TRANSACTION MODELS AND ARCHITECTURES

E. Bertino[1] B. Catania[1] A. Vinai[2]

[1]Dipartimento di Scienze dell'Informazione

Università Milano

Via Comelico, 39

20135, Milano (Italy)

[2] Dipartimento di Informatica e Scienze dell'Informazione

Università di Genova

Via Dodecaneso, 35

16136, Genova (Italy)

Abstract

This paper discusses several issues related to transaction models and techniques to support such models. We first present the basic notions related to the concept of transactions, by introducing a simple model, known as flat transaction model, and discussing a number of transaction properties, known as ACID properties. We then describe mechanisms and techniques used by Database Management Systems (DBMS) to implement such properties. In particular, we discuss concurrency control and recovery techniques. We then briefly survey Transaction Processing Monitors, which are today widely used in large information systems to coordinate transactions spanning several DBMS and different software systems. We then conclude the paper by illustrating extensions to the flat transaction model and other developments in the area of transaction models and architectures.

1 INTRODUCTION

A database management system (DBMS) is an hardware/software system able to handle, in an organized and uniform way, large volumes of data representing information from real world application environments. DBMS are today a mature yet still evolving technology, whose application scope continuously widens. Today, DBMS support a large variety of applications, from financial and administrative applications to medical and scientific applications and to manufacturing applications. New applications arise every day, often demanding extensions to current DBMS technology.

The main tasks of a DBMS are to provide an efficient and suitable environment for accessing and storing data items, to protect data from failures and attempts to unauthorized accesses, and to ensure data quality. Failures may arise because of errors in the application programs or in the DBMS itself, or because of hardware crashes, such as disk crashes. If no proper mechanism is in place, failures may undermine data correctness and cause serious data losses. Another source of potential errors is in the concurrent executions of applications programs. Because of performance requirements and the high volume of operations typical of most applications, DBMS allow application programs to be concurrently executed. Because different programs may access the same data, accesses to data must be properly synchronized.

Because of the importance of data protection and correctness, a large effort in DBMS research and development has been devoted to devising proper techniques to deal with errors and failures. A key notion, underlying all those techniques, is represented by the notion of *transaction*. A transaction can be seen as a set of data operations representing a meaningful unit of work from the application point of view[1]. The execution of a transaction is in general characterized by a set of properties, known as *ACID* properties. Those properties are the key elements in ensuring data protection against errors and failures[2]. The development of algorithms, techniques and systems implementing those properties together with their theoretical formalization has characterized a large part of database research in both academic institutions and industrial laboratories.

[1] We elaborate more on different perspectives in Section 2.

[2] For completeness, it is important to note that data protection also involves preventing data accesses from unauthorized users and application programs; we do not deal with those issues here and refer the reader to [3].

Historically, transaction processing systems were first developed to support on-line applications. Those applications are characterized by short programs that are activated by users from possibly remote terminals. In general, such applications have large numbers of terminals (hundreds or thousands) and data are often concurrently accessed. Examples of those applications are airline reservation systems. Earlier transaction processing systems included IBM's Customer Information and Control Systems (CICS) and Transaction Processing Facility (TPF) [9]. More modern transaction processing systems have been developed for Unix platforms. They include Encina and Tuxedo [9]. Transaction processing systems were in parallel developed as part of DBMS. The most fundamental advancements in transaction processing for DBMS have been in the framework of relational systems, such as System R, System R* and Ingres. Basic concepts concerning transaction synchronization and recovery were developed within those systems. The continuing evolution of database technology has been, however, always demanding extensions and innovations to transaction processing techniques. This makes the area of transaction systems an important research area with a strong impact on commercial developments.

The aim of this paper is the formal definition of the concept of transaction, together with a survey of models and architectures supporting such concept. In particular, in Section 2 the basic concepts underlying transactions are introduced, whereas a simple transaction model is presented in Section 3. Section 4, 5, and 6 deal with several aspects related to transaction processing systems. Problems related to concurrency control are surveyed in Section 4, introducing several protocols ensuring database consistency when concurrent accesses to data are allowed. Section 5 deals with recovery, whose aim is to detect failures and restore the database to a consistent state. Isolation allows a system to give the illusion that each transaction is executed alone, even if transactions are executed concurrently. This topic is dealt with in Section 6. Section 7 presents a brief survey of transaction processing monitors, allowing the integration of different systems and the management of resources, so that applications spanning several systems and sources are executed accordingly to the transactional model. A brief outline of the most well-known extensions to the flat transaction model is then presented in Section 8. Finally, Section 9 presents some conclusions.

2 BASIC CONCEPTS

In general, users interact with a database system through special application programs called *transactions*. The term *transaction* is used in the database literature with different meanings:

1. A transaction is the request or input message activating the execution of a set of operation(s) on the database.
2. A transaction represents all the effects of the execution of a set of operation(s) on the database.
3. A transaction is the program executing a set of operations on the database.

Those different interpretations arise because of the different perspectives of users involved in transaction execution and management. End users only see the request and the reply and, consequently, think of transactions in these terms. Operators see the effects of the requested execution, so they take that view. System administrators deal with naming, security, installation and maintenance of transaction programs. They therefore often think of the transaction as the program source rather than the program execution. In this paper we adopt the second definition: *A transaction is a partially ordered set of read/write operations; it represents the effect on the database of the processing of programs executing functions required by the users.*

Transaction processing systems pioneered many concepts in distributed computing and fault-tolerant computing. They introduced the notion of transaction ACID properties - atomicity, consistency, isolation, and durability - as unifying concepts for fault-tolerant and correct computations in both centralized and distributed settings. A transaction can be considered as a collection of operations with the following properties:

1. *Atomicity* (also called all-or-nothing property): it refers to the fact that all the operations of a transaction must be treated as a single unit; therefore, either all operations are executed or none.
2. *Consistency*: it deals with the correctness of concurrently executing transactions. If executed alone, a transaction transforms a database from a consistent state to another consistent state. When transactions are executed concurrently, the DBMS must assure that the database consistency is

preserved as if each transaction were executed alone.

3. *Isolation*: it requires each transaction to observe a consistent database, that is, not to read intermediate results of other transactions.

4. *Durability*: it requires the results of a committed transaction be made permanent in the database in spite of possible system failures.

Example 1 *Consider a banking debit transaction on Mr. Money current account. Such transaction would consist of releasing money and updating the account. It is atomic if it never happens that only one operation is performed; for instance, the money is released but the account is not updated. It is consistent if the amount of money released is the same as the amount debited to the account. It is isolated if the transaction program has not to worry about other programs concurrently reading and/or updating the account (for example Mrs. Money making a concurrent deposit). And it is durable if, once the transaction has completed, the account balance exactly reflects the withdrawal.* □

The ACIDity properties of transactions are ensured by two classes of algorithms or *protocols*. The first class of protocols deals with *concurrency* whereas the second deals with *recovery*. Those protocols are implemented by specific subsystems within the DBMS. Concurrency control protocols are used to synchronize concurrent transactions. They, thus, ensure the C-property. Recovery protocols support the abstraction of failure atomicity, that is, they make sure that no incomplete transaction executions arise because of failures. In the event of a crash, a recovery protocol typically: (1) *undoes* the effects of the transactions that were executing, but had not yet completed, at the time of the crash; (2) *redoes* the transactions that had completed, but possibly had not yet installed their changes into the database, at the time of the crash. Therefore, recovery protocols ensure both the A-property and the D-property. Finally, the I-property is jointly ensured by the concurrency control and the recovery protocols. The concurrency protocol must make sure that no other transaction accesses a data item being modified by a transaction. The recovery subsystem must notify the concurrency control subsystem of when the transaction has completed, so that access the data items modified by the completed transaction can be given to waiting transactions.

Note, however, that the recovery subsystem uses in several situations the services provided by the concurrency control, for example when performing undoing and redoing activities. We have somehow simplified the above discussion for the sake of clarity.

3 A SIMPLE TRANSACTION MODEL

A large number of transaction models has been proposed and implemented. Most of them are extensions of a basic transaction model; transactions handled by the basic model are usually referred to as *flat transactions.*

Flat transactions are the simplest kind of transaction model. They have been used into all commercially available DBMS, and are now being introduced in operating systems and communication systems. The implementation techniques are well-known, and so are the limitations.

3.1 Flat transactions

A flat transaction is the basic block in organizing an application into atomic actions. The operations part of the same transaction are delimited by two special operations. The BeginWork operation starts a (flat) transaction. The CommitWork operation indicates to the DBMS that the transaction has completed its execution and wishes to install its updates into the database. If the Commit call successfully completes, the DBMS guarantees that all operations enclosed between BeginWork and CommitWork are executed according to the ACID properties. Durability in particular guarantees that nothing[3] can cause the performed updates to be lost. Alternatively, a transaction can terminate by issuing an Abort operation to indicate that some errors occurred during the execution and therefore all (partial) updates performed by the transaction must be removed from the database. Note that an application program may contain several transactions.

The above transaction model is called *flat* because there is only one layer of control by the application. Everything inside the pair *BeginWork/CommitWork* or *BeginWork/Abort* is at the same level; that is, the

[3]That is, nothing within the specification of the system. If more errors than the system was designed for occur at the same time, this guarantee does not hold.

transaction will either survive (commit), or it will be rolled back (abort).

A good characteristic of flat transactions is that they cover not only database operations: the ACID properties hold for everything executed between BeginWork and CommitWork.[4] This is particularly important when handling messages. As an example consider the message sent from a transaction to an automated teller machine - ATM. The ACID properties make sure that either the message will be sent (the cash will be dispensed), which means that the entire transaction will be successfully completed, or the transaction will fail, in which case no money will be dispensed. This general notion of transactions involving several subsystems, not necessarily only DBMS, is supported by specialized systems known as Transaction Processing (TP) Monitors, that we discuss in Section 7.

In the following, a formal definition of flat transaction is proposed to the reader.

Definition 1 *A flat transaction T_i is a set on which is defined a partial order $<_i$, where:*

- $T_i \subseteq \mathcal{I} \cup \{a_i, c_i\}$, *where:*
  1. *$\mathcal{I}$ is the power set of $\{read_i[x], write_i[x] \;/\; x$ is a data item$\}$;*
  2. *a_i indicates that the transaction was not successfully completed (i.e. it has been aborted);*
  3. *c_i indicates that the transaction was successfully completed (i.e. it has been committed);*
- $a_i \in T_i$ *iff* $c_i \notin T_i$;
- *if* $t=a_i$ *or* $t=c_i$, *then* $\forall$ *other operation* $p \in T_i$, $p <_i t$;
- *if* $read_i[x]$, $write_i[x] \in T_i$, *then either* $read_i[x] <_i write_i[x]$ *or* $write_i[x] <_i read_i[x]$.

$\diamond$

[4] A popular misunderstanding is that transactions are bound to databases. Although it is true that, historically, transactions have evolved from the database field, it must be clearly understood that they establish a general execution paradigm that ideally covers all the subsystems involved in their execution.

3.2 Flat transactions and SQL

SQL is the most well-known query language for relational databases. It was initially developed for System R[5]. SQL stands for *Structured Query Language*. It is also usually known with its former name, Sequel. Many SQL implementations allow SQL queries to be submitted from a program written in a general purpose language, such as Pascal, C, COBOL, PL/1 or Fortran. This extends the programmer's ability to manipulate the database. Embedded SQL is a slightly modified form of SQL than can be used within a host programming language. There is a large number of available commercial database systems that use embedded SQL to interact with the database; among them SQL/Data System (SQL/DS), DB2, ORACLE and Ingres.

When a programmer writes Embedded SQL code in a host language source file, any SQL statement must be preceded by *exec sql* statement. To support transactions, specific SQL statements are provided by the various DBMS. Such statements are invoked from within application programs and allow the application programmers to denote the start and ending of a transaction as well as to abort a transaction. In the following example, a very simple Embedded SQL program in C hosted by Oracle taken from [15] is presented, showing how a transaction is programmed at application level.

Example 2 *Consider the Embedded SQL program illustrated in Figure 1. The program prompts a user repeatedly for a customer identifier* `cid` *and replies by printing the customer name and discount, halting when the user inputs an empty string. An Embedded SQL program must be able to use normal program variables, usually referred to as host variables. In Figure 1, cust_name, cust_discnt and cust_id variables are declared to contain retrieved values for, respectively, customer name* `cname`*, discount* `discnt` *and* `cid`*, a value written by the user. In order to use these host variables in an Embedded SQL statement, they must first be declared to the precompiler. This is accomplished declaring variables between* `exec sql begin declare section` *and* `exec sql end declare section` *parenthesis.*

[5]The System R research project began at the IBM San Jose Research Laboratory in 1974. The goal of the project was to demonstrate the practicality of the newly proposed relational data model. A great number of fundamental concepts were established by this research, and several commercial products derived from System R research.

```
#include <stdio.h>                                  /* header for ORACLE datatypes */
#include <ctype.h>

exec sql include sqlca;                                    /* communication area */
int promptl(char[ ], char[ ], int);
char prompt[ ] = "Please enter customer identifier"

main()
{ exec sql begin declare section;                   /*declare SQL host variables */
     VARCHAR cust_id[5], cust_name[14];             /* ORACLE character strings */
     float cust_discnt;                             /* host var for discnt value */
     VARCHAR user_name[20], user_pwd[10];
  exec sql end declare section;

  exec sql whenever sqlerror stop;                  /* error trap condition */
  strcpy(user_name, arr, "poneilsql");
  user_name.len = strlen(cust_id, arr);
  strcpy(user_pwd, arr, "XXXX"); user_pwd.len = strlen("XXXX");
  exec sql connect :user_name
       identified by :user_pwd;                     /* ORACLE format: connect */
  while((prompt1(prompt, cust_id, arr, 4)) >= 0)    /* main loop: cid */
      {cust_id.len = strlen(cust_id.arr);           /* set lenght of cust_id */
       exec sql select cname,
            discnt into :cust_name, :cust_discnt    /* retrieve cname, discnt */
                 from customers where cid = :cust_id;
       cust_name.arr[cust_name.len] = ' ';          /* null terminated string */
       exec sql commit work;

       printf("Customer =  %s; discount = %f ", cust_name.arr, cust_discnt); }

  exec sql disconnect; }                            /* disconnect from database */
```

Figure 1: An example of Oracle Embedded SQL Program

Obviously, it is essential that host language variables used in Embedded SQL code have data type known to the database system.

At the beginning, an Embedded SQL program has to connect with the SQL DBMS and to the right database. This is possible through the `exec sql connect :user_name identified by :user_pwd` *statement. Note that colons used to identify the host variables as arguments for SQL syntax. In particular, the connect statement has the effect of starting the transaction, thus implements the* BeginWork *primitive. To disconnect from the database, the* `exec sql disconnect` *statement is used. Note that the statement at the end of main loop of main(),* `exec sql commit work`*, is used after a sequence of reads from a table, before any user interaction such as printing results. In the program of such example, potential errors in executing Embedded SQL are dealt. At the beginning of the program* `exec sql include sqlca` *allocates space for certain errors and statistics to be communicated by the database system monitor. The statement* `exec sql whenever sqlerror stop` *sets up an error trap condition to capture simple default behavior in dealing with erroneous calls. The effect of these statements is to stop the program if an error arises at runtime execution.* □

3.3 Limitations of flat transactions

The major restriction of flat transactions is that there is no way to either commit or abort parts of such transactions, or committing results in several steps. For a simple state transformations, such as debiting or crediting an account, an atomic unit of work is certainly appropriate. Those are the applications for which flat transactions were originally designed. However, many other applications exist, for which this model is too restrictive. Flat transactions have a simple structure which is not able to model more complex application structures. To support such applications, it could be desirable to provide the transaction model with more expressive power. This may lead to the risk of loosing one of the major virtues of flat transactions - simplicity - which makes ACID transactions so successful. Two examples below shows a fairly simple application structures that is not well supported by flat transactions. The first example refers to a trip planning; the latter refers to bulk updates executed by a banking transaction.

Example 3 *Suppose that Mr. Money wants to go from Pittsburgh, USA, to Finale Ligure, Italy. Since there are no direct flights between the two places, a number of connecting flights must be booked, perhaps some trains, and so on. The travel agent, using a transaction system, tries to find a good travel plan for Mr. Money. The transaction produced is the following:*

- *BeginWork*
- *book flight from Pittsburgh to London*
- *book flight from London to Milan, same day*
- *book flight from Milan to Genoa, same day*

Problem: there is no way to reach Finale Ligure from Genoa in the same day, except by a rental car. Mr. Money, however, prefers to travel at a lower cost.
What can the travel agent do in that situation? One solution might be to have Mr. Money to go from Milan to Nice (France) and then go by train from there. Or the situation might require Mr. Money to fly from Milan to Torino, or perhaps somewhere else.

The point is that, given a flat transaction model, the travel agent has only two choices: (1) Issue RollBackWork. This gives up much more of the previous work than needed. In particular, there is no need of undoing the reservation on the flight to London. (2) Explicitly cancel all reservations that are no longer useful. This solution might be very expensive if the amount of work is larger than that in the example above. □

What would be needed in such an application is a *selective* rollback. Rather than aborting the whole transaction, it should be possible to roll back only the last two reservations, maintaining the first. If such a feature is not available, the application designer is forced to use a programming style where updates no longer needed must be reversed by the application.

Example 4 *Let consider that, in our simplified banking system, at the end of a month a bank wants to modify one million of its accounts by crediting or debiting the interests. Suppose now that the whole work were done by only one flat transaction, whose code were approximately the following:*

UpdateAccounts()

BeginWork;

for each account A:

Update A;

CommitWork.

If all goes right, then all updates will be permanently stored at commit time. On the contrary, if, for example, 999,999 accounts have been modified and after that a system crash occurs, 999,999 updates have to be rolled back, according to the atomicity property. Of course, this is not desirable because the updates performed are not invalid in any sense, and moreover, rolling back 999,999 updates takes about as long as it took to do them the first time. On the other hand, it has to be guaranteed that each account tuple is updated exactly once. With flat transactions, the only way to assure this guarantee is the "bulk" transaction shown above. The solution of making each tuple a transaction in order to minimize the loss in case of system crash is not good, because in this case the system will have no information about which account was updated last before the crash (that is where it has to pick up work again after recovery[6]).

To overcome the limitations of the flat transaction model, several extensions to this model have been proposed expanding its scope, generalizing its semantics, and making it more powerful. We will discuss such extensions in Section 8.

4 CONCURRENCY CONTROL

Concurrency control is an important functionality that each multi-user DBMS should provide. The aim of a concurrency control mechanism is to ensure database consistency when there are concurrent accesses to data by several users; it uses synchronization techniques to handle concurrent transactions. The following

[6] In terms of concepts that are rigorously defined in Section 5, the system have information about which was the last successful update because it writes that update to the log. For the moment, it is sufficient to note that this is a "dirty" solution because the log is not part of a data model such as SQL. Using this dirty solution means exceeding the capabilities of a simple flat transaction environment. But this is exactly what new and extended transaction models are all about.

example explains the extreme importance of concurrency control in a database system.

Example 5 *Consider a database organizing some information about a bank clients's account. Suppose now that Mr. Money owns two accounts: a checking account and a deposit account, whose balances are, respectively, $100 and $1,000; the checking account is also owned by Mrs. Money. Mr. Money wants to transfer some funds from one account to the other. A bank employee therefore invokes a transaction T which transfers $150 from the deposit account to the checking account. At the same time, Mrs. Money performs a deposit, at another branch of the same bank, of the amount of $500 on the checking account. Let T' be the transaction invoked on behalf of Mrs. Money. The concurrent execution of those transactions may cause several problems.*

Execution 1

T	*T'*
Read dep_acct	
dep_acct = dep_acct - 150	
	Read check_acct
Write dep_acct	
	check_acct = check_acct + 500
Read check_acct	
	Write check_acct
	Commit T'
check_acct = check_acct	
Write check_acct	
Commit T	

In the above concurrent execution, because of an interference between T and T', T' has succeeded in its execution, but it does not produce any effect on the database, since the amount of money deposited by T' has been lost. Such concurrent execution is not correct and it should not allowed by the database management system. Note in particular that the updates performed by T' do not have any effect on the *database state. Such anomaly is called lost update.* □

4.1 Serializability

How should a valid concurrency control mechanism handle the situation illustrated in Example 5? The interleaving, in such example, produces an incorrect database state. Each transaction would produce a correct state when executed alone. In other words, this problem would not arise if T and T' were executed one after the other; such an execution is called *serial execution.* In general, it is not necessary to impose such restrictive conditions.

Example 6 *The following execution, although not serial, does not cause any anomaly.*

Execution 2

T	*T'*
Read dep_acct	
	Read check_acct
dep_acct = dep_acct - 150	
	check_acct = check_acct + 500
Write dep_acct	
	Write check_acct
	Commit T'
Read check_acct	
check_acct = check_acct + 150	
Write check_acct	
Commit T	

The above execution shows that, even though operations from different transactions interleave, the effect on the database is the same that would have been obtained executing T and T' sequentially. □

Executions proved to be equivalent to a serial execution are called *serializable.*

4.2 Concurrency control protocols

Different protocols have been proposed to ensure the serializability property for concurrent transactions. Among the proposed protocols, the most well-known are *two phase locking* and *timestamp ordering* protocols. Timestamping-based protocols assign a unique timestamp to each transaction. Each data item is associated with the timestamp of the last transaction reading/writing it. In the following, both protocols are surveyed.

4.2.1 Lock-based protocols

Locking-based protocols delay executions of conflicting operations, by locking on data items before performing read/write operations. In other words, a transaction can access a data item only if it has obtained the appropriate lock on that data item.

There are various modes in which a data item can be locked. The basic ones are the following:

1. *Shared.* If a transaction T has obtained a shared mode lock on item Q, then T can read, but not write, Q.

2. *Exclusive.* If a transaction T has obtained an exclusive mode lock on item Q, then T can both read and write Q.

Each transaction must thus acquire a lock in an appropriate mode on data item Q, depending on the type of operations it will perform on Q.

Given a set of lock modes, one can define a *compatibility function* on them as follows. Let A and B represent arbitrary lock modes. Suppose that a transaction T_i requests a lock mode A on item Q on which transaction T_j ($T_i \neq T_j$) currently holds a lock of mode B. If transaction T_i can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then mode A is said *compatible* with mode B. Such function can be represented by a matrix M, where M(i,j) = true if lock mode i is compatible with lock mode j; M(i,j) = false otherwise. Figure 2 shows the compatibility matrix M just described.

A transaction requires a shared lock on data item Q by Ls(Q) statement. In similar way, an exclusive lock is required using Lx(Q) statement. A data item Q is unlocked through Un(Q) statement.

Compatibility matrix	Shared	Exclusive
Shared	true	false
Exclusive	false	false

Figure 2: Lock compatibility matrix M. M(i,j) = true if lock mode i is compatible with lock mode j.

The *two-phase locking protocol* imposes each transaction to require locks and unlocks on data items in two different phases:

1. *Increasing phase.* A transaction, during this phase, can only obtain new locks, but it cannot release any lock.
2. *Decreasing phase.* A transaction, during this phase, can only release locks, but it cannot obtain further lock.

Initially, a transaction performs an increasing phase, during which it acquires all locks necessary to its operations. When the transaction releases one lock, it enters in a decreasing phase, and new lock requests will not be granted.

As previously said, two phase locking protocol ensures serializability. It is, however, deadlock-sensitive[7].

Example 7 *Let T_i and T_j be two transaction acting on data items a, b and c as follows:*

Execution 3

T:

Read(a)

[7]A system is said to be in *deadlock* if exists a set of transactions such that each transaction in the set is waiting for another transaction, belonging to the set, to continue its execution (see Subsection 4.4).

Read(b)

Read(c)

Write(a)

Commit T

T':

Read(a)

Read(b)

Commit T'

If the two-phase locking protocol is used, then T_i must lock a in exclusive mode. Therefore, no concurrent execution of the two transactions is possible. The only possible processing is the serial one. Notice, however, that T_i needs an exclusive lock on data item a only at the end of its execution, when it writes a. Thus, if T_i could initially lock data item a in a shared mode, changing it into exclusive later, it could be allow more parallelism, since T_i and T_j could access data item a and b simultaneously. □

The above observation leads to a modification to the basic two-phase locking, allowing *lock conversion.* That is, a particular mechanism is provided to *promote* a shared lock into an exclusive lock and to *regress* an exclusive lock into a shared lock. Conversion cannot occur arbitrarily. It can only occur during the increasing phase; a lock can be regressed only during the decreasing phase.

4.2.2 Timestamp ordering protocol

In the above described locking protocols, the order among conflicting transactions is determined during the increasing phase, when two transactions lock the first data item they have in common, in incompatible mode. Another way to establish the serializability order is selecting an order in advance, among each pair of transactions. The most common approach to implement such scheme, is the use of a *timestamp ordering.*

Each transaction T_i, in the system, is assigned a unique timestamp, denoted by TS(T_i). This label is assigned by the database system before the transaction T_i starts its execution. If a timestamp has been assigned to T_i, and a new transaction T_j enters the system, then TS(T_i) < TS(T_j). Two simple methods are used to implement such scheme:

1. Using the system clock as timestamp. The transaction timestamp is the value that clock has when the transaction enters the system.

2. Using a logic counter that is increased after the assignment of a new timestamp. The transaction timestamp is the value of the counter when the transaction enters the system.

Transactions timestamps establish a serializability order. Thus, if TS(T_i) < TS(T_j), the system must ensure that the produced schedule[8] is equivalent to a serial schedule in which T_i appears before T_j. To implement this scheme, two timestamp values are associated with each data item Q:

1. *W-timestamp(Q)*. It denotes the greatest timestamp among the timestamps of all transactions that succeeded in executing a write(Q) operation.

2. *R-timestamp(Q)*. It denotes the greatest timestamp among all transactions that succeeded in executing a read(Q) operation.

These values are updated whenever a new write(Q) or read(Q) operation is performed.

The timestamping ordering protocol ensures that all conflicting read/write operations are executed in the timestamping order. The protocol is organized as follows:

1. Suppose that a transaction T_i requires to perform a read(Q) operation,

   (a) If TS(T_i) < W-timestamp(Q), this means that T_i needs to read a value of Q that has already been overwritten. Then the read operation must be rejected and T_i must be rolled back.

   (b) If TS(T_i) $\geq$ W-timestamp(Q), it follows that the reading may be executed, and R-timestamp(Q) is set to the maximum value between R-timestamp(Q) and TS(T_i).

2. Suppose that transaction T_i requires to perform a write(Q) operation,

[8] A schedule represents the order in which operations are executed in the system, according to the chronological order.

(a) If $\mathrm{TS}(T_i) < \text{R-timestamp}(Q)$, this means that the value of data item Q, which T_i is modifying, has previously been read by some other transaction and then is not possible to perform any update.

(b) If $\mathrm{TS}(T_i) < \text{W-timestamp}(Q)$, this means that T_i is trying to write an obsolete value of Q. The write operation is therefore simply ignored.

(c) Otherwise, the write operation is performed, and W-timestamp(Q) is set to the maximum value between W-timestamp(Q) and TS(T_i).

Each transaction that is rolled back, owing to the concurrency control, is assigned a new timestamp and it is restarted. The timestamp ordering protocol ensures serializability. This follows from the fact that conflicting operations are processed in timestamp order. Such protocol ensures also the absence of deadlocks, since transactions never wait.

The above timestamp ordering protocol may however result in cascading rollbacks. To overcome this problem, one must ensure that a transaction can read only values modified by transactions successfully completed. This can be achieved by associating a *commit bit* with each transaction T_i, and a pointer to that bit from each data item written by T_i. Initially, the commit bit is set to "false". It will be set to "true" only after the commit of the transaction. Therefore, a transaction wishing to read a data item can do it immediately if the commit bit is set to "true"; otherwise it must wait until this happens.

4.3 Multiversioning

The concurrency control approaches discussed so far ensure serializability, by delaying an operation, or aborting the transaction that requested the execution of the operation.

In multiversioning protocols, each write(Q) operation generates a new version of Q. When a read(Q) operation is issued, the system selects one of Q versions. The concurrency control protocol must ensure that the selection of the version guarantees serializability. It is moreover crucial, for performance reasons, that a transaction can easily and quickly determine which data version must read.

The most common technique used for multiversion schemes is timestamping. Each transaction T_i is assigned a unique static label, denoted by TS(T_i). This label is assigned as in the timestamp ordering protocol.

Each data item Q has a sequence of versions $<Q_1, Q_2, \ldots, Q_m>$. Each version contains three components:

1. *Content*: it is the value of version Q_k.

2. *W-timestamp(Q_k)*: it denotes the timestamp of the transaction that created version Q_k.

3. *R-timestamp(Q_k)*: it denotes the greatest timestamp among the timestamp of all transactions that successfully read version Q_k.

A transaction T_i creates a new version Q_k of a data item Q, requiring the execution of a write(Q) operation. Content field of version stores the value written by T_i. W-timestamp and R-timestamp are initialized with TS(T_i) value. R-timestamp is updated whenever a transaction T_j reads the content of Q_k, and R-timestamp(Q_k) $<$ TS(T_j).

The scheme works as follows. Suppose that a transaction T_i wishes to execute a read(Q) or write (Q) operation. Let Q_k be a version of Q such that W-timestamp(Q_k) is the timestamp lower or equal to TS(T_i).

1. If T_i performs read(Q), the returned value is content of version Q_k.

2. If T_i performs write(Q), and if TS(T_i) $<$ R-timestamp(Q_k), T_i is rolled back; otherwise a new version Q_k is created.

The motivation for the first condition is clear: a transaction always reads the most recent version. The latter condition forces a transaction to abort if it is "too late" in performing a write. More precisely, if T_i tries to write a version that some other transaction could have read, it is not correct to allow the execution of such write.

The multiversioning protocol has the nice feature that a read request never fails and never waits. In real database systems, where reading is more frequent than writing, this advantage can be very significant. It has, however, three drawbacks. First, reading a data item requires also updating the R-timestamp field, and this may require two disk accesses rather than one. Second, transactions conflict are solved through rollbacks rather than waits. This could be very expensive. Finally, cascading rollbacks may arise. To avoid cascading rollbacks, the same techniques of the timestamping concurrency control protocol can be adopted.

4.4 Deadlock detection and resolution

As already mentioned, a system is in deadlock if a set of transactions exists such that each transaction in the set waits another transaction in the same set to continue its own processing. More precisely, a set of waiting transactions $\{ T_0, T_1, \ldots, T_n\}$ exists such that T_0 waits for a data item that is currently held by T_1; T_1 waits for a data item that is currently held by T_2 ; ...; T_{n-1} waits for a data item that is currently held by T_n, and T_n waits for a data item that is currently held by T_0. The only solution to this undesirable situation is some drastic action, as, for example, rolling back one among the transactions involved in the deadlock.

There are two main solutions to the deadlock problem. A deadlock prevention protocol can be used to ensure that the system never enters a deadlock state. Alternatively, one may allow the system to enter a deadlock state, and then recover from the deadlock using a deadlock detection and resolution scheme. Although the prevention protocols were conceptually very simple, it is not actually used in any real system, as it may result in useless rollbacks. The interested reader can find a description of the most well-known prevention protocols in [11].

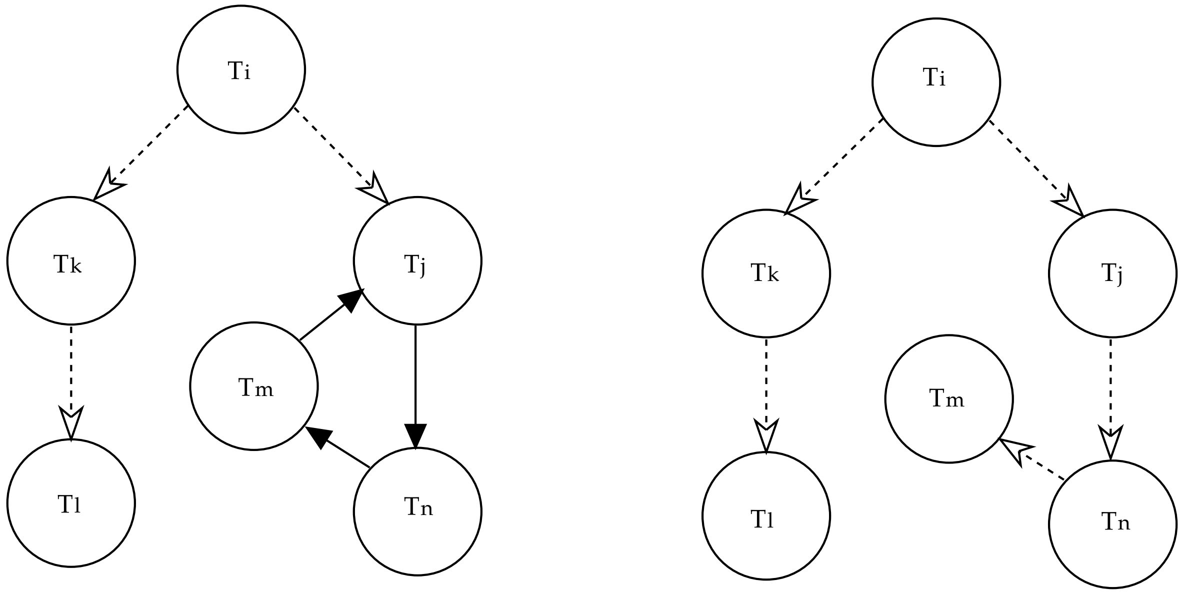
In the following, deadlock detection and deadlock resolution algorithms are surveyed.

4.4.1 Deadlock detection

An algorithm analyzing the system state is periodically invoked to detect if any deadlock has occurred. The algorithm has to perform two main tasks:

1. Maintaining information about the current data allocation to the transactions, and about pending data requests.

2. Solving the deadlock when it is actually found.



(a) Wait-for graph with a cycle

(b) Wait-for graph with no cycles

Figure 3: Example of cyclic and acyclic graphs

Deadlocks can be precisely characterized with a direct graph, called *wait-for graph.* This graph consists of a pair G = (V, E), where V is a set of vertex and E is a set of edges. Each element belonging to E is a ordered pair (T_i, T_j). If (T_i, T_j) $\in$ E, then there is a direct edge from T_i to T_j, and this means that T_i needs, to continue doing its work, that T_j releases a certain data item.

When T_i requires a data item currently held by a transaction T_j, an edge (T_i, T_j) is inserted in the wait-for graph. This edge is removed only when T_j releases the data item requested by T_j.

A deadlock occurs in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is blocked. To detect deadlocks, the system maintain the wait-for graph and periodically invokes an algorithm looking for cycles in the graph. Figure 3 illustrates two examples of cyclic and acyclic wait-for graph. While the wait-for graph on the left side contains a cycle, emphasized with bold arrows, and therefore a deadlock occurs, the wait-for graph on the right side does not contain any cycle. Thus, in this case, no deadlock is detected.

4.4.2 Deadlock resolution

If the detection algorithm signals a deadlock, the system must solve the blocking situation. The most common solution is to roll back one or more transactions. To do so, three different issues need to be addressed:

1. *Selection of a victim.* Given a set of blocked transactions, one must determine which transactions have to roll back in order to break the deadlock. Transactions to roll back are those with the minimum cost. Unfortunately, the term "minimum cost" is not precise. Several factors are involved in the rollback cost of a transaction:
   (a) How many computation the transaction has already performed, and how much longer the transaction will compute, before completing.
   (b) How many data items the transaction has used.
   (c) How many data items the transaction still needs, before completing.
   (d) How many transactions would be involved in the transaction rollback.
2. *Rollback.* Once the transaction victim is identified, one must be determined how far the transaction should be rolled back. The simplest solution is to perform a total rollback, that is, rolling back the transaction at its beginning and restarting it. It could be better, however, to roll back only the part of the transaction which is sufficient to break the deadlock. This method requires the system to maintain additional information about the state of the executing transactions (see Section 8).
3. *Starvation.* If the choice of the victim is cost-based, it may happen that the same transaction is always selected as victim. This means that the transaction can never complete its execution. Clearly, it must be ensured that a transaction is selected as victim only a small number of times. The most common solution is including in the cost the number of times the transaction has been rolled back.

5 COMMIT PROTOCOLS AND RECOVERY

5.1 Faults

A computer system, as other electrical or mechanical devices, is subject to failures. Techniques for preventing errors and recovering from them have been widely investigated in areas such as distributed systems and telecommunication systems. The specific technique to adopt depends on the type of foreseen failures. When discussing such techniques in the database context, it is useful to distinguish three common causes of failures:

1. *Disk crash.* Information residing on disks is lost. The most common types of such failure are head crashing and errors occurring during a data transfer operation.

2. *Power failure.* Information stored in main memory and registers is lost.

3. *Software errors.* Some results generated while processing the transaction can be incorrect and can return the user erroneous results; moreover the database could enter an inconsistent state. The most common types of such failure are logical errors (the program cannot continue its normal execution because of certain internal conditions, such as an erroneous input, data not found, overflow, etc.) and system errors (the system has entered in a undesirable state, so the program cannot continue its normal execution, but it can be, however, reexecuted later.)

Several kinds of failures can occur in a system, each of which needs to be handled in a different way. The simplest failures to handle are those that do not result in information loss. Obviously, the more difficult to handle are the ones resulting in information loss.

A fundamental component of any database system is the *recovery subsystem*, responsible for detecting failures and restoring the database to the (consistent) state existing before the failure. Recovering a database means avoiding data loss and inconsistency. It is useful to classify the various storage media with respect to how the information they store is affected by the various types of failures:

1. *Volatile storage.* Information residing in volatile storage is lost upon system crashes. Examples of such types of storage are main memory and cache memory.

2. *Nonvolatile storage.* Information residing in nonvolatile storage usually survives system crashes. Examples of such type of storage are disk and magnetic tape. However, both media are subject to media failures (for example head crashes), that can result in a loss of information.

3. *Stable storage.* Information residing in stable storage is never lost. To implement an approximation of this kind of media, information is duplicated in several nonvolatile storage media having independent probability of failure.

5.2 An Abstract Model of Transaction Executions

In order to discuss recovery techniques, it is necessary to specify in a more appropriate way, what "successful completion" of a transaction means. To do this, one establishes a simple abstract model. According to this model, a transaction is always in one of the following states:

1. *Active*, the initial state.

2. *Partially committed*, after the execution of the last instruction.

3. *Failed*, after discovering that the transaction cannot normally continue.

4. *Aborted*, after rolling back the transaction and recovering the database to the state that existed before the transaction started.

5. *Committed*, after "successful" completion.

Figure 4, adapted from [11], shows the state diagram. A transaction starts in the *active* state. After executing the last instruction, it enters the *partially committed* state. Now, the transaction has completed its processing, but a failure is still possible, since its actual output cannot yet be permanently installed on nonvolatile storage; thus, a hardware failure could hamper the successfully completion.

Particular care is necessary if *external observable writes* occurs; these are not erasable writes (for example those on terminals or on printers). Several systems force these writes to be executed only after the transaction enters a *commit* state. A way to implement this approach is temporarily storing values associated with external write in a nonvolatile storage, and executing actual writes only at commit

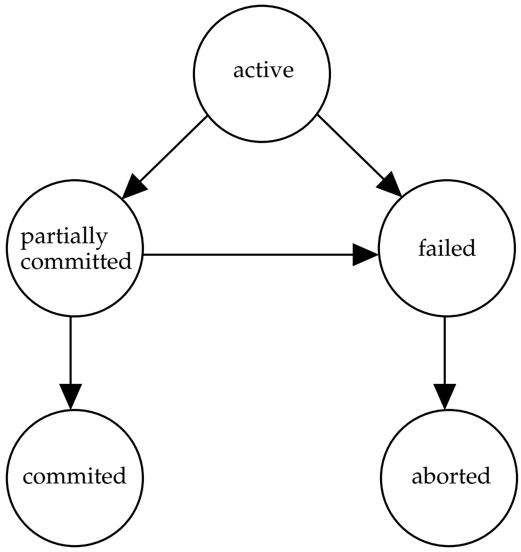


Figure 4: State diagram of a transaction

confirmation. A committed transaction will always be able to complete external writes, except when a disk failure occurs. If one wants to assure that a disk failure does not prevent a succeeding external observable write, stable storage could be used.

A transaction enters in the *failed* state after it is determined that the transaction cannot continue its normal processing. Such transaction has to be eliminated by a rollback operation. Once the rollback is performed, the transaction enters the *aborted* state. Now the system has two options:

1. *Reexecute the transaction.* This can take place only if the transaction is aborted because of some software or hardware errors not produced by the transaction internal logic. A reexecuted transaction is considered a new transaction.

2. *Eliminate the transaction.* This usually takes place if there are internal errors that can only be corrected by rewriting the application program.

A transaction enters the *commit* state if it is *partially committed* and it is guaranteed that it will never be aborted.

Example 8 *Consider the usual simplified banking system and be T a transaction transferring \$100 from account A to account B, whose balance are, respectively, \$1,000 and \$15,000; during T processing, after*

A modification, and before B modification, a system crash occurs. Memory contents are lost, then it is not possible to determine what happened to the transaction. Possible actions are:

1. *Reexecuting T. In this way, a state will be produced in which A assumes the value of \$800, and B assumes the value of \$15,100.*
2. *Not reexecuting T. In the current state A assumes the value of \$900, and B has still the value of \$15,000.*

In both cases the crash causes an inconsistent database state; this trivial recovery scheme in not properly working. □

The problem lays in having modified the database without having the assurance that the transaction will indeed commit.

Next subsection will introduce a recovery scheme widely used to handle logical errors, system errors and system crashes.

5.3 Log-based mechanisms

The idea of recovery with log is very simple: during a transaction processing all write operations are stored in a particular file, managed by the system, called *log file*. Log information is used if failures occur before the transaction completion. If a system crash occurs before a transaction completes, or if the transaction fails, log information is simply ignored. Conceptually, the log can be thought of as a sequential file. The actual implementation uses multiple physical files to make record storage easy. Each record inserted in the log is assigned a unique identifier (called log sequence number, LSN). LSN are generated as an increasing sequence. Typically, they are logical addresses of the pertinent log record.

The nonvolatile version of log is stored in stable storage (for example, on disk); two identical copies of log are usually kept on different devices.

Whenever log records are written, they are placed only on log file buffers in volatile memory. At commit time all log records, up to LSN, are written in sequences of log pages, on stable storage. This operation is said "forcing" the log up to LSN. The system, besides forcing log because of activities of buffer and transaction management, can periodically force log buffer pages as they fill up.

The most well-known recovery mechanisms with log are the *incremental log with deferred updates* and the *incremental log with immediate updates*.

5.3.1 Incremental log with deferred updates

During a transaction processing, all writing operations are deferred until the transaction enters the *partially committed* state. All updates are stored in the log file.

The execution of a transaction T_i proceeds as follows: before T_i starts its execution, a record $<T_i$, starts$>$ is written on the log. During the processing, any $write[x]$ operation results in a new log record. Such record consists of the following fields:

1. the transaction name (i.e. T_i);
2. the data item name (i.e. x);
3. the new data value.

Finally, when T_i enters the *partially committed* state, the record $<T_i$, commits$>$ is written on the log. The record associated with a transaction, in the log, are used to perform deferred writes.

Since a failure can occur while the updates are actually executed, all log records are written on stable storage. Once done, updates can take place. At this point, the transaction enters the *committed* state.

By using the log, the system can handle any failure involving the loss of information on volatile memory. The recovery scheme uses the redo(T_i) procedure, which sets all data items updated by T_i to the new values.

The redo operation must be *idempotent*, that is, multiple execution of the operation are equivalent to a single execution. This is required to guarantee a correct behavior even if a failure occurs during the recovery procedure.

After a failure, the recovery subsystem analyses the log to determine which transactions have to be reexecuted. A transaction T_i needs to be reexecuted if the log contains both $<T_i$, starts$>$ and $<T_i$, commits$>$ records. Therefore, if a system crash occurs after the transaction completion, the information stored on the log is used to recover the database state to a consistent state.

Consider, finally, the case in which a second system crash occurs while the system is performing the recovery procedure that restored the state after the first crash. Some changes could have been made in the database, because of the redo operation, or no changes could have been made. When the system comes up after the second crash, the recovery procedure proceeds exactly like the first one. For each $<T_i$, commits$>$ found in the log, a redo operation in performed. In other words, the recovery actions restart from the beginning. Since redo writes values on the database that are independent by the values currently on the database, the result of a second attempt of redo, is the same as if redo had succeeded the first time.

5.4 Incremental log with immediate updates

Another recovery technique is based on the application of all updates directly to the database and the use of an *incremental log* with all changes performed by the database. If a failure occurs, the log information is used to recover the system state to a consistent state.

Before a transaction T_i starts, all *write[x]* operations must be preceded by the writing of a new log record. Each of such records consists of the following fields:

1. the transaction name (i.e. T_i);
2. the data item name (i.e. x);
3. the data item old value;
4. the data item's new value.

When T_i enters the *partially committed* state, a $<T_i$, commits$>$ record is written on the log.

Since the log information is used to rebuild the database state, the actual update on the database cannot be allowed before the pertinent log record is written on stable storage.

Also with this approach, the system can manage any failure not resulting in loss of information from nonvolatile memory. The recovery scheme uses two recovery procedures:

- undo(T_i), which restores the values of all data items updated by T_i to the old values;

- redo(T_i), which sets the values of all data items updated by T_i to new values.

The set of data items updated by T_i and its old and new values can be found in the log.

Redo and undo operations must be idempotent to guarantee a correct behavior even if a failure occurs during the recovery process.

After a failure, the recovery scheme consults the log to determine which transactions needs to be reexecuted, and which must be rolled back. The procedure is the following:

1. the transaction T_i needs to be rolled back if the log contains the $<T_i$, starts$>$ record, but it does not contain the $<T_i$, commits$>$ record;

2. the transaction T_i needs to be reexecuted if the log contains both the $<T_i$, starts$>$ and $<T_i$, commits$>$ records.

5.5 Checkpoints

When a crash system occurs, it is necessary to consult the log to determine which transactions must be reexecuted and which must be rolled back. In general the entire log file must be scanned. This leads to two problems:

1. The search time is high, and it impacts system performance.

2. Many transactions needing to be reexecuted have already actually written their updates on the database and they do not need *really* to be reexecuted.

To overcome this drawback, the *checkpoints* mechanism is used. During the execution, the system maintains the log according to one of the above approaches. Besides, the system periodically performs the following sequence of actions:

1. it forces all log records resident in main memory to stable storage;

2. it forces all buffer blocks to disk;

3. it forces the $<$ checkpoint $>$ record to stable storage.

With the checkpoint mechanism one can refine the recovery scheme. After a failure occurs, the recovery scheme scans the log to find the last transaction T_i that started its execution before last checkpoint took place. Such transaction can be found searching backwards in the log the first $<$ checkpoint $>$ record and then, going on, the subsequent $<T_i$, starts$>$ record.

As the transaction T_i is identified, redo and undo (only in the case of incremental log with immediate updates) operations must be applied to T_i and to all transactions T_j that started their execution after T_i. The following steps are required:

1. for each transaction T_k such that the $<T_k$, commits$>$ appears in the log, redo(T_k) must be executed;
2. for each transaction T_k such that the $<T_k$, commits$>$ does not appear in the log, undo(T_k) must be executed.

5.6 ARIES

A recovery method particularly simple and efficient was introduced by C. Mohan et al. [13], and is called ARIES (Algorithm for Recovery and Isolation Exploiting Semantics). To ensure data and transaction recovery, ARIES stores in a log the progresses of each transaction, and all actions determining changes in data.

It is a recovery method based on the incremental log with immediate update approach. More specifically, ARIES uses a log protocol, widely accepted, known as the *Write Ahead Logging* protocol. In systems WAL-based, an updated page is written on the same nonvolatile memory location from where it was read. This means that an "update on place" is performed on nonvolatile memory.

The WAL protocol enforces the fact that the log records representing changes to data must be already in stable storage before the modified data replace the previous version on nonvolatile storage. That is, the system does not write an updated page on nonvolatile memory until the undo portions of the log records describing the updates to the page are written on stable storage. To increase the reliability of this protocol, systems using WAL recovery store in each page the Log Sequence Number (LSN) of the log record describing the most recent update executed on that page. In many WAL-based systems, updates executed during a rollback are kept on the log by means of *Compensation Log Records* (CLR). The eventuality of

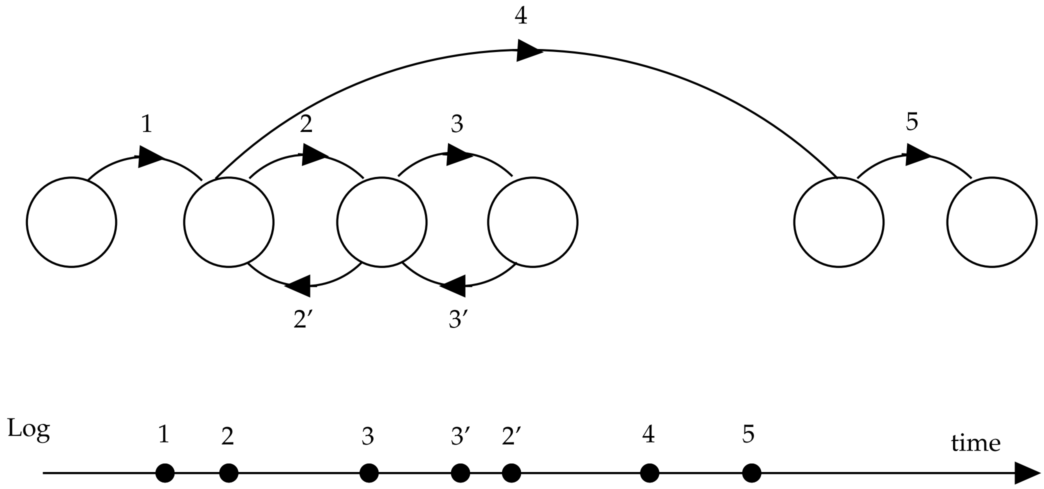


Figure 5: Example of partial rollback

never rolling back a CLR's update depends on the particular system. As it will be shown later, in ARIES a CLR update is never rolled back and then CLR records are only redo log records, that is, they contain only redo information.

ARIES also stores, typically using CLR records, all updates performed during total (and partial) rollbacks under normal and restarting processing. Figure 5, adapted from Ref. [13], gives an example of partial rollback where a transaction, having executed three updates (respectively 1, 2 and 3), rolls back two of them (updates 2 and 3), and then goes on. Because of the undo of the two updates, 2' and 3' CLR records are written. At the bottom of Figure 5, the execution order of updates is depicted.

By properly chaining CLRs to log records written during forward processing, a limited amount of logging activity needs to be executed during rollbacks, also in case of repeated failures during restarting or within nested rollback[9].

The reader may note in Figure 6 (adapted from Ref. [13]) that rolling back a log record results in a new CLR record. Such record, besides containing a description of the compensating action (for redo purposes), contains a pointer (UndoNxtLSN) to the just rolled back log record predecessor. Predecessor information is quickly available, since each log record, including CLRs, contains a pointer (PrevLSN) to

[9]A nested rollback takes place if a total rollback occurs after a partial one, whose termination point is a point in the transaction preceding the termination point of the first rollback.

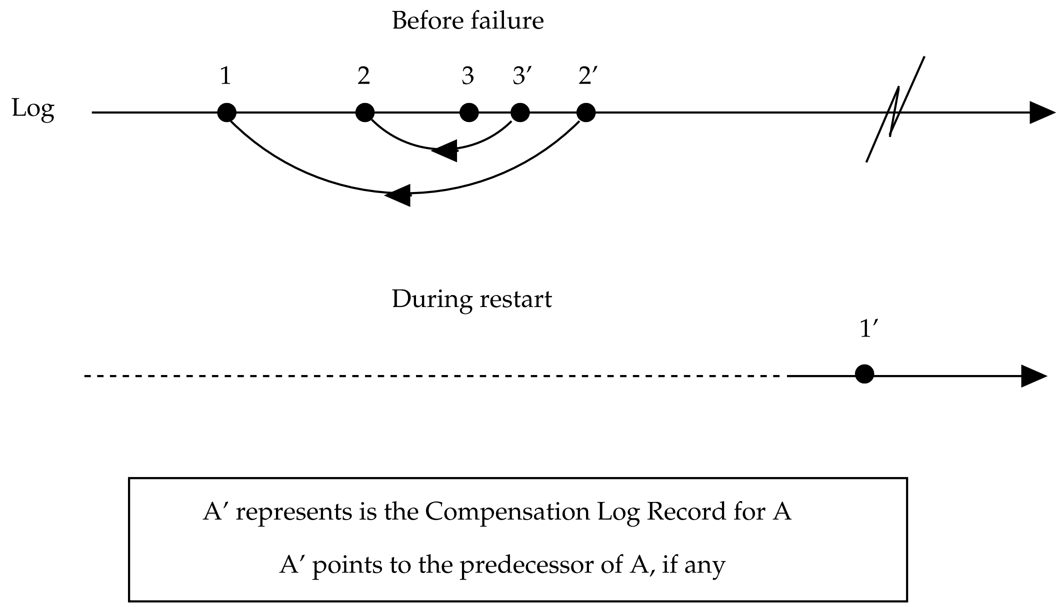


Figure 6: ARIES techique to avoid compensating compensations and duplicated compensations

the most recent previous log record written by the same transaction. UndoNxtLSN allows to precisely determine how much of the transaction has not been undone so far. In Figure 6, log record 3', which is the CLR for log record 3, points to log record 2, which is the predecessor of log record 3. Therefore, during transaction rollback, the UndoNxtLSN field of the most recently written CLR, keeps track of rollback progresses. It tells the system from which point to continue, if a system failure were to interrupt the completion of rollback or if a nested rollback were executed. It allows the system to bypass log records that have already been rolled back. Since CLRs are available to describe which actions are actually performed during the original action undoing, the undo action does not need to be the exact reverse of the original action. Therefore, a form of logical undo is possible supporting a high degree of concurrency.

ARIES uses a single LSN on each page to track the page state. Whenever a page is updated and a log record is written, the LSN of the log record is placed on the page_LSN field of the updated page. By tagging this page with the LSN allows ARIES, for restart and recovery purposes, to precisely track the page state with respect to the logged updates for that page. This allows ARIES to support innovative locking modes, using which, before confirming an update on a record field on behalf of a transaction, another transaction may be authorized to modify the same data item.

Periodically, during normal processing, ARIES performs checkpoints. Checkpoint records identify active transactions, their states, and the most recently written LSNs. Moreover they identify updated data items in the buffer. The last information is needed to determine from where the redo step must start its processing. During recovery, ARIES first scans the log, starting from the last checkpoint, until the end of the log. During this *analysis step* information on updated pages and on transactions that were executing when the checkpoint was performed, are collected. The analysis step uses information on updated pages to find the starting point to scan the log for the immediately subsequent redo step. The analysis step, moreover, determines all transactions that must be rolled back during the undo phase. For each active transaction also the most recently written LSN log record is determined. Then, during the redo step, ARIES "replays the history" with respect to updates logged on stable storage, but whose effects on the database pages were not reflected on nonvolatile storage before the failure occurred. This simply reestablishes the database state existing when the failure occurred. A log record update is performed if page_LSN field in the affected page is lower that LSN log record.

The next step is the undo step, during which all updates are undone in reverse chronological order, in a single log scanning. This is done keeping the maximum of the LSNs of the next log record to be processed, for each of the transactions yet to be completely undone, until no transaction remains to be undone. Unlike what happens during the redo step, the undo processing is not a conditional operation. That is, ARIES does not compare page_LSN fields of the affected page with LSN log record to decide whether to perform the undo operation or not. When it finds a non-CLR record during the undo step, if it is a undo-redo log record or a only-undo, then the updates in the record must be performed. In any case, the next record to process for such transaction is determined by examining PrevLSN field of the non-CLR record. Since CLRs are never undone, when a CLR is encountered during undo, it is used to determine next log record to process, using CLR UndoNxtLSN field.

Finally, in the case of transactions that were already being undone at time of failure, ARIES will only undo the actions not yet undone. This is possible since for such transaction, history has been replayed and since last CLR written for each transaction points (directly or indirectly) to next non-CLR record to be undone.

6 ISOLATION LEVELS

One of the key properties that transactions must preserve is *isolation*. In a shared database, a completely isolated system gives the illusion to each transaction that it is being executed alone all by itself, even though transactions are executed concurrently. Providing strict isolation may, however, reduce the performance. There are many applications for which it is acceptable to read not completely consistent data. To this purpose, many DBMS offer different degrees - 0, 1, 2, or 3 - of isolation to transactions. Degree 0 isolation, also known as *chaos*, is least restrictive among all. This degree of isolation does not allow more than one transaction to simultaneously update a data item. However, a transaction may read dirty data, that is, data being modified by another transaction and not yet committed by this transaction. Moreover, this degree of isolation does not prevent the updates performed by a transaction from being overwritten by another transaction before the completion of the first transaction. Degree 1 isolation, also called *browse*, does not allow data updated by uncommitted transactions to be overwritten. However a transaction may still perform dirty reads. Degree 2 isolation, known also as *cursor stability*, in addition to providing degree 1 isolation, restricts transactions to read only committed data. Degree 3 isolation, also called *repeatable reads*, is the most restrictive among all and provides complete isolation. With degree 3 isolation, a transaction that wants to read/write a data item must wait for the completion of all other transactions that previously wrote/read such data item. By allowing transactions to specify a low degree of isolation, a system can improve the response time, although at the expense of consistency.

In the remainder of this section, we first formally define the various isolation degrees. Then we show how they can be achieved by using the lock-based concurrency control protocols. Although a transaction may perform either commit or abort operations, the analysis in this section is limited to those executions containing only committed transactions. Isolation degrees may be described with histories, that can be defined as follows.

Definition 2 *A history H on $T = \{T_0, \ldots, T_n\}$, where $\forall T_i$ $i = 0, \ldots n$ is a flat transaction, is a partial order with ordering relation $<_H$, where:*

- $H = \bigcup_{i=0}^{n} T_i$ *and*

- $<_H \supseteq \bigcup_{i=0}^{n} <_i$

With degree 0 isolation, a transaction is not allowed to update a data item, while another transaction is updating the same data item. In terms of the history, any two write operations on the same data item by two different transaction must be ordered. Formally:

Definition 3 *A history H is a degree 0 isolation history if* $\forall\ w_i[x],\ w_j[x] \in H$, *either* $w_i[x] <_H w_j[x]$ *or* $w_j[x] <_H w_i[x]$.

In similar way other isolation levels can be defined:

Definition 4 *A history H is a degree 1 isolation history if*

1. $\forall\ w_i[x],\ w_j[x] \in H$, *either* $w_i[x] <_H w_j[x]$ *or* $w_j[x] <_H w_i[x]$, *and*
2. $\forall w_i[x] <_H w_j[x],\ c_i <_H w_j[x]$.

Definition 5 *A history H is a degree 2 isolation history if*

1. $\forall\ o_i[x],\ o_j[x] \in H$ *two conflicting operations, either* $o_i[x] <_H o_j[x]$ *or* $o_j[x] <_H o_i[x]$, *and*
2. $\forall w_i[x] <_H o_j[x],\ c_i <_H o_j[x]$.

Definition 6 *A history H is a degree 3 isolation history if*

1. *H is serializable (as defined in [2]), and*
2. $\forall w_i[x] <_H o_j[x],\ c_i <_H o_j[x]$.

The most well-known protocols for achieving different isolation degrees are based on the lock. Timestamp-based protocols have also been defined. In the following, we only present lock-based protocols. Interested readers are referred to [1], for a detailed explanation of lock-based and timestamp-based protocols.

Degree 0 Locking Protocol: a transaction T_i observes degree 0 isolation if it acquires an exclusive lock on a data item x before writing it and T_i releases this lock after performing the write operation.

Degree 1 Locking Protocol: a transaction T_i observes degree 1 isolation if:

1. T_i acquires an exclusive lock on a data item x before writing it and T_i releases this lock only at its commit time, and
2. T_i does not acquire any more exclusive lock once it releases an exclusive lock on a data item

Degree 2 Locking Protocol: a transaction T_i observes degree 2 isolation if:

1. T_i acquires a shared lock on a data item x before reading it and acquires an exclusive lock before writing it and T_i releases its shared locks after performing the read operation but holds all its exclusive locks until its commit time, and
2. T_i does not acquire any more exclusive locks once it releases an exclusive lock on a data item

Degree 3 Locking Protocol: a transaction T_i observes degree 3 isolation if:

1. T_i acquires a shared lock on a data item x before reading it and acquires an exclusive lock before writing it and T_i holds all its exclusive locks until its commit time, and
2. T_i does not acquire any more locks on a data item.

7 TRANSACTION PROCESSING MONITORS

The main function of a Transaction Processing (TP) monitor is the *integration* of different systems and the management of resources, so that applications spanning several systems and resources are executed according to the transactional model. As Figure 7 suggests, a TP monitor interfaces different software systems, and its main goal is to make these systems able to work together in a particular mode, known as *transaction-oriented processing*. Consequently, a TP monitor model is not a monolithic piece of software; integration functions are distributed on a arbitrary number of separated system components, generally executed through several processes and, depending on the environment, on different nodes. Note that TP

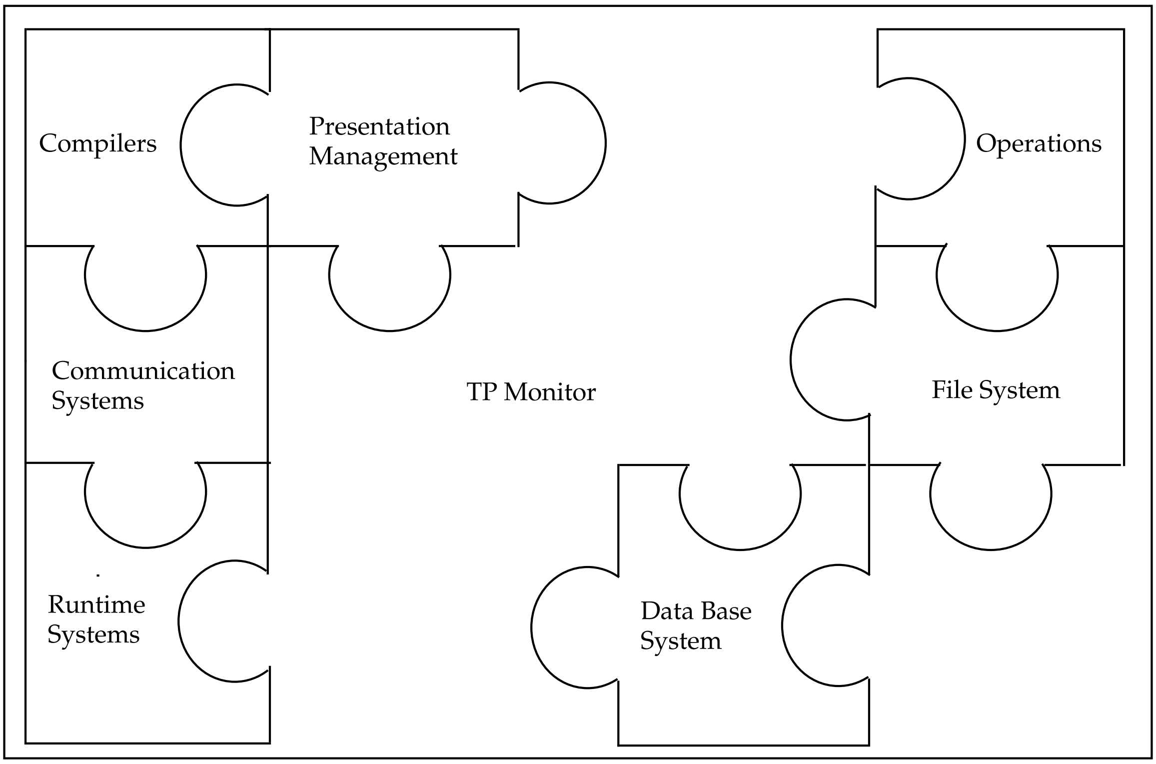


Figure 7: Integration function of the TP monitor with respect to different system components

monitors are different from DBMS. A DBMS, in addition to providing transactional services, provides a number of functions, such as query processing, data dictionaries, views. By contrast, a TP monitor is focused on transactional services. Very often, a TP monitor coordinates several DBMS, which are thus seen very much as the other resources. It is important to note, however, that not every DBMS can be managed as a resource by any TP monitor. For a DBMS to be managed by a given TP monitor, it must support the specific transaction protocol of the TP monitor. Transaction protocols are, however, being currently standardized. Such standardization efforts open the way to improving interoperability among DBMS and, more in general, among software systems. In the following, we define what transaction oriented-processing means and which services must be provided by a transaction processing system.

7.1 Transaction-oriented processing

Transaction-oriented processing can be characterized by some properties:

1. *Sharing.* In a transaction-oriented system, reads and writes are allowed on databases shared by several users.

2. *Variable requests.* Requests performed by users are random; they may be distributed according to some statistics, but individual requests cannot be foreseen.

3. *Repetitive workload.* Users do not perform arbitrary programs, but they rather require the system to execute some functions belonging to a predefined set. Each function is an instance of a type of transaction; that is, it invokes a transaction program implementing the required function. Typical transaction processing systems provide hundreds of different functions.

4. *Mainly simple function.* Most functions are limited in size and have the duration of typical batch jobs. They are quite similar to classical batch computation, except that they respect ACID properties, have not recoverable output, and allow some degree of data sharing.

5. *Multiple terminals.* In very large OnLine Transaction Processing (OLTP) systems, there are 1,000 to 100,000 terminals (clients).

6. *Intelligent clients.* OLTP terminals are often intelligent clients (workstations) that can execute their own processing keeping some of their data items, execute window management applications, and so on.

7. *High reliability.* Because of the large number of users, the system must be high available and reliable.

8. *The system performs recovery procedures.* Since data sharing is in use, there must be a formal assurance of consistency. After a failure occurs, all users must be informed about the current state of their environment, about which functions have been executed and which have not been executed, and so on. The guiding principle in this case is based on the ACID properties of transactions.

9. *Automatic load balance.* The system should have a high throughput, assuring at the same time, a short response time to the majority of user requests.

7.2 Transaction processing system services

A transaction processing system must allow the application programmer to take advantage of a programming environment, which provides transactional services. For data sharing, applications can use the services provided by a database manager. Database systems provide most of the mechanisms required to synchronize concurrent accesses to shared data, and to restore these data to a consistent state upon either a failure or a program error. DBMS, however, are not able to handle functions according to request/response style, which is characteristic of online transaction processing (OLTP) systems. DBMS are designed to process dynamic or pre-compiled SQL functions, able to support batch transactions and ad hoc queries that are executed by a process per user (terminal). This approach is not adequate to the general case of transaction-oriented processing, where requests are presented with a most general client-server style, requiring some predefined services, and where a transaction can include multiple interactions among different clients and servers.

Apart from data sharing access techniques, several other system services are required to support transaction-oriented processing. Services that a transaction-oriented system must provide, possibly by cooperation with a DBMS, are the following:

1. *Handling heterogeneity.* If the application function requires access to different (heterogeneous) databases or to different resource devices (for example different subsystems for vendors), local mechanisms are not sufficient to ensure ACID properties for the entire function. Services must link all operations on autonomous systems into a single function.

2. *Communications control.* If the application function (or some resource manager invoked by the application) establishes communications with a remote process, the state of such communications must be controlled on behalf of the transaction services. Naturally, this requires an adequate support from the underlying communication mechanism. This mechanism is called *Transactional Remote Procedure Call* (TRPC).

3. *Terminals management.* Functions are invoked by users through terminals of different types (workstations, conventional terminals, automated teller machines, bar code readers, etc.). Programs

implementing functions must then communicate with these terminals by messages. Since ACID properties of a transaction must be perceived by the user, besides by the program, sending and receiving messages must be part of transactions. This control must be performed by the TP monitor. In particular, the TP monitor must handle the problem arising when a response message is actually sent by the user, in case a failure occurs at time the message has been sent.

4. *Presentation services.* If the terminal uses sophisticated presentation services (such as a X-window system), recovery of the window environment, the cursor position and so on, must be ensured if a workstation failure occurs. This means that in a transactional environment a X-client must act as a resource manager, implementing ACID properties for its objects (windows) in cooperation with other system resource managers.

5. *Context management.* After a transaction has been executed on behalf of a user, next functions invocation (a user's input) often needs to have available the context[10] of the previous transaction executed by that user, terminal, or application. Some types of context are bound at only the terminal or the user, independently of any previous execution. Maintaining different types of context management is one of the duties of a TP monitor.

6. *Start/restart.* Since services are responsible for all components the application needs for its execution, the TP monitor must also manage the restart phase, after any failure. By doing that, all subsystems reach a state which is consistent with respect to the ACID properties.

Note that the above list of services is not complete. We have discussed those services which have a more or less immediate impact on the way programs are written in a transactional environment.

To summarize, we can say that a TP monitor offers the application programmer an environment providing the following features: all local executions are performed under a transaction protection; the program has not to worry about concurrency, failures, erasing, and so on. This is all managed by underlying transaction services. All external services of the above mentioned types automatically participate to the client's transaction. Therefore, the program has not to worry about matters such as recovering broken connections, and context rebuilding.

[10] A context is the set of values of local variables, registers, and data obtained by the transaction execution.

8 EXTENDED TRANSACTION MODELS

In Section 3 we have discussed the limitations of the flat transaction model. Several efforts have been made to solve each of such limitations. These researches have lead to the definition of new proposals extending the standard model. For example, complex updates (see Example 3) are supported by savepoints or by nested transactions; bulk updates (see Example 4) are supported by chained transactions, and so on. A new type of application results in a new, specialized, transaction model.

In the remainder of this section, the most well-known extensions to the flat transaction model are presented. The interested reader could refer to [9] and [6] for other kinds of extended models.

8.1 Flat transactions with Savepoints

The "all-or-nothing" characteristic of flat transaction model is both a virtue and a vice. It gives the simplest of all possible failure semantics, but in order to face possible errors during the execution, the application programmer has only two possibilities: he can insert into the application some special logic to recover from errors (the conventional method), or insert a RollBackWork command, giving up everything that will have been executed up to the error occurrence.

The latter approach is simpler, and for very short applications is appropriate. There are however situations in which a substantial amount of work has been done, and not all is invalidated by a single error in the transaction execution. If this happens, giving up the entire work is both undesirable and expensive. Having the option of stepping back to an earlier state inside the same transaction, the application can return to a proper state after an error occurs. This is what savepoints are used for.

A *savepoint* is established by invoking a function, usually called SaveWork, which causes the system to record the current state of execution[11]. The application program is returned a label that can be subsequently used to refer to that savepoint. Typically, the label is a monotonically increasing number. To reestablish a savepoint, the application invokes the RollBackWork function, and, rather than requesting the entire transaction to be aborted, it passes the number of the savepoint it wants to restore. This approach is illustrated in Figure 8. In such figure, shadowed actions are those rolled back due to the invocation

[11] By state of execution computer literature means usually the actual value of local variables, registers and shared data.

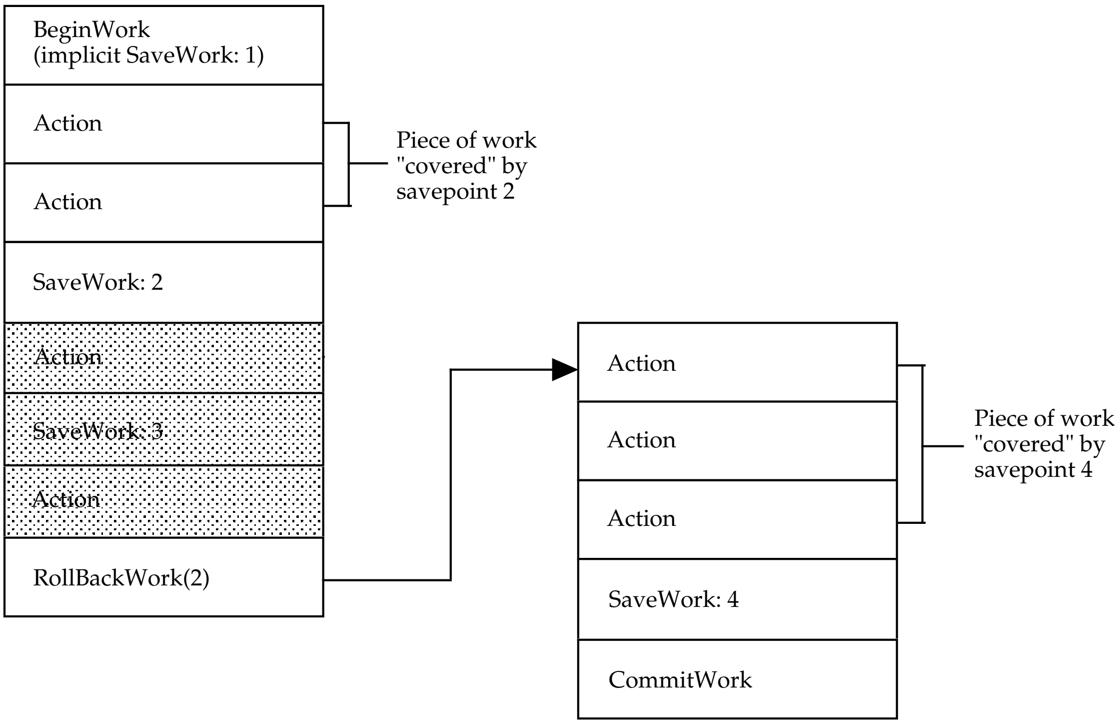


Figure 8: Example of use of savepoints inside a transaction

of RollBackWork(2) statement; the arrow points to pieces of work performed after the execution of such statement.

A closer inspection to the figure shows some interesting aspects about savepoints. First, the successful execution of the BeginWork statement establishes the first savepoint of the transaction. Second, there is a substantial difference between rolling a transaction back completely (aborting it), and rolling back to savepoint 1. In the first case, the transaction disappears from the system, is disconnected from the program that was running it, and loses all its resources. When rolling back to savepoint 1, the transaction stays alive, keeps its resources, and simply returns to an "empty state" where it has not yet done anything.

8.2 Chained Transactions

Chained transactions are a variation of savepoints. They try to achieve a compromise between rollback and the amount of lost work. This requires, however, a careful programming by the application developer.

The basic idea of chained transaction is that, rather than taking savepoints, the application program commits what it has done so far, thereby giving up its rights to perform a rollback. At the same time, however, it is required to stay within the same transaction scope. In particular, the data items the program has acquired during the previous processing are not released. This special commit operation is called ChainWork operation. It is a combination of CommitWork and BeginWork operations. Note that this is not the same as calling CommitWork and then starting a new transaction. The combination of both statements keeps the database context bound to the application, whereas a normal commit gives up all the context. Note that using a normal commit/begin sequence, with the next transaction reestablishing the previous context, would not work. The commit and the begin are separate statements and therefore, between their executions, some other transaction could have changed the database. The next transaction in the sequence would thus find the database state to be very different from what it expects.

According to the transaction chaining model, one can commit one transaction, release all data items that are no longer needed, and pass on the processing context that is required to the next transaction that is started. Note that the commitment of the first transaction and the beginning of the next one are executed as one atomic operation. This means that no other transaction can have seen (or changed) any of the data that are passed from one transaction to the other. Apart from the commit/begin sequence, each transaction behaves like a flat transaction.

We can notice the following differences between a transaction in the chain and one savepoint interval. First, since the chaining step irrevocably completes a transaction, rollback is limited to the currently active transaction. This corresponds to restoring the previous savepoint only, rather than selecting an arbitrary savepoint. Second, the commit allows the application to free locks that it does not later need. This provides a higher degree of concurrency than using a savepoint, which does not free any locks acquired so far. Finally, savepoints allow for a more flexible state restoration than flat transactions. After a crash, the entire chain of transactions is rolled back irrespective of any savepoints taken so far.

8.3 Nested Transactions

Nested transactions are a generalization of savepoints. Whereas savepoints allow organizing a transaction into a *sequence* of actions that can be rolled back individually, nested transactions form a *hierarchy* of pieces of work. A nested transaction consists of a top-level transaction, controlling the activity of lower-level transactions, called *subtransactions.* Each of these subtransactions becomes a self-contained but dependent action (with the A, C, and I properties of ACID), which can be completed or rolled back individually. Nesting has as many levels as the abstract layers in the application, without limitations.

The most popular definition of nested transaction is that developed by Moss [14], which can be summarized as follows:

1. A nested transaction is a tree of transactions; the sub-trees are either nested or flat transactions.

2. Transactions at the leaf level are flat transactions. The distance from the root to the leaves is arbitrary.

3. The transaction at the root of the tree is called the *top-level transaction*; all the others are called *subtransactions.* A transaction predecessor in the tree is called a *parent*; a subtransaction at the next lower level is also called a *child.*

4. A subtransaction can either commit or roll back; its commit takes effect only if the parent transaction commits. By induction, therefore, any subtransaction can finally commit only if the root transaction commits.

5. Rolling back a transaction in the tree causes all its subtransactions to roll back. This condition, together with the previous one, is the reason why sub-transactions have only A, C, and I properties, but not the D property.

In Moss's model, the actual work can only be performed by the leaf transactions. Only these transactions can access the database, send messages, and acquire other types of resource. The aim of higher-level transactions is to organize the processing flow and determine when to invoke subtransactions.

The rules establishing the behavior of nested transactions are the following:

1. *Commit rule.* The commit of a subtransaction makes its results accessible only to the parent transaction. The subtransaction will release its results to the outside world only if it has committed itself locally and all its ancestors up to the root have committed.

2. *Rollback rule.* If a (sub)transaction is rolled back, all its subtransactions are also rolled back, regardless of what they have done. This rule is applied recursively down the nesting hierarchy. The immediate consequence is that if the root transaction is rolled back, all its subtransactions are also rolled back.

3. *Visibility rule.* All changes made by a subtransaction become visible to the parent transaction at subtransaction commitment. All data items held by a parent transaction can be made accessible to its subtransactions. Changes made by a subtransaction are not visible to its siblings, in case they execute concurrently.

It is important to note that subtransactions are not exactly equivalent to flat transactions. Subtransactions are valid only within the confines of the surrounding higher-level transaction. They are *atomic* from the point of view of the parent transaction; they are *consistency preserving* with respect to the local function they implement; they are *isolated* from all other activities inside and outside the parent transaction; but they are not durable because of the commit rule.

8.4 Distributed Transactions

A distributed transaction consists of a flat transaction that runs in a distributed environment and therefore has to visit several nodes in the network, depending on where the data are located. There exist several differences between distributed transactions and nested transactions.

Nested transactions are structured by the functional decomposition of the application, that is, by what the application views. The structure of a distributed transaction depends on the distribution of data in a network. The invocation structure of a distributed transaction is however exactly the same of a nested transaction.

As mentioned, the decomposition into subtransactions does not reflect a hierarchical structure in the programs to be executed, but is induced by the position of the data in the network. Simple flat transactions (from the application perspective) may be executed as distributed transactions if they run in a distributed database system and the data they access are scattered across multiple nodes. If a subtransaction issues CommitWork, this signals the commit of the entire transaction, which forces all other subtransactions to commit. By comparison, in a nested transaction this is simply a local commit of that subtransaction. Distributed transactions normally cannot roll back independently; their decision to abort also affects the entire transaction. This all means that the connection between subtransactions and their parents is much stronger in the distributed transaction model than those existing in the nested transaction model.

8.5 Multi-Level Transactions

Multi-level transactions are a generalized and more liberal version of nested transactions. They allow a premature commitment of a subtransaction (also called pre-commit). However rather than just letting the subtransaction disappear, they assume the existence of a compensating transaction, able to semantically reverse the actions performed by the original subtransaction in case the parent transaction decides (or is forced) to roll back. This compensating transaction can, of course, be another nested or multi-level transaction.

Note again the important differences in techniques used to achieve atomicity: as long as strict commitment control can be exercised over the modified data, it is possible at any point in time to decide to roll back, which then can be performed by just reestablishing the old value.

8.6 Open Nested Transactions

Open nested transactions are the "anarchic" version of multi-level transactions. Subtransactions can either abort or commit independently of what the status of the parent transaction is. There are no restrictions with respect to the semantic relationships of parent and child transactions. In particular, no data hierarchy, or anything like it, is assumed.

Another way to describe open nested transactions is to say that they are a means for firing off other top-level transactions without exercising any further control over them.

8.7 Workflow Transactions

An important application area has recently entered the database management systems world: managing all activities that belong to the business sphere of a given company. The goal is to better support business processes of companies and organizations.

Today, companies must face with several problems, such as competition with other companies practicing the same activity, and reduction of business management costs, assuring at the same time a fast development of new services/products. To do this, they must constantly optimize business procedures and make existing information systems and applications more flexible.

Initially, business procedures were entirely handled by humans. As a result of introducing information technology, processes have partially or totally been automated by programs executed by computers able to perform procedures and to enforce rules previously handled by humans.

The *workflow technology* plays a fundamental role in automating business procedures. It provides software to support business processes, modeling them through *workflow specifications*.

There is no workflow definition commonly adopted. The term workflow is used to denote: a business process, a process specification, the software implementing and automating a process, the software supporting coordination and cooperation among people who implement a process.

Following the definition adopted in [8], *a workflow is a collection of tasks* organized in such a way to perform business processes (for instance, telephonic purchase orders and insurance claims processing). A process can be performed by one or more software systems, one or more groups of people, or a combination of them. Besides to be a collection of tasks, *a workflow defines processes invocation order* and *processes invocation conditions*; it moreover handles *processes synchronization* and *information flow*.

Workflow Management - WFM - is a technology which supports reengineering of business processes. It deals with defining workflow, or describing those aspects of a process that can be relevant to establish the control and the coordination of the various tasks constituting it. Moreover, a WFM system must support (re)design and (re)implementation of processes because of changes in information and business processing systems.

To efficiently support WFM, existing processing environments must be extended to achieve new distributed environments that:

1. are *component-oriented*, which means supporting integration and interoperability among system components;

2. support *workflow applications*, corresponding to implementations of information or business processes accessing to heterogeneous, autonomous and distributed - HAD - systems.

3. assure application correctness and reliability[12] in case of concurrent executions and failures;

4. support evolution, addition and elimination of workflow applications and system components when reengineering is necessary.

Several commercial systems have been developed to support WFM. Nowadays, WFM systems for office automation can support document management, application management, and/or human coordination, collaboration and co-decision. Although many of such WFM systems meet the above requirements, they only support a limited interoperability (in term of HAD system types they can integrate and tasks they can support). Moreover, they are not able to ensure correctness and reliability of applications when concurrently executing and in spite of failures, and they also suffer from low performance.

To meet the above requirements, as pointed out in [8], two different technologies need to be combined with those supplied by commercial WFM systems. They are:

- *Distributed Object Management* (DOM);
- *Customized Transaction Management* (CTM).

DOM supports interoperability and integration among HAD systems and applications implementing information and business processes. DOM also allows a WFM system to deal with problems such as replacement, migration, and evolution of HAD systems and modifications in their functions. Moreover, DOM technology provides an object model that makes managing complexity easy, by means of functionalities that are typical of object oriented systems, such as abstraction, inheritance and polymorphism.

CTM technology assures correctness and reliability of applications that implement information and business processes, also fulfilling some important requirements, such as isolation, coordination and collaboration among tasks. Moreover, CTM handles changes in correctness and reliability requirements of a process, and guarantees correctness and reliability.

[12] When multiple objects (databases, files, document, etc.) are accessed by a workflow execution, several consistency problems may arise because of concurrent execution and failures. This leads to the requirement for concurrency control, recovery and transactions coordination mechanisms.

In the following, the generic term *process* is used to refer to business processes, information processes, or both business processes and their corresponding information processes. WFM includes everything from modeling processes to synchronization of activities among systems and people performing processes. In particular, workflow management involves:

1. *modeling processes and workflow specifications*: this requires designing workflow models and methodologies to represent processes as workflow specifications;

2. *processes reengineering*: this requires ad hoc methodologies to optimize business processes;

3. *workflow implementation and automation*: this requires methodologies/technologies to use information systems, and people to implement, perform and control tasks according to workflow specifications.

The modeling phase of a process starts with a request of information to people having expert knowledge about the process. When a sufficient amount of information is collected, workflow specifications are produced to formally model the process. On the other hand, executing workflow specifications requires a *workflow model.* This model typically includes a set of useful concepts to describe processes, their tasks, dependencies among tasks, and roles of specified tasks.

Workflow specifications are usually expressed by *specification languages.* Workflow specification languages of commercial WFM systems use rules, constraints, and/or graphical constructs to describe ordering and synchronization among tasks of a workflow, and use tasks attributes to describe tasks and their roles.

The aim of reengineering methodologies is optimizing business processes. Process optimization strategies depend on reengineering objectives (for instance, reducing business costs and introducing new products/services).

Workflow implementation deals with all problems about realizing a workflow, using computers, software, information systems and WFM systems. Workflow automation deals, instead, with scheduling and controlling the workflow execution.

Transaction mechanisms for workflow applications require several extensions that we discuss in the following subsection.

8.7.1 Customized Transaction Management

A Customized Transaction Management system can specifically support correctness, reliability and functional requirements for workflow applications. To ensure workflow correctness and reliability, each workflow application must be associated with a *transactional model*. DBMS and TP monitors provide as a default an ACID transactional model to every application that uses their transaction processing services. The basic transaction model imposes to verify all ACID properties; this allows applications to interleave only if their concurrent execution is equivalent to a serial execution. The basic transaction model is not appropriate for many workflow applications. For instance, the basic model ensures process isolation, but it does not permit any cooperation among processes. This is too restrictive for workflows requesting process interaction.

CTM is the technology satisfying workflow requirements. CTM supports definition of extended transactional models. *Transactional workflows* enables the coordinated execution of several processes, can involve human resources and support the use of ACID properties for any process or for the entire workflow. Transactional workflow are supported by an extended transactional model defining correctness and reliability criteria for workflows. Workflow requirements are typically so different that a single extended model is not sufficient to satisfy workflow necessities.

The need of introducing extended models to extend the traditional transaction model to provide further functionalities requested by applications, and to improve system performance, has been recognized a long time ago. Several extended models have been proposed; however, many of these extensions resulted in new models offering correctness guarantees that are appropriate for an application, but do not assure correctness for other applications. Moreover, an extended model can impose restrictions not acceptable for an application, although required by others. If no extended models satisfy an application requirement, a new model must be defined, and so on.

Transactional workflow specification involves the specification of customized models by defining workflow structures and correctness criteria. A customized model specification is based on the fact that a transaction consists of a set of component transactions (corresponding to processes constituting a workflow) and of a set of transactional dependencies among them (corresponding to the workflow structure and to correctness criteria). If a transactional workflow allows nested levels, the constituent transactions may contain in turn component transactions. Each transactional workflow T has two types of dependencies:

- intra-transaction dependencies, defining relations among the component transactions of T;
- inter-transaction dependencies, defining relations among T and all other transactional workflows.

A first category of intra-transactions consists of *state dependencies*. Dependencies in this category are conditions on transaction states. Defining intra-transaction state dependencies involves the use of transaction semantics (for example, BeginWork, AbortWork and CommitWork) to specify the workflow processes ordering. Such specifications are more precise than those achieved by workflow specification languages of WFM systems.

The second category of intra-transaction dependencies consists of *correctness dependencies*, that specify which concurrent executions of transactional workflows preserve consistency and produce correct results, thereby defining correctness criteria.

Commercial DBMS, TP monitors and multidatabases currently support only a small set of extended models. For example, CORBA transaction service (see [16] for more explanations) is essentially a TP monitor. Therefore, it has the same limitations of other TP monitors. The CTM concept and the research to develop CTM technologies are quite recent. The Transaction Specification and Management Environment -TSME - ([7]) is an example of a research effort to develop a CTM technology. To support specifications of an extended model, and the configuration of the corresponding transaction management mechanism, TSME provides a transaction dependencies specification facility and a programmable mechanism for managing transactions.

9 CONCLUDING REMARKS

In this paper, we have presented the main concepts underlying the notion of transaction as well as techniques giving a concrete implementation to these concepts. Before concluding the paper, we would like to give a look to other developments in the transaction area. Because of its relevance to database technology, the transaction processing area has been and still is a very active area of research and development.

Roughly speaking, developments can categorized along three directions: extensions arising from new data models; extensions arising from developments in system architectures; extensions arising from new applications. In the remainder we briefly discuss such directions.

The database area has been characterized by intense research in the area of data models with the goal of developing models semantically richer than the relational models. Very-well known models include the object-oriented data model, the deductive data model, the active data model, and the most recent object-relational model. The semantic expressivity of those models has required extensions to transaction models in several respects. Modeling notions such as inheritance hierarchies, composite objects and versions have required introducing new locking modes [10]. Triggers, typical of active database systems, have required investigating different *coupling modes* for processing updates generated by trigger executions; for example, such updates can be executed during the transaction which generated them or as a separate transaction. Moreover, the fact that more semantics about the data is represented by those models makes it possible to exploit the semantics for more efficient concurrency control algorithms, thus resulting in *semantics-based* algorithms.

Among the developments in system architectures that have an impact on transaction processing techniques, we would like to mention main-memory systems and parallel systems. In main-memory systems, the entire database resides in main memory with back-up copies on secondary storage. Such architecture has required to revisit recovery techniques to ensure that information needed to recover the database can be saved on secondary storage without performance penalty on the normal main-memory operations. An approach proposed in [12] is based on the use of two independent processors, a main processor and a recovery processor. Parallel architectures are crucial in database systems because of their role in substantially improving performance. In such architectures, data sets are often partitioned across multiple nodes. Such an approach allows the same operation to be performed in parallel on all data belonging to the same set. It requires, however, to maintain consistency among the various partitions and in all related access structures (see for an example the locking protocol for the two-level indices presented in [4]).

DBMS are being used in many different environments; many of them require substantial extensions to transaction models and architectures. Among the application environments, we would like to recall the following: design applications, highly secure systems, real-time systems, mobile computing systems, www applications. Design applications are characterized by the fact that a change to the data may require days or weeks or even months, unlike conventional transactions that last few milliseconds. Often, in such applications data are extracted from a common database, cached at the user workstation, and modified by the users through specialized editors (for example CAD editors). To support such applications, notions such as long-duration transactions, check-in/check-out mechanisms, and versions have been introduced. Highly secure systems require transactions accessing highly-sensitive data be executed in complete isolation from other transactions. This requires the use of special concurrency control techniques to prevent any interference among transactions accessing data of different sensitivity. Commercial products are today available using such techniques. Real-time systems represent an important and difficult area. Such systems have transactions with stringent deadlines; each transaction must complete with a certain time. In general, a transaction in such a system is assigned a priority denoting the "urgency" that the transaction has of completing its execution. Because of this requirement, a transaction with high priority may not be able to wait for a lock to be released and, therefore, special approaches must be used in scheduling transactions. Mobile computing systems are characterized by the fact that users perform their tasks on mobile computing devices - called mobile hosts. Mobile hosts are connected via wireless networks to fixed hosts. Mobile hosts interact with the fixed hosts, among other things, to retrieve and store data. Mobile hosts may, however, became disconnected from the fixed hosts. The main problem is that users may continue working on their mobile hosts, even when being disconnected from the network. This situation requires extending recovery and concurrency control techniques, developed for distributed systems, to this new environment. Finally, www applications and Java-like systems will raise new requirements toward transaction managements. Research to identify those requirements and related approaches is just at its beginning[5].

References

[1] V. Atluri, E. Bertino, and S. Jajodia. A Theoretical Formulation for Degrees of Isolation in Databases. *Information and Software Technology*. To appear.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[3] E. Bertino, S. Jajodia, and P. Samarati. Database Security: Research and Practice. *Information Systems*, 20(7):537-556, October 1995.

[4] D.M. Choy and C. Mohan. Locking Protocols for Two-Tier Indexing of Paritioned Data. In *Proc. of the Int. Workshop on Advanced Transaction Models and Architectures (ATMA)*, Goa (India), August 31-September 2, 1996.

[5] L. Daynes, M. Atkinson, and P. Valduriez. Efficient Support for Customizing Concurrency Control in Persistent Java. In *Proc. of the Int. Workshop on Advanced Transaction Models and Architectures (ATMA)*, Goa (India), August 31-September 2, 1996.

[6] A.K. Elmagarmid. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publisher, San Mateo, California, 1990.

[7] D. Georgakopolous, M. Hornik, P, Krychniak, and F. Manola. Specification and Management of Extended Transactions in a Programmable Transaction Environment. In *Proc. of the 10th Int. Conference on Data Engineering*, Houston, Texas, February 1994.

[8] D. Georgakopolous, M. Hornik, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119-153, 1995. Kluwer Academic Publishers, Boston.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publisher, San Mateo, California, 1993.

[10] W. Kim, E. Bertino, and J. Garza. Composite Objects Revisited. In *Proc. of the Int. ACM-SIGMOD Conference on Management of Data*, Portland, Oregon, May 1989.

[11] H.F. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill Book Company Co., Singapore, 1986.

[12] T. Lehman and M. Carey. A Recovery Algorithm for a High-Performance Memory-Resident Database Systems. In *Proc. of the Int. ACM-SIGMOD Conference on Management of Data*, San Francisco, California, May 27-29, 1987.

[13] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1): 94–162, March 1992.

[14] J.E.B. Moss. *Nested Transaction: An Approach to Reliable Distributed Computing.* Technical Report MIT, LCS-TR-260, 1981.

[15] P. O'Neil. *Database - Principles, programming, Performance*. Morgan Kaufman, 1994.

[16] *Object Transaction Service*. OMG TC Document 92.11.1, 1992.