating log files. When one fills, it is closed; and the oldest file is purged and used for the next transaction log. As a file becomes the oldest file, it is rolled off to secondary storage, typically magnetic tape. The magnetic tape silos may be collocated with the system or may be located at a remote site for disaster-recovery purposes.

The Log Manager also provides a public interface to the logs for the Transaction Manager and the Resource Managers. This is used primarily for recovery as the transaction log provides all information necessary to return a damaged database to its last consistent state.

A major performance advantage of transaction processing systems is provided by the transaction log. Rather than having to randomly write all updates within the scope of a transaction to disk before the transaction can be committed, it is only necessary to flush the transaction log to disk. This is a serial write that is very fast. In addition, in very busy systems, the Log Manager can wait until there are several blocks to write and still only penalize the transaction with a few milliseconds of delay. This makes logging even more efficient. In fact, the busier the system gets, the more efficient logging becomes. As a result, transaction-oriented systems typically show much better performance than nontransactional systems.

Some transaction logs use physical logging in which actual before and after images are logged. For large records or rows with minor changes, log blocks can be compressed. An alternate technique is to log the operations, such as SQL statements (logical logging).

Log files are often redundant. They use RAID arrays or are mirrored on separate disk units to ensure that the system can be recovered following a failure.

Transaction Manager

The Transaction Manager is responsible for creating the information needed to recover the system following a failure. It receives begin, commit, and abort commands from the application and passes these to the appropriate Resource Managers. The Resource Managers will write these to the log along with the before and after images of data items that they modify. In effect, the Transaction Manager is responsible for the “A” (atomicity), “C” (consistency), and “D” (durability) ACID properties of the transactions. Locks provides the “I” (isolation) of ACID.

For large systems, there may be multiple transaction logs to improve performance. The Transaction Manager must coordinate the transactions written to and read from these logs.

Using the transaction log, the Transaction Manager has several responsibilities:

- *transaction rollback* – Should any participant in a transaction abort the transaction, state changes made so far are undone by applying the before images from the transaction log.

- *Resource Manager restart* – If a Resource Manager fails, the Transaction Manager restarts it and presents it with the log records necessary to bring it to its last consistent state. Completed transactions are rolled forward via the after images, and incomplete transactions are rolled back via the before images.

- *system restart* – The Transaction Manager restarts all Resource Managers as described above. In addition, it resolves any transactions which were left in doubt (that is, it is not known whether or not the Resource Manager received the last commit or abort).

- *media recovery* – Should a disk fail, the transaction log is used to reconstruct the data held by that disk. This function can also be used to recover corrupted or accidentally deleted files or tables.

The authors then describe in detail the Transaction Manager structure with sample implementation code.

Several advanced topics are discussed:

- Heterogeneous transactions, which include within their scope systems that use different databases or are from different vendors.

- Highly available commit coordinators using redundant Log Managers and Transaction Managers.

- Transfer of the commit coordinator role to a more reliable or higher performance node.

- Disaster recovery at a remote site by replicating the transaction file to the remote system in real time.

A Sample Resource Manager

A sample Resource Manager is designed in detail (three chapters). This design embodies the concepts presented previously in the book.

System Surveys

Finally, several transaction processing systems are reviewed. These include IBM's IMS and CICS, HP's NonStop Guardian and ACMS, X/Open DTP, UTM, ADABAS, Encina, and Tuxedo.

Postscript

Jim Gray was reported missing at sea during a sailing trip off the California coast on January 28, 2007. His contributions to transaction-oriented computing are fundamental to the implementation of active/active systems, a primary focus of the Availability Digest.