TRANSACTION PROCESSING

A business transaction is an interaction in the real world, usually between an enterprise and a person, where something, such as money, products, or information, is exchanged (1). It is often called a computer-based transaction, or simply a transaction, when some or the whole of the work is done by computers. Similar to the traditional computer programs, a transaction program includes functions of input and output and routines for performing requested work. A transaction can be issued interactively by users through a Structured Query Language (SQL) or some sort of forms. A transaction can also be embedded in the application program written in a high-level language such as C, Pascal, or COBOL.

A transaction processing (TP) system is a computer system that processes the transaction programs. A collection of such transaction programs designed to perform the functions necessary to automate given business activities is often called an application program (application software). Figure 1 shows a transaction processing system. The transaction programs are submitted to clients, and the requests will be scheduled by the transaction processing monitor and then processed by the servers. A TP monitor is a piece of software that connects multiple clients to multiple servers to access multiple data resources (databases) in TP systems. One objective of the TP monitor is to optimize the utilization of system and network resources when clients and servers execute on different processors.

Transaction processing is closely associated with database systems. In fact, most earlier transaction processing systems such as banking and airlines reservation systems are database systems, where data resources are organized into databases and transaction processing is supported by database management systems (DBMS). In traditional database systems, transactions are usually simple and independent, and are characterized as short duration in that they will be finished within minutes (probably seconds). Traditional transaction systems have some limitations for many advanced applications such as cooperative work, where transactions need to cooperate with each other. For example, in cooperative environments, several designers might work on the same project. Each designer starts up a cooperative transaction. Those cooperative transactions jointly form a transaction group. Cooperative transactions in the same transaction group may read or update each other's uncommitted (unfinished) data. Therefore, cooperative transactions may be interdependent. Currently, some research work on advanced transaction processing has been conducted in several related areas such as computer-supported cooperative work (CSCW) and groupware, workflow, and advanced transaction models (2–6). In this paper, we will first discuss traditional transaction concepts and then examine some advanced transaction models.

Because of recent developments in laptop or notebook computers and low-cost wireless digital communication, mobile computing began to emerge in many applications. As wireless computing leads to situations where machines and data no longer have fixed locations in the network, distributed transactions will be difficult to coordinate, and data consistency will be difficult to maintain. In this paper we will also briefly discuss the problems and possible solutions in mobile transaction processing.

This paper is organized as follows. First, we will introduce traditional database transaction processing, including concurrency control and recovery in centralized database transaction processing. The next section covers the topics on distributed transaction processing. Then we discuss advanced transaction processing and define an advanced transaction model and a correctness criterion. Mobile transaction processing is also presented. Finally future research directions are included.

DATABASE TRANSACTION PROCESSING

Because database systems are the earlier form of transaction processing systems, we will start with database transaction processing.

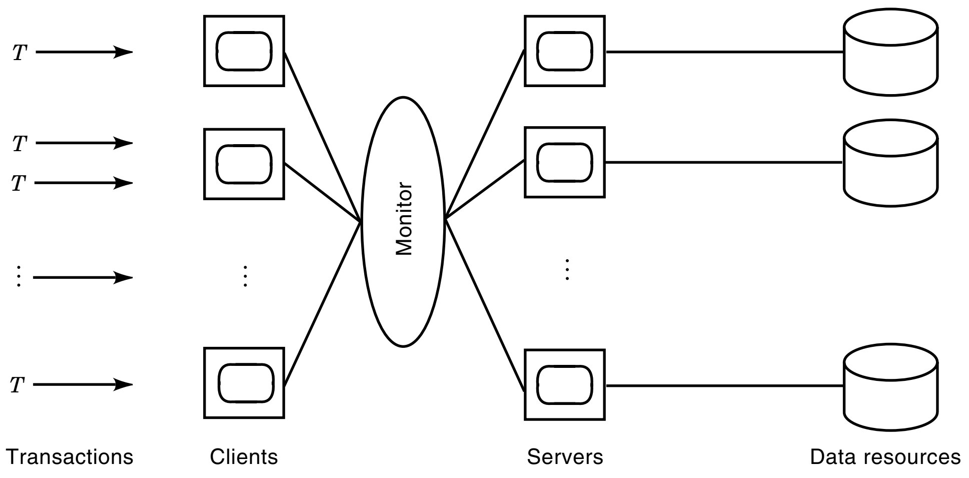

Figure 1. TP monitor between clients and data resources.

Databases Transactions

A database system refers to a database and the access facilities (database management system) to the database. One important job of database management systems is to control and coordinate the execution of concurrent database transactions.

A database is a collection of related data items that satisfy a set of integrity constraints. The database should reflect the relevant state as a snapshot of the part of the real world it models. It is natural to assume that the states of the database are constrained to represent the legal (permissible) states of the world. The set of *integrity constraints* such as functional dependencies, referential integrity, inclusion, exclusion constraints, and some other user-defined constraints are identified in the process of information analysis of the application domain. These constraints represent real-world conditions or restrictions (7). For example, functional dependencies specify some constraints between two sets of attributes in a relation schema while referential integrity constraints specify constraints between two sets of attributes from different relations. For detailed definitions and discussions on various constraints we refer readers to Refs. 7 and 8. Here we illustrate only a few constraints with a simple example.

Suppose that a relational database schema has following two table structures for Employee and Department with attributes like Name and SSN:

```
Employee (Name, SSN, Bdate, Address, Dnumber)

Department (Dname, Dnumber, Dlocation).

Name—employee name
SSN—social security number
Bdate—birth date
Address—living address
Dnumber—department number
Dname—department name
Dlocation—department location
```

Each employee has a unique social security number (SSN) that can be used to identify the employee. For each SSN value in the Employee table, there will be only one associated value for Bdate, Address and Dnumber in the table, respectively. In this case, there are functional dependencies from SSN to Bdate, Address, Dnumber. If any Dnumber value in the Employee relation has the same Dnumber value in the Department relation, there will be a referential integrity constraint from *Employee*'s Dnumber to *Department*'s Dnumber.

A database is said to be "consistent" if it satisfies a set of integrity constraints. It is assumed that the initial state of the database is consistent. Because an empty database always satisfies all constraints, often it is assumed that the initial state is an empty database. It is obvious that a database system is not responsible for possible discrepancies between a state of the real world and the corresponding state of the database if the existing constraints were inadequately identified in the process of information analysis. The values of data items can be queried or modified by a set of application programs or transactions. Because the states of the database corresponding to the states of the real world are consistent, a transaction can be regarded as a transformation of a database from one consistent state to another consistent state. Users' access to a database is facilitated by the software system called a DBMS, which provides services for maintaining consistency, integrity, and security of the database. Figure 2 illustrates a simplified database system. The transaction scheduler provides functions for transaction concurrency control, and the recovery manager is for transaction recovery in the presence of failures, which will be discussed in the next section.

The fundamental purpose of the DBMS is to carry out queries and transactions. A query is an expression, in a suitable language, that determines a portion of the data contained in

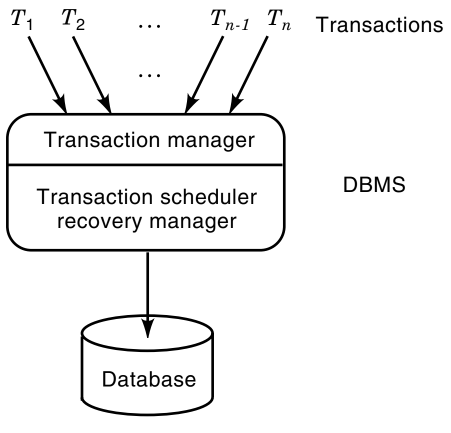

Figure 2. Database system and DBMS.

the database (9). A query is considered as a read-only transaction. The goal of query processing is extracting information from a large amount of data to assist a decision making process. A transaction is a piece of programming that manipulates the database by a sequence of read and write operations.

read(X) or R(X), which transfers the data item *X* from the database to a local buffer of the transaction

write(X) or W(X), which transfers the data item *X* from the local buffer of the transaction back to the database

In addition to *read* and *write* operations, a transaction starts with a *start* (or *begin*) operation, and ends with a *commit* operation when the transaction succeeds or an *abort* when the transaction fails to finish. The following example shows a transaction transferring funds between two bank accounts (*start* and *end* operations are omitted).

Example 1. Bank transfer transaction.

```
read($X$)
$X \leftarrow X + 100$
write($X$)
read($Y$)
$Y \leftarrow Y - 100$
write($Y$)
```

Here X and Y stand for the balances of savings and credit accounts of a customer, respectively. This transaction transfers some money (100 dollars) from the savings account to the credit account. It is an atomic unit of database work. That is, all these operations must be treated as a single unit.

Many database systems support multiple user accesses or transactions to the database. When multiple transactions execute concurrently, their operations are interleaved. Operations from one transaction may be executed between operations of other transactions. This interleaving may cause inconsistencies in a database, even though the individual transactions satisfy the specified integrity constraints. One such example is the *lost update* phenomenon.

Example 2. For the lost update phenomenon, sssume that two transactions, crediting and debiting the same bank account, are executed at the same time without any control. The data item being modified is the account balance. The transactions read the balance, calculate a new balance based on the relevant customer operation, and write the new balance to the file. If the execution of the two transactions interleaves in the following pattern (supposing the initial balance of the account is 1500), the customer will suffer a loss:

Debit Transaction	Credit Transaction
read balance (\$1500)	
	read balance (\$1500)
withdraw (\$1000)	
	deposit (\$500)
balance := \$1500 − \$1000	
	balance := \$1500 + \$500
	Write balance (\$2000)
Write balance (\$500)	

The final account balance is \$500 instead of \$1000. Obviously these two transactions have produced an inconsistent state of the database because they were allowed to operate on the same data item and neither of them was completed before another. In other words, neither of these transactions was treated as an atomic unit in the execution. Traditionally, transactions are expected to satisfy the following four conditions, known as ACID properties (9–11):

Atomicity is also referred to as the all-or-nothing property. It requires that either all or none of the transaction's operations are performed. Atomicity requires that if a transaction fails to commit, its partial results cannot remain in the database.

Consistency requires a transaction to be correct. In other words, if a transaction is executed alone, it takes the database from one consistent state to another. When all the members of a set of transactions are executed concurrently, the database management system must ensure the consistency of the database.

Isolation is the property that an incomplete transaction cannot reveal its results to other transactions before its commitment. This is the requirement for avoiding the problem of *cascading abort* (i.e., the necessity to abort all the transactions that have observed the partial results of a transaction that was later aborted).

Durability means that once a transaction has been committed, all the changes made by this transaction must not be lost even in the presence of system failures.

The ACID properties are also defined in RM-ODP (Reference Model of Open Distributed Processing) (12). ODP is a standard in a joint effort of the International Standardization Organization (ISO) and International Telecommunication Union (ITU), which describes systems that support heterogeneous distributed processing both within and between organizations through the use of a common interaction model.

Consistency and isolation properties are taken care of by the concurrency control mechanisms, whereas the maintenance of atomicity and durability are covered by the recovery services provided in a transaction management. Therefore, concurrency control and recovery are the most important tasks for transaction management in a database system.

Concurrency Control and Serializability

The ACID properties can be trivially achieved by the sequential execution of transactions. However, this is not a practical solution because it severely damages system performance. Usually, a database system is operating in a multiprogramming, multiuser environment, and the transactions are expected to be executed in the database system concurrently. In this section, the concepts of transaction concurrency control, the schedule of transactions, and the correctness criterion used in concurrency control are discussed.

A database system must monitor and control the concurrent executions of transactions so that overall correctness and database consistency are maintained. One of the primary tasks of the database management system is to allow several users to interact with the database simultaneously, giving users the illusion that the database is exclusively for their own use (13). This is done through a concurrency control mechanism.

Without a concurrency control mechanism, numerous problems can occur: the lost update (illustrated earlier in an example), the temporary update (or the uncommitted dependency), and the incorrect summary problems (7,14). The unwanted results may vary from annoying to disastrous in the critical applications. Example 3 shows a problem of temporary updates where a transaction T_B updates a data item f_1 but fails before completion. The value of f_1 updated by T_B has been read by another transaction T_A.

Example 3. Consider an airline reservation database system for customers booking flights. Suppose that a transaction A attempts to book a ticket on flight F_1 and on flight F_2 and that a transaction B attempts to cancel a booking on flight F_1 and to book a ticket on flight F_3.

Let f_1, f_2, and f_3 be the variables for the seat numbers that have been booked on flights F_1, F_2, and F_3, respectively. Assume that transaction B has been aborted for some reason so that the scenario of execution is as follows:

Transaction A	Transaction B
	$R[f_1]$
	$f_1 = f_1 - 1$
	$W[f_1]$
$R[f_1]$	
$f_1 = f_1 + 1$	
$W[f_1]$	
	$R[f_3]$
	$f_3 = f_3 + 1$
	$W[f_3]$
$R[f_2]$	
$f_2 = f_2 + 1$	
$W[f_2]$	
	Abort transaction B
Commit transaction A	

It is obvious that both transactions are individually correct if they are executed in a serial order (i.e., one commits before another). However, the interleaving of the two transactions shown here causes a serious problem in that the seat on fight F_1 canceled by transaction B may be the last available one and transaction A books it before transaction B aborts. This results in one seat being booked by two clients.

Therefore, a database system must control the interaction among the concurrent transactions to ensure the overall consistency of the database. The execution sequence of operations from a set of transactions is called a *schedule* (15,16). A schedule indicates the interleaved order in which the operations of transactions were executed. If the operations of transactions are not interleaved (i.e., the executions of transactions are ordered one after another) in a schedule, the schedule is said to be serial. As we mentioned earlier, the serial execution of a set of correct transactions preserves the consistency of the database. Because serial execution does not support concurrency, the equivalent schedule has been developed and applied for comparisons of a schedule with a serial schedule, such as view equivalence and conflict equivalence of schedules. In general, two schedules are *equivalent* if they have the same set of operations producing the same effects in the database (15).

Definition 1. Two schedules S_1, S_2 are *view equivalent* if

1. for any transaction T_i, the data items read by T_i in both schedules are the same,
2. for each data item x, the latest value of x is written by the same transaction in both schedules S_1 and S_2

Condition 1 ensures that each transaction reads the same values in both schedules, and Condition 2 ensures that both schedules result in the same final systems.

In conflict equivalence, only the order of conflict operations needs to be checked. If the conflict operations follow the same order in two different schedules, the two schedules are conflict equivalent.

Definition 2. Two operations are in *conflict* if

1. they come from different transactions
2. they both operate on the same data item and at least one of them is a write operation

Definition 3. Two schedules S_1, S_2 are *conflict equivalent* if for any pair of transactions T_i, T_j in both schedules and any two conflicting operations $o_{ip} \in T_i$ and $o_{jq} \in T_j$, when the execution order o_{ip} precedes o_{jq} in one schedule, say, S_1, the same execution order must exist in the other schedule, S_2.

Definition 4. A schedule is *conflict serializable,* if it is conflict equivalent to a serial schedule. A schedule is *view serializable* if it is view equivalent to a serial schedule.

A conflict serializable schedule is also view serializable but not vice versa because definition of view serializability accepts a schedule that may not necessarily be conflict serializable. There is no efficient mechanism to test schedules for view serializability. It was proven that checking for view serializability is an NP-complete problem (17). In practice, the conflict serializability is easier to implement in the database systems because the serialization order of a set of transactions can be determined by their conflicting operations in a serializable schedule.

The conflict serializability can be verified through a conflict graph. The conflict graph among transactions is constructed as follows: for each transaction T_i, there is a node in the graph (we also name the node T_i). For any pair of conflicting operations (o_i, o_j), where o_i from T_i and o_j from T_j, respectively, and o_i comes before o_j, add an arc from T_i to T_j in the conflict graph.

Examples 4 and 5 present schedules and their conflict graphs.

Example 4. A nonserializable schedule is shown here. Its conflict graph is shown in Fig. 3.

Schedule	T_1	T_2	T_3
read(A)	read(A)		
read(B)		read(B)	
$A \leftarrow A + 1$	$A \leftarrow A + 1$		
read(C)			read(C)
$B \leftarrow B + 2$		$B \leftarrow B + 2$	
write(B)		write(B)	
$C \leftarrow C * 3$			$C \leftarrow C * 3$
write(C)			write(C)
write(A)	write(A)		
read(B)			read(B)
read(A)		read(A)	
$A \leftarrow A - 4$		$A \leftarrow A - 4$	
read(C)	read(C)		
write(A)		write(A)	
$C \leftarrow C - 5$	$C \leftarrow C - 5$		
write(C)	write(C)		
$B \leftarrow 6 * B$			$B \leftarrow 6 * B$
write(B)			write(B)

Example 5. A serializable schedule is shown here. Its conflict graph is shown in Fig. 4.

Schedule	T_1	T_2	T_3
read(A)	read(A)		
$A \leftarrow A + 1$	$A \leftarrow A + 1$		
read(C)	read(C)		
write(A)	write(A)		
$C \leftarrow C - 5$	$C \leftarrow C - 5$		
read(B)		read(B)	
write(C)	write(C)		
read(A)		read(A)	
read(C)			read(C)
$B \leftarrow B + 2$		$B \leftarrow B + 2$	
write(B)		write(B)	
$C \leftarrow 3 * C$			$C \leftarrow 3 * C$
read(B)			read(B)
write(C)			write(C)
$A \leftarrow A - 4$		$A \leftarrow A - 4$	
write(A)		write(A)	
$B \leftarrow 6 * B$			$B \leftarrow 6 * B$
write(B)			write(B)

The following theorem shows how to check the serializability of a schedule.

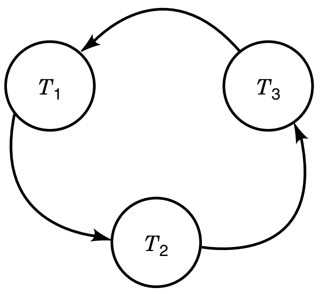
$$\begin{tikzcd} T_1 \arrow[dr, bend right] & & T_3 \arrow[ll, bend right] \\ & T_2 \arrow[ur, bend right] & \end{tikzcd}$$

Figure 3. Conflict graph 1 (with a cycle).

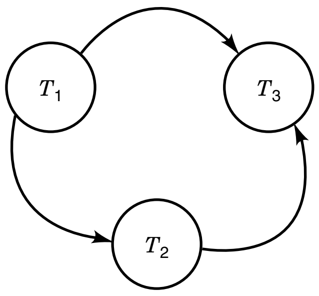
$$\begin{tikzcd} T_1 \arrow[rr, bend left] \arrow[dr, bend right] & & T_3 \\ & T_2 \arrow[ur, bend right] & \end{tikzcd}$$

Figure 4. Conflict graph 2 (without cycle).

Theorem 1. A schedule is conflict serializable if and only if its conflict graph is acyclic: (15).

Intuitively, if a conflict graph is acyclic, the transactions of the corresponding schedule can be topologically sorted such that conflict operations are consistent with this order, and therefore equivalent to a serial execution in this order. A cyclic graph implies that no such an order exists. The schedule in Example 4 is not serializable because there is cycle in the conflict graph; however, the schedule in Example 5 is serializable. The serialization order of a set of transactions can be determined by their conflicting operations in a serializable schedule.

In order to produce conflict serializable schedules, many concurrency control algorithms have been developed such as two-phase locking, timestamp ordering, and optimistic concurrency control.

The Common Concurrency Control Approaches

Maintaining consistent states in a database requires such techniques as semantic integrity control, transaction concurrency control, and recovery. Semantic integrity control ensures database consistency by rejecting update programs that violate the integrity constraints of the database. This is done by specifying the constraints during the database design. Then the DBMS checks the consistency during transaction executions. Transaction concurrency control monitors the concurrent executions of programs so that the interleaved changes to data items still preserve the database consistency. Recovery of a database system ensures that the system can cope with various failures in the system and recover the database to a consistent state.

A number of concurrency control algorithms have been proposed for the database management systems. The most fundamental algorithms are two-phase locking (18,19), timestamp ordering (20,21), optimistic concurrency control (22), and serialization graph testing (23,24).

Two-phase locking (2PL) is one of the most popular concurrency control algorithms based on the locking technique. The main idea of locking is that each data item must be locked before a transaction accesses it (i.e., if conflicting operations exist, only one of them can access the data at a time, and the other must wait until the previous operation has been completed and the lock has been released). A transaction may involve accesses to many data items. The rule of 2PL states that all locks of the data items needed by a transaction should be acquired before a lock is released. In other words, a transaction should not release a lock until it is certain that it will not request any more locks. Thus, each transaction has two phases: an *expanding phase* during which new locks on data items can be acquired but none can be released; and a *shrinking phase* in which the transaction releases locks and no new locks are required.

The 2PL algorithm is a very secure way to ensure that the order of any two transactions is compatible with the order of their conflicting operations. More precisely, if $o_{ip} \in T_i$ precedes $o_{jq} \in T_j$ in the schedule and o_{ip} is in conflict with o_{jq}, then all other conflicting operations of T_i, T_j must have the same order of precedence. The 2PL algorithms guarantee the conflict serializability of a schedule for concurrent transactions. However, 2PL algorithms may lead to deadlocks when a set of transactions wait for each other in a circular way. For example, two transactions T_1 and T_2 both write data items a and b. T_1 holds a lock on a and waits for a lock on b, while T_2 holds a lock on b and waits for a lock on a. In this case, T_1 and T_2 will be waiting for each other, and a deadlock occurs. When a deadlock occurs, some transactions need to be aborted to break the cycle.

Timestamp ordering (TO) is used to manage the order of the transactions by assigning timestamps to both transactions and data items. Each transaction in the system is associated with a unique timestamp, assigned at the start of the transaction, which is used to determine the order of conflicting operations between transactions. Each data item is associated with a read timestamp, which is the timestamp of the latest transaction which has read it, and a write timestamp, which is the timestamp of the latest transaction which has updated it. Conflicting operations must be executed in accordance with their corresponding transaction timestamps. A transaction will be aborted when it tries to read or write on a data item whose timestamp is greater than that of the transaction. The serializable order of transactions is the order of their timestamps.

Both 2PL and TO concurrency control algorithms are considered pessimistic approaches. The algorithms check every operation to determine whether the data item is available according to the locking or timestamp, even though the probability of conflicts between transactions is very small. This check represents significant overhead during transaction execution, with the effect of slowing down the transaction processing.

Optimistic concurrency control (OCC) (22) is another approach in which no check is done while the transaction is executing. It has better performance if it is used in the environment where conflicts between transactions are rare. During transaction execution, each transaction executes three phases in its life time. The following three phases are used in the OCC protocol:

1. *Read Phase.* The values of the data items are read and stored in the local variables of the transaction. All modifications on the database are performed on temporary local storage without updating the actual database.
2. *Validation Phase.* According to the mutually exclusivity rules, a validation test is performed to determine whether the updates can be copied to the actual database.
3. *Write Phase.* If the transaction succeeds in the validation phase, the actual updates are performed to the database; otherwise, the transaction is aborted.

Optimistic approaches are generally used in conjunction with timestamps. A timestamp is assigned to a transaction at the end of its read phase or before the validation phase. The serialization order of transactions is then validated using the timestamps.

In a *serialization graph*-based concurrency control protocol, an on-line serialization graph (conflict graph) is explicitly maintained. The serialization graph testing (SGT) scheduler maintains a serialization graph for the history that represents the execution it controls. When a SGT scheduler receives an operation o_i of transaction T_i from the transaction manager, it first adds a node for T_i in the serialization graph (SG). The scheduler then checks whether there exists a previously scheduled operation o_k of transaction T_k conflicting with o_i. If there is one, an arc from T_k to T_i is added to the SG. The operations of transaction T_i can be executed as long as the graph is acyclic. Otherwise, the transaction, which causes a cycle in the graph, is aborted. Because the acyclic serialization graph guarantees the serializability of the execution, the SGT scheduler produces the correct schedules for the concurrent transactions. However, it is not necessarily recoverable and is much less cascadeless or strict (14) as defined later.

A schedule S is said to be recoverable, if for every transaction T_i that reads data items written by another transaction T_j in S, T_i can be committed only after T_j is committed. That is, a recoverable schedule avoids the situation where a committed transaction reads the data items from an aborted transaction. A recoverable schedule may still cause cascading aborts, because it allows the transactions to read from uncommitted transactions. For example, a transaction T_2 reads a data item x after x is updated by a transaction T_1, which is still active in an execution. If T_1 is aborted during the processing, T_2 must be aborted. Cascading aborts are undesirable.

To avoid cascading abortion in a schedule S, every transaction should read only those values written by committed transactions. Thus, a cascadeless schedule is also a recoverable schedule.

Because a cascadeless schedule allows transaction to write data from an uncommitted transaction, an undesirable situation may occur (14). For instance, consider the scenario of an execution

$$W_{T_1}[x, 2]W_{T_2}[x, 4], \text{Abort}(T_1)\text{Abort}(T_2)$$

where two transactions T_1 and T_2 write the same data item x, with values 2 and 4, respectively, and both are aborted later. The value of the data item x is called a before image if it will be replaced by a new value. The *before image* is saved in the log. In this case, the before image of data item x for transaction T_2 is 2 written by an aborted transaction T_1.

The term *strict schedule* was introduced in Ref. 14 to describe a very important property from a practical viewpoint. A schedule of transactions is called strict, if the transactions read or write data items only from committed transactions. Strict schedules avoid cascading aborts and are recoverable. They are conservative and offer less concurrency.

The concurrency control algorithms presented above such as 2PL, TO, and SGT do not necessarily produce strict schedules by themselves.

If a strict schedule using 2PL algorithm is required, the locks being held by any transaction can be released only after the transaction is committed.

A TO approach with a strict schedule will not allow a transaction T to access the data items that have been updated by a previous uncommitted transaction even if transaction T holds a greater timestamp.

Serialization graph testing can produce a strict schedule in such a way that each transaction cannot be committed until it is a source node of the serialization graph. That is, a transaction T could not be involved in a cycle of the serializable testing graph if previous transactions which T reads or writes from have all been committed.

Recoverability of Transactions

In addition to concurrency control, another important goal of transaction management is to provide a reliable and consistent database in the presence of various failures. Failures may corrupt the consistency of the database because the execution of some transactions may be only partially completed in the database. In general, database systems are not failure-free systems. A number of factors cause failures in a database system (9) such as:

1. *Transaction Abortions.* The situation can be caused by the transaction itself, which is caused by some unsatisfactory conditions. Transaction abortion can also be forced by the system. These kinds of failure do not damage the information stored in memory, which is still available for recovery.
2. *System Crashes.* The typical examples of this type of failure are system crashes or power failures. These failures interrupt the execution of transactions, and the content of main memory is lost. In this case, the only available accessible information is from a stable storage, usually a disk.
3. *Media Failures.* Failures of the secondary storage devices that store the database are typical of media failure. Because the content of stable storages has been lost, the system cannot be recovered by the system software only. The common technique to prevent such unrecoverable failures is to replicate the information on several disks.

The first two types of failures are considered in the recovery of transactions. Transactions represent the basic units of recovery in a database system. If the automicity and durability of the execution of each transaction have been guaranteed in the presence of failures, the database is considered to be consistent.

Typically, the piece of software responsible for recovery of transactions is called the recovery manager (RM). It is required to ensure that whenever a failure occurs, the database is brought back to the consistent state it was in before the failure occurred. In other words, the recovery manager should guarantee that updates of the database by the committed transactions are permanent, in contrast to any partial effects of uncompleted transactions that should be aborted.

The basic technique for implementing transactions in the presence of failures is based on the use of logs. A log is a file that records all operations on the database carried out by all transactions. It is supposed that a log is accessible after the failures occur. The log is stored in *stable storage,* which is the most resilient storage medium available in the system. Stable storage is also called *secondary storage.* Typically, it is implemented by means of duplexed magnetic tapes or disks that store duplicate copies of the data. The replicated stable storage is always kept mutually consistent with the primary copy of the disk or tape. The database is stored permanently on stable storage. The updates on a database by a transaction are not directly written into the database immediately. The operations of the transactions are implemented in the *database buffer* located in main memory (also referred to as volatile storage). It is only when the contents of the database buffer have been *flushed* to stable storage that any update operation can be regarded as durable.

It is essential that the log record all the updates on the database that have been carried out by the transactions in the system before the contents of the database buffer have been written to database. This is the rule of *write-ahead log.*

A log contains the information for each transaction as follows:

- transaction identifier;
- list of update operations performed by the transaction (For each update operation, both the old value and new value of the data items are recorded.); and
- status of the transaction: tentative, committed, or aborted.

The log file records the required information for undoing or redoing the transaction if a failure occurs. Because the updates were written to the log before flushing the database buffer to the database, the recovery manager can surely preserve the consistency of the database. If a failure occurs before the commit point of a transaction is reached, the recovery manager will abort the transaction by *undo*ing the effect of any partial results that have been flushed into the database. On the other hand, if a transaction has been committed but the results have not been written into the database at the time of failure, the recovery manager would have to *redo* the transaction, using the information from the log, in order to ensure transaction durability.

DISTRIBUTED TRANSACTION PROCESSING

In many applications, both data and operations are often distributed. A database is considered distributed if a set of data that belongs logically to the same system is physically spread over different sites interconnected by a computer network. A site is a host computer and the network is a computer-to-computer connection via the communication system. Even though the software components which are typically necessary for building a database management system are also the principal components for a distributed DBMS (DDBMS), some additional capacities must be provided for a distributed database, such as the mechanisms of distributed concurrency control and recovery.

One of the major differences between a centralized and a distributed database system lies in the transaction processing. In a distributed database system, a transaction might involve data residing on multiple sites (called a global transaction). A global transaction is executed on more than one site. It consists of a set of subtransactions—each subtransaction involving data residing on one site. As in centralized databases, global transactions are required to preserve the ACID properties. These properties must be maintained individually on each site and also globally. That is, the concurrent global transactions must be serializable and recoverable in the distributed database system. Consequently, each subtransaction of a global transaction must be either performed in its entirety or not performed at all.

Serializability in a Distributed Database

Global transactions perform operations at several sites in a distributed database system (DDBS). It is well understood that the maintenance of the consistency of each single database does not guarantee the consistency of the entire distributed database. It follows, for example, from the fact that serializability of executions of the subtransactions on each single site is only a necessary (but not sufficient) condition for the serializability of the global transactions. In order to ensure the serializability of distributed transactions, a condition stronger than the serializability of single schedule for individual sites is required.

In the case of distributed databases, it is relatively easy to formulate a general requirement for correctness of global transactions. The behavior of a DDBS is the same as a centralized system but with distributed resources. The execution of the distributed transactions is correct if their schedule is serializable in the whole system. The equivalent conditions are:

- Each local schedule is serializable;
- The subtransactions of a global transaction must have a compatible serializable order at all participating sites.

The last condition means that for any two global transactions G_i and G_j, their subtransactions must be scheduled in the same order at all the sites on which these subtransactions have conflicting operations. Precisely, if G_{ik} and G_{jk} belongs to G_i and G_j, respectively, and the local serializable order is G_{ik} precedes G_{jk} at site k, then all the subtransactions of G_i must precede the subtransactions of G_j at all sites where they are in conflict.

Various concurrency control algorithms such as 2PL and TO have been extended to distributed database systems. Because the transaction management in a distributed database system is implemented by a number of identical local transaction managers, the local transaction managers cooperate with each other for the synchronization of global transactions. If the timestamp ordering technique is used, a global timestamp is assigned to each subtransaction, and the order of timestamps is used as the serialization order of global transactions. If a two-phase locking algorithm is used in the distributed database system, the locks of a global transaction cannot be released at all local sites until all the required locks are granted. In distributed systems, the data item might be replicated. The updates to replicas must be atomic (i.e., the replicas must be consistent at different sites). The following rules may be used for locking with n replicas:

1. Writers need to lock all n replicas; readers need to lock one replica.
2. Writers need to lock all m replicas ($m > n/2$); readers need to lock $n - m + 1$ replicas.
3. All updates are directed first to a primary copy replica (one copy has been selected as the primary copy for updates first and then the updates will be propagated to other copies).

Any one of these rules will guarantee consistency among the duplicates.

Atomicity of Distributed Transactions

In a centralized system, transactions can either be processed successfully or aborted with no effects left on the database in the case of failures. In a distributed system, however, additional types of failure may happen.

For example, network failures or communication failures may cause network partition, and the messages sent from one site may not reach the destination site. If there is a partial execution of a global transaction at a partitioned site in a network, it would not be easy to implement the atomicity of a distributed transaction. To achieve an atomic commitment of a global transaction, it must be ensured that all its subtransactions at different sites are capable and available to commit. Thus an agreement protocol has to be used among the distributed sites. The most popular atomic commitment protocol is the two-phase commitment (2PC) protocol.

In the basic 2PC, there is a *coordinator* at the originating site of a global transaction. The participating sites that execute the subtransactions must commit or abort the transaction unanimously. The coordinator is responsible for making the final decision to terminate each subtransaction. The first phase of 2PC is to request from all *participants* the information on the execution state of subtransactions. The participants report to the coordinator, which collects the answers and makes the decision. In the second phase, that decision is sent to all participants. In detail, the 2PC protocol proceeds as follows for a global transaction T_i (9):

Two-Phase Commit Protocol

Phase 1: Obtaining a Decision

1. Coordinator asks all participants to prepare to commit transaction T_i:
   a. Add [prepare T_i] record to the log
   b. Send [prepare T_i] message to each participant
2. When a participant receives [prepare T_i] message it determines if it can commit the transaction:
   a. If T_i has failed locally, respond with [abort T_i]
   b. If T_i can be committed, send [ready T_i] message to the coordinator
3. Coordinator collects responses:
   a. All respond "ready"; decision is commit
   b. At least one response is "abort"; decision is abort
   c. At least one fails to respond within time-out period, decision is abort

Phase 2: Recording the Decision in the Database

1. Coordinator adds a decision record ([abort T_i] or [commit T_i]) in its log

2. Coordinator sends a message to each participant informing it of the decision (commit or abort)
3. Participant takes appropriate action locally and replies "done" to the coordinator

The *first phase* is that the coordinator initiates the protocol by sending a "prepare-to-commit" request to all participating sites. The "prepare" state is recorded in the log, and the coordinator is waiting for the answers. A participant will reply with a "ready-to-commit" message and record the "ready" state at the local site if it has finished the operations of the subtransaction successfully. Otherwise, an "abort" message will be sent to the coordinator, and the subtransaction will be rolled back accordingly.

The *second phase* is that the coordinator decides whether to commit or abort the transaction based on the answers from the participants. If all sites answered "ready-to-commit," then the global transaction is to be committed. The final "decision-to-commit" is issued to all participants. If any site replies with an "abort" message to the coordinator, the global transaction must be aborted at all the sites. The final "decision-to-abort" is sent to all the participants who voted the "ready" message. The global transaction information can be removed from the log when the coordinator has received the "completed" message from all the participants.

The basic idea of 2PC is to make an agreement among all the participants with respect to committing or aborting all the subtransactions. The atomic property of global transaction is then preserved in a distributed environment.

The 2PC is subject to the blocking problem in the presence of site or communication failures. For example, suppose that a failure occurs after a site has reported "ready to commit" for a transaction, and a global commitment message has not yet reached this site. This site would not be able to decide whether the transaction should be committed or aborted after the site is recovered from the failure. A three-phase commitment (3PC) protocol (14) has been introduced to avoid the blocking problem. But 3PC is expensive in both time and communication cost.

Transaction Processing in Heterogeneous Systems

Traditional distributed database systems are often homogeneous because local database systems are the same, using the same data models, the same languages, and the same transaction managements. However, in the real world, data are often partitioned across multiple database systems, file systems, and applications, all of which may run on different machines. Users may run transactions to access several of these systems as single global transactions. A special case of such systems are multidatabase systems or federated database systems.

Because the 2PC protocol is essential to support the atomicity of global transactions and, at the same time, the local systems may not provide such support, layers of software are needed to coordinate and the execution of global transactions (25) for transactional properties of concurrency and recovery. A TP monitor is a piece of software that connects multiple clients to multiple servers to access multiple databases/data resources as shown in Fig. 1. Further discussions on TP monitors can be found in Ref. 1.

ADVANCED TRANSACTION PROCESSING

In traditional database applications such as banking and airline reservation systems, transactions are short and noncooperative and usually can be finished in minutes. The serializability is a well-accepted correctness criterion for these applications. Transaction processing in advanced applications such as cooperative work will have different requirements, need different correctness criteria, and require different systems supports to coordinate the work of multiple designers/users and to maintain the consistency. Transactions are often called advanced transactions if they need nonserializable correctness criteria. Many advanced transaction models have been discussed in the literature (2–5). In this section, we will briefly examine some advanced transaction models and then present a general advanced transaction model and its correctness criterion.

Advanced Transaction Model

In addition to advanced transactions, we can also see other similar terms such as nontraditional transactions, long transactions, cooperative transactions, and interactive transactions. We will briefly list some work on advanced transaction processing or cooperative transactions processing in advanced database transaction models (2,3), groupware (4,26,27), and workflow systems (5,28).

- Advanced Database Transaction Models (3)
  1. Saga (29). A transaction in Saga is a long-lived transaction that consists of a set of relatively independent steps or subtransactions, T_1, T_2, . . ., T_n. Associated with each subtransaction T_i is a compensating transaction C_i, which will undo the effect of T_i. Saga is based on the compensation concept. Saga relaxes the property of isolation by allowing a Saga transaction to reveal its partial results to other transactions before it completes. Because a Saga transaction can interleave its subtransactions with subtransactions of other sagas in any order, consistency or serializability is compromised. Saga preserves atomicity and durability of traditional transaction by using forward and backward recoveries.
  2. Cooperative Transaction Hierarchy (30). This model supports cooperative applications like computer aided design (CAD). It structures a cooperative application as a rooted tree called a cooperative transaction hierarchy. The external nodes represent the transactions associated with the individual designers. An internal node is called a transaction group. The term cooperative transaction refers to transactions with the same parent in the transaction tree. Cooperative transactions need not to be serializable. Isolation is not required. Users will define correctness by a set of finite automata to specify the interaction rules between cooperative transactions.
  3. Cooperative SEE Transactions (31). This model supports cooperative work in software engineering environments (SEEs). It uses nested active transactions with user defined correctness. ACID properties are not supported.

4. DOM Transaction Model for distributed object management (32). This model uses open and closed nested transactions and compensating transactions to undo the committed transactions. It also use contingency transactions to continue the required work. It does not support ACID properties.
5. Others (3). Open nested transactions, ConTract, Flex, S, and multilevel transactions models use compensating transactions and contingency transactions. The ACID properties are compromised. The polytransaction model uses user defined correctness. Tool Kit also uses user defined correctness, and contingency transactions to achieve the consistency.

- Groupware (2,26,33). Most groupware systems synchronize cooperative access to shared data in a more or less ad-hoc manner. Groupware systems involve multiple concurrent users or several team members at work on the same task. The members, or users, are often in different locations (cities or even countries). Each team member starts up a cooperative transaction, each cooperative transaction should be able to see the intermediate result of other cooperative transactions, and these cooperative transactions jointly form a cooperative transaction group. When they read or update the uncommitted data from other cooperative transactions, nonserializable synchronization and concurrency mechanisms are required to maintain consistency. A cooperative editing system is an example.
- Workflow applications (5). Workflow is used to analyze and control complicated business processes. A large application often consists of a collection of tasks. Each task can be viewed as a cooperative transaction processed by one user or designer, and these tasks are partially ordered by control and data flow dependencies. The workflow supports the task coordination specified in advance through the control flow. Serializability is not preserved either.

These applications have some common properties: (1) users are often distributed; (2) they conduct some cooperative work in an interactive fashion; and (3) this interactive cooperative work may take a long time. These applications have the following special consistency requirements:

1. A transaction may read intermediate results produced by other transactions.
2. The consistency between individual and group needs to be maintained.

Based on this summary, we give the following definition.

Definition 5. An advanced transaction (cooperative transaction group) is defined as a set (group) of cooperative transactions $T_1, T_2, \ldots, T_n$, with the following properties:

1. Each cooperative transaction is a sequence (or partial order) of read(x) and write(y) operations.
2. For the same data item, there might be more than one read(x), written as read^1(x), read^2(x), . . ., in a cooperative transaction, and each read(x) will get a different value depending on the time and interaction with other transactions.
3. Similarly, for each y, there might be more than one write(y), written as write^1(y), write^2(y), . . ., each of which will produce an individual version of data item y.

The first part shows that an advanced transaction is a cooperative transaction group. If the size of the group is one, it will become a single transaction. The property 1 is the same as that in traditional transactions. The second and third properties indicate some cooperative features. The first read(x) may read other transaction's committed or uncommitted data depending on the concurrency control employed. After the first read operation on x, the data item might be updated by another transaction or another cooperative transaction; then it can read the new value in the next read(x). Similarly, after the first write operation on x, because of the cooperative feature, a transaction may read some new data from other transactions and then issue another write(x) to incorporate this to the current processing. The later write(x) can undo the previous write or do a further update to show the new semantics.

To further justify the second and third properties of the definition, we discuss their compatibilities with interactive and noninteractive transactions in advanced transaction applications.

- *Interactive transactions.* A cooperative transaction can be formed with great flexibility because a user can dynamically issue an operation depending on the most current information. If a data item has been updated recently after the first read, the cooperative transaction may wish to read the data again because of the cooperative feature. In order to incorporate the recent changes in to its own transaction, it can perform additional operations or compensate for the previous operations. That is also the flexibility of interactive work.
- *Noninteractive transactions.* In some database transaction models, the transactions are not as interactive as those on-line transactions from groupwares and transaction workflow applications (3). To maintain system consistency and meet the application requirements, all of them use compensating transactions, contingency transactions, or triggers, where a compensating transaction is a transaction undoing the effect of a previous transaction; a contingency transaction is a transaction to continue or extend a previous transaction; and the trigger is a mechanism to invoke another transaction (if the trigger condition is true) to restore the consistency. A compensating transaction, a contingency transaction, or a trigger can be viewed as an extension of a transaction that violates the consistency requirements during the execution, and the extended part will have the read and write operations on some data items in common. They are another type of interaction. These interactions need to be programmed in advance; therefore, they are not as flexible as interactive transactions. But the interactive features are still required even for these noninteractive database transaction applications.

Similar to distributed database transactions, the advanced transaction definition could be extended to a distributed advanced transaction as follows:

Definition 6. A distributed advanced transaction (distributed cooperative transaction group) is defined as a set (group) of cooperative transactions $T_1, T_2, \ldots, T_n$, with the following properties:

1. Each transaction T_i consists of a set of subtransactions T_i^j at site j, $j \in [1..m]$, m is the number of sites in a distributed system. Some T_i^j might be empty if T_i has no subtransaction at site j.
2. Each subtransaction is a sequence (or partial order) of read(x) and write(y) operations.
3. For the same data item x, there might be more than one read(x), denoted as $\text{read}^1(x)$, $\text{read}^2(x)$, . . ., in a cooperative transaction, each read(x) will get a different value depending on the time and interaction with other transactions.
4. Similarly, for each y, there might be more than one write(y), denoted as $\text{write}^1(y)$, $\text{write}^2(y)$, . . ., each of which will produce an individual version of data item y.

Just as the serializability theory plays an important role in the traditional transaction model in developing concurrency control and recovery algorithms, a general correctness theory for advanced transactions is also required to guide transaction management for advanced applications. In the next subsection, we will present such a correctness criterion.

f-Conflict Serializability

As in the traditional transactions, we can assume that for write operations on x, there must be a read operation before the first write in a cooperative transaction. It is natural to read the data first before the update, [i.e., one's update may depend on the read value or one may use a read operation to copy the data into the local memory, then update the data and write it back (when the transaction commits)].

In advanced transaction applications, cooperative transactions could read and write a data item more than once, which is different from traditional transactions. The reason for reading a data item more than once is to know the recent result and therefore make the current transaction more accurate. However, this will violate the serializability, because a cooperative transaction may read a data item before another transaction starts and also read the data updated by the same transaction. If so, the schedule between these two transactions will not be serializable. However, from the semantic point of view, the most important read or write on the same data item will be the last read or write. If we give high priority to the last read or write conflicts in developing the correctness criteria, we could have an f-conflict (final conflict) graph, and based on this we will present an f-conflict serializability theorem as a general correctness criterion for advanced transaction processing.

Definition 7. The f-conflict graph among transactions is constructed as follows. For each transaction T_i, there is a node in the graph (we also name the node T_i). For any pair of final conflicting operations (o_i, o_j), where o_i from T_i and o_j from T_j, respectively, and o_i comes earlier than o_j, add an arc from T_i to T_j in the conflict graph.

Definition 8. A schedule is f-conflict serializable if and only if its f-conflict graph is acyclic.

The f-conflict serialization order of a set of transactions can be determined by their f-conflicting operations in an f-conflict serializable schedule. From the definitions, we can see the relationship between conflict serializability and f-conflict serializability.

Theorem 2. If a schedule is conflict serializable, it is also f-conflict serializable; the reverse is not true.

The conflict serializability is a special case of f-conflict serializability in traditional transaction processing.

Definition 9. A schedule of distributed advanced transactions is f-conflict serializable if and only if

1. the schedule of subtransactions at each site is f-conflict serializable, and
2. the f-conflict serialization order at all sites are the same.

Advanced transactions or cooperative transactions may have different application-dependent requirements and require different system supports to coordinate the work of multiple users and to maintain the consistency. As a result, different synchronization, coordination, and control mechanisms within a cooperative transaction group are developed. The f-conflict serializability in conjunction with application-dependent semantics could be used for designing and testing advanced transaction processing approaches. The application-dependent requirements can be reflected in the detailed transaction structures. For example, when there are several write operations on the same x, the later write might be to undo and then redo the operation (or perform a different operation). The undo operations might be reversing operations or compensating operations, and the redo operations could be contingency operations or new operations that may need to keep the intention (user intention) of the original write (6,27), or to incorporate the new semantics.

In a recent work, we have verified a cooperative editing system, REDUCE, according to this theory, and have shown that the schedules from this system is f-conflict serializable (34).

Mobile Transaction Processing

In both centralized and distributed database systems, data and machines have fixed locations. Because of the popularity of laptop or notebook computers and the development of relatively low-cost wireless digital communication based on the wireless local network, mobile computing began to emerge in many applications. The mobile computing environment consists of mobile computers, referred as mobile hosts, and a wired network of computers, some of which are mobile support stations through which mobile hosts can communicate with the wired network. Each mobile support station manages those mobile hosts within its cell—the geographical area it covers. Figure 5 shows a wired and wireless connected networking environment.

Because mobile hosts may move between cells, the handoff of control from one cell to another is necessary. Wireless

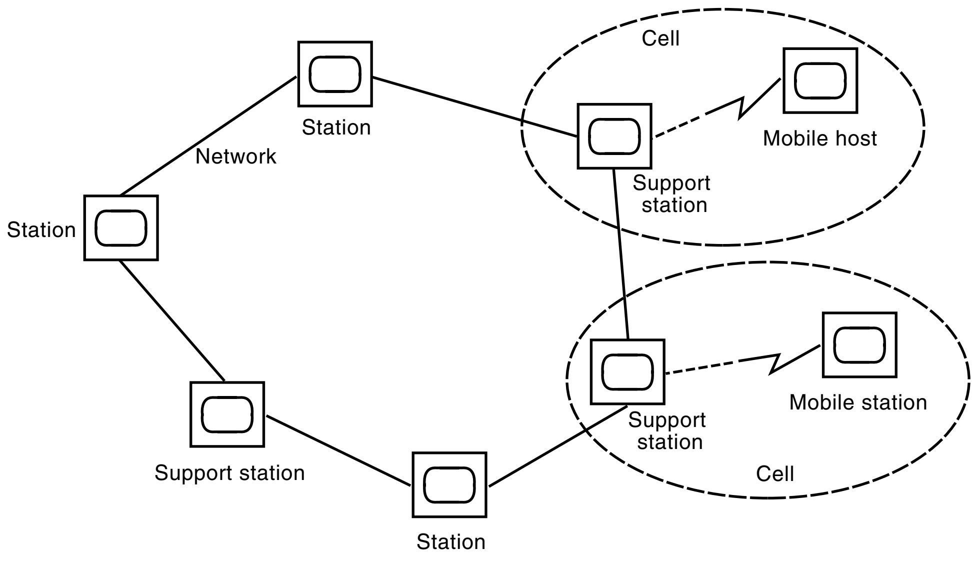

Figure 5. Wired and wireless networking environment.

computing creates a situation where machines and data no longer have fixed locations and network addresses. A consequence is that the route between a pair of hosts may change over time if one of the two hosts is a mobile computer.

Because the wireless communication may be charged for on the basis of connection time, and the battery is the limited power resource for mobile computers, most mobile computers will be disconnected for substantial periods. During the disconnection, mobile hosts may remain in operation. The users of the mobile host may issue query or update transactions on the data that reside locally. This may cause some problems related to recoverability and consistency. In wireless distributed systems, partitioning via disconnection is a normal mode of operation, whereas in wired distributed systems, the partitioning is considered a failure. Because the partitioning is normal, it is necessary to allow data access to proceed, despite partitioning. But this will cause system inconsistency when there are concurrent updates on replicas of the same data from different computers (mobile and nonmobile).

Mobile computing systems can be viewed as an extension of distributed systems (i.e., a dynamic type of distributed system where links in the network change dynamically). These intermittent links represent the connection between mobile units and the base stations (35). Many problems in distributed transaction processing are inherited by mobile transaction systems. Mobile transactions also introduce some new problems, such as location awareness and frequent disconnection from servers. In wired distributed database systems, location transparency is an important feature of transaction systems. However, mobile applications may be location dependent (i.e., the same query may have different results when submitted from different places). Failures occur much more frequently in mobile computing because of the frequent switching off and on of mobile units and the frequent handoff when mobile units move across the boundary of cells. This makes it difficult to preserve atomicity of mobile transactions. Failure handling and recovery is a new challenge in mobile systems.

The transaction execution and commitment model in the mobile environment is also different from traditional distributed transactions. A traditional distributed transaction consists of a set of subtransactions that are executed concurrently at multiple sites and there is one coordinator to coordinate the execution and commitment of the subtransactions. A mobile transaction is another kind of distributed transaction where some parts of the computation are executed on the mobile host and others on fixed hosts. The entire transaction can be submitted in a single request from the mobile unit, or the operations of a transaction are submitted in multiple requests, possibly to different support stations in different cells. The former submission involves a single coordinator for all the operations of the transaction, whereas the latter may involve multiple coordinators. For example, after submitting some operations (and getting partial results back), the mobile host might need to submit the remaining operations to another cell because it has moved to a new cell. The execution of the mobile transaction is not fully coordinated by a single coordinator because it depends on the movement of the mobile unit, to some extent. In this case, the interactive execution of transactions must be supported. Similar to other advanced transactions, a mobile transaction tends to be long lived because of the high latency of wireless communication and long disconnection time. A mobile transaction tends to be error-prone because mobile hosts are more prone to accidents than fixed hosts. Mobile transactions may access a distributed and heterogeneous system because of the mobility of the transaction (36).

Because mobile units are often disconnected from the rest of the network while still in operation, it will be difficult to maintain consistency under disconnection. For example, a data item is cached in the mobile unit and has been updated only by a mobile host, the update result can be propogated to other sites to achieve consistency when the mobile hosts reconnects. Whereas in some other cases, inconsistency may arise:

- If the data item is cached in a mobile computer as a read only copy and it is updated by the other computer while the mobile computer is disconnected, the cached data will become inconsistent or out of date.
- If updates can occur at the mobile computer and elsewhere, inconsistencies might occur.

To simplify the processing of read-only transactions, one could use a version vector, storing several versions of the data, and then read the consistent version. But this still does not solve the most difficult concurrent update problems. These problems generated in mobile computing are very similar to the problems in advanced transaction processing/cooperative transaction processing discussed earlier.

FUTURE RESEARCH DIRECTIONS

The future work on transaction processing will continue in the direction on new transaction models. Even though the advanced transaction model and f-conflict serializability provide a guideline for advanced application, many particular applications still need user-defined correctness, and often employ the semantic information for semantic serializability and semantic atomicity.

In advanced database applications such as CAD and cooperative work, the transactions are often cooperative or interactive or on-line analysis processing. We need to design mechanisms for advanced models to support partial rollbacks, reread, and rewrite operations to reflect the cooperative features.

Advanced transactions are very long when compared with traditional transactions. The arbitrary abortion of such long transactions is not appropriate because aborting long transactions means increasing the processing cost and response time. In an environment with short (traditional) transactions and long/cooperative transactions, long/cooperative transactions should not be aborted because of conflict operations with short transactions. On the other hand, because the quick response is often required or preferred for short transactions, long transactions should not block the short transactions.

In order to support both traditional and advanced transaction processing, more comprehensive transaction processing approaches for coexistence of various transactions need to be developed. The following features are expected and preferred for such an approach:

1. It allows short and long transactions (and mobile transactions) to coexist.
2. Short transactions can be processed in the traditional way, as if there were no advanced or cooperative transactions; therefore, they cannot be blocked by long transactions.
3. Advanced or cooperative transactions will not be aborted when there is a conflict with short transactions; rather, it will incorporate the recent updates into its own processing.
4. Some correctness criteria are preserved when there are interactions between advanced transactions and traditional transactions.

As database systems are being deployed in more and more complex applications, the traditional data model (e.g., the relational model) has been found to be inadequate and has been extended (or replaced) by object-oriented data models. Related to this extension is another research direction: transaction processing in object-oriented databases, including semantic-based concurrency control and recovery in object-oriented databases. Ref. 37 presents a brief introduction and some future research topics on this area as well as a comprehensive list of references on advanced transaction processing.

ACKNOWLEDGMENT

We thank Anne Fuller for her comments and review on an earlier version of this paper.

BIBLIOGRAPHY

1. P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing.* San Mateo, CA: Morgan Kaufmann, 1997.
2. K. Abrer et al., Transaction models supporting cooperative work-TransCoop experiences, in Y. Kambayashi and K. Yokota (eds.), *Cooperative Databases and Applications,* Singapore: World Scientific, 1997, pp. 347–356.
3. A. K. Elmagarmid, *Database Transaction Models for Advanced Applications,* San Mateo, CA: Morgan Kaufmann, 1992.
4. C. A. Ellis and S. J. Gibbs, Concurrency control in groupware systems. *Proc. ACM SIGMOD,* pp. 399–407, 1989.
5. M. Rusinkiewicz and A. Sheth, Specification and execution of transactional workflows, in W. Kim (ed.), *Modern Database Systems,* Reading, MA: Addison-Wesley, 1994, pp. 592–620.
6. C. Sun et al., A generic operation transformation scheme for consistency maintenance in real-time cooperative editing systems, *Proc. ACM Group97,* Phoenix, AZ, 1997, pp. 425–434.
7. R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems.* Menlo Park, CA: Benjamin/Cummins, 1989.
8. A. Silberschatz, H. Korth, and S. Sudarshan, *Database systems concepts,* 3 New York: McGraw-Hill, 1991.
9. S. Ceri and G. Pelagate, *Distributed Databases: Principles and Systems.* New York: McGraw-Hill, 1984.
10. T. Haerder and A. Reuter, Principles of transaction-oriented database recovery, *ACM Comput. Surv.,* **15** (4): 287–317, 1983.
11. J. N. Gray, The transactions concept: Virtues and limitations, *Proc. 7th Int. Conf. Very Large Data Base,* pp. 144–154, 1981.
12. ISO/IEC DIS 10746-2, Basic reference model of open distributed Processing—Part 2: descriptive model [Online]. Available: http://www.dstc.edu.au/AU/ODP/standards.html
13. D. Agrawal and A. El Abbadi, Transaction management in database systems, *Database Trans. Models Adv. Appl.,* 1992, pp. 1–32.
14. C. J. Date, *An Introduction to Database System,* Reading, MA: Addison-Wesley, 1982, Vol. 2.
15. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Reading, MA: Addison-Wesley, 1987.
16. H. Korth and A. Silberschatz, *Database Systems Concepts,* New York: McGraw-Hill, 1991, 2nd ed.
17. C. Papadimitriou, *The Theory of Database Concurrency Control,* Computer Science Press, 1986.
18. K. P. Eswaran et al., The notions of consistency and predicate locks in a database system, *Commun. ACM,* **19** (11): 1976, pp. 624–633.
19. J. N. Gray, Notes on database operating systems, *Lect. Notes Comput. Sci.,* **6**: 393–481, 1978.
20. P. A. Bernstein and N. Goodman, Timestamp based algorithms for concurrency control in distributed database systems, *Proc. 6th Int. Conf. VLDB,* 1980, pp. 285–300.
21. L. Lamport, Time, clocks and the ordering of events in a distributed system, *Commun. ACM,* **21** (7): 1978, pp. 558–565.
22. H. T. Kung and J. T. Robinson, On optimistic methods for concurrency control, *Proc. Conf. VLDB,* 1979.

23. D. Z. Badal, Correctness of concurrency control and implications in distributed databases, *COMPSAC Conf.,* pp. 588–593, 1979.

24. M. A. Casanova, Concurrency control problem of database systems, *Lect. Notes Comput. Sci.,* **116**: 1981.

25. A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts,* New York: McGraw-Hill, 1991, 3rd ed.

26. S. Greenberg and D. Marwood, Real time groupware as a distributed system: Concurrency control and its effect on the interface, *Proc. ACM Conf. CSCW'94,* pp. 207–217, 1994.

27. C. Sun et al., Achieving convergency, causality-preservation and intention preservation in real-time cooperative editing systems, *ACM Trans. Comput.-Hum. Interact.,* **5** (1): 1–42, 1998.

28. D. Jean, A. Cichock, and M. Rusinkiewicz, A database environment for workflow specification and execution, in Y. Kambayashi and K. Yokota (eds.), *Cooperative Databases and Applications,* Singapore: World Scientific, 1997, pp. 402–411.

29. H. Garcia-Molina and K. Salem, Sagas, *Proc. ACM SIGMOD Conf. Manage. Data,* 1987, pp. 249–259.

30. M. Nodine and S. Zdonik, Cooperative transaction hierarchies: A transaction model to support design applications, in A. K. Elmagarmid (ed.), *Database Transaction Models for Advanced Applications,* San Mateo, CA: Morgan Kaufmann, 1992, pp. 53–86.

31. G. Heiler et al., A flexible framework for transaction management in engineering environments, in A. Elmagarmid (ed.), *Transaction Models for Advanced Applications,* San Mateo, CA: Morgan Kaufmann, 1992, pp. 87–112.

32. A. Buchmann, M. T. Ozsu, and M. Hornick, A transaction model for active distributed object systems, in A. Elmagarmid (ed.), *Transaction Models for Advanced Applications,* San Mateo, CA: Morgan Kaufmann, 1992, pp. 123–158.

33. C. A. Ellis, S. J. Gibbs, and G. L. Rein, Groupware: Some issues and experiences. *Commun. ACM,* **34** (1): 39–58, 1991.

34. Y. Zhang et al., A novel timestamp ordering approach for co-existing traditional and cooperation transaction processing, to appear in *Int. J. Intell. and Cooperative Inf. Syst.,* an earlier version in Proceedings of 3rd IFCIS Conference on Cooperative Information Systems, New York, 1998.

35. M. H. Dunham and A. Helal, Mobile computing and databases: Anything new? *SIGMOD Rec.,* **24** (4): 5–9, 1995.

36. A. K. Elmagarmid, J. Jing, and T. Furukawa, Wireless client/server computing for personal information services and applications, *SIGMOD Rec.,* **24** (4): 16–21, 1995.

37. K. Ramamritham and P. K. Chrysanthis, *Advances in Concurrency Control and Transaction Processing,* Los Alamitos, CA: IEEE Computer Society Press, 1997.

Reading List

R. Alonso, H. Garcia-Molina, and K. Salem, Concurrency control and recovery for global procedures in federated database systems, *Q. Bull. Comput. Soc. IEEE Tech. Comm. Database Eng.,* **10** (3): 5–11, September, 1987.

P. A. Bernstein and N. Goodman, Concurrency control in distributed database systems, *Comput. Surv.,* **13** (2): 188–221, 1981.

J. Cao, Transaction management in multidatabase systems. Ph.D. thesis, Department of Mathematics and Computing, University of Southern Queensland, Australia, 1997.

U. Dayal, M. Hsu, and R. Latin, A transactional model for long running activities, *Proc. 17th Conf. Very Large Databases,* pp. 113–122, 1991.

C. A. Ellis, S. J. Gibbs, and G. L. Rein, Design and use of a group editor, in G. Cockton (ed.), *Enginering for Human Computer Interaction,* Amsterdam: North-Holland, 1990, pp. 13–25.

J. N. Gray, *Transaction Processing: Implementation Techniques,* San Mateo, CA: Morgan Kaufmann, 1994, pp. 207–217.

G. Kaiser and C. Pu, Dynamic restructuring of transactions, in A. Elmagarmid (ed.), *Transaction Models for Advanced Applications,* San Mateo, CA: Morgan Kaufmann, 1992.

M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1991.

Y. Kambayashi and K. Yokota (eds.), *Cooperative Databases and Applications,* Singapore, World Scientific, 1997.

C. Mohan et al., ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging, *ACM Transactions on Database Systems,* **17** (1): March, 1992, pp. 94–162.

C. Pu, G. Kaiser, and N. Huchinson, Split transactions for open-ended activities, *Proc. 14th Conf. Very Large Databases,* Los Angeles, 1988, pp. 26–37.

T. Rodden, A survey of CSCW systems, *Interact. Comput. Interdisc. J. Hum.-Compu. Interac.,* **3** (3): 319–353, 1991.

Y. Zhang and Y. Yang, On operation synchronization in cooperative editing environments, in *IFIP Transactions A-54 on Business Process Re-engineering,* 1994, pp. 635–644.

Y. Zhang
University of Southern Queensland
X. Jia
City University of Hong Kong

TRANSACTION-TIME DATABASES. See Temporal databases.