Database System Implementation

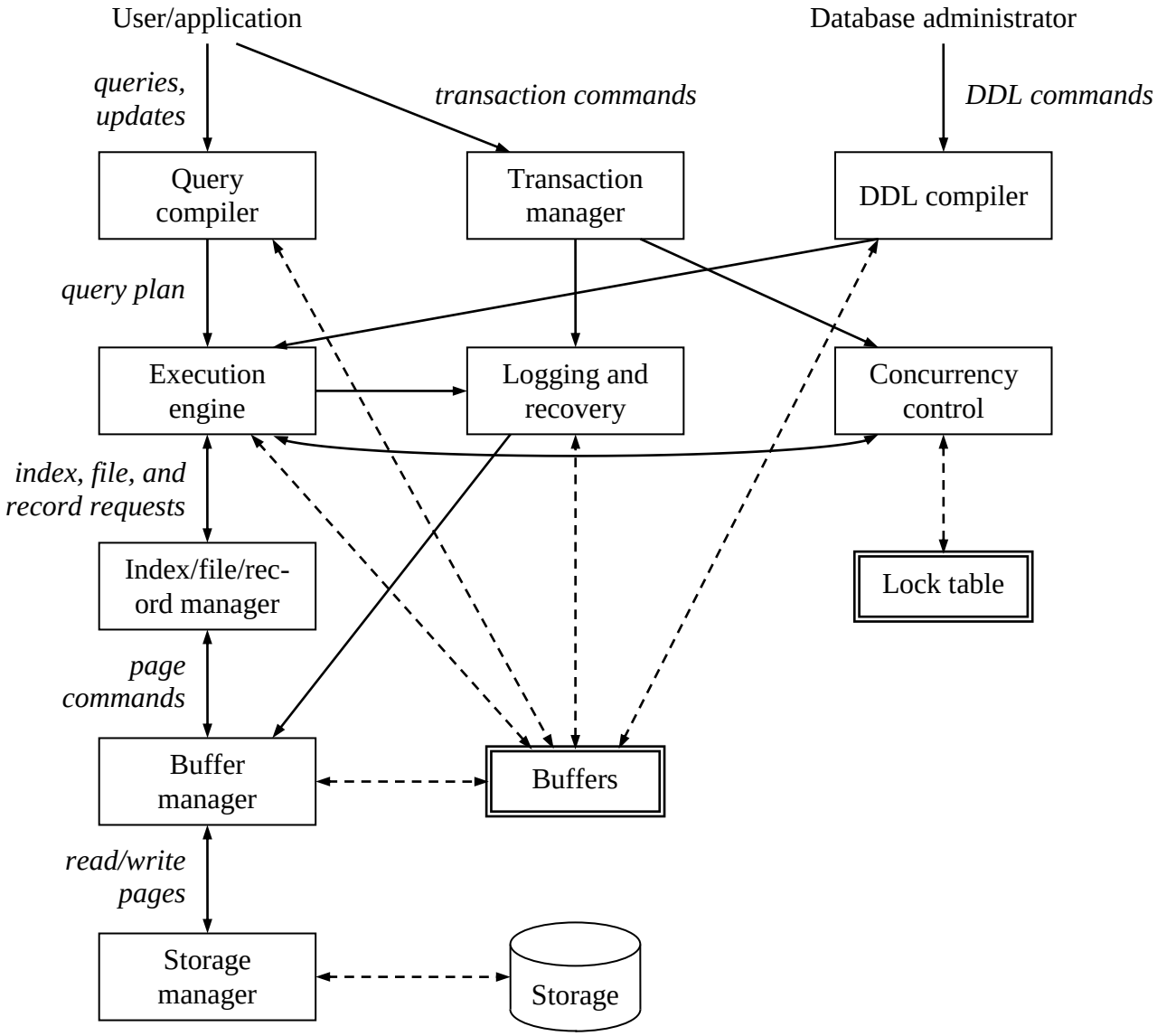
Textbook: Hector Garcia-Molina – Jeffrey D. Ullman – Jennifer Widom: *Database Systems – The Complete Book*, Second Edition, Pearson Prentice Hall, 2009, Chapters 17 and 18

Prerequisite: Database Systems course.

Topics: System failures and logging techniques against them; concurrency control.

Introduction

Database Management System Components



The figure illustrates the architecture of a general database management system (DBMS). Single-lined boxes denote the components of the system, whereas double-lined boxes represent in-memory data structures. Solid arrows denote control flow accompanied by data flow, and dashed arrows denote only data flow. The great majority of interactions with the DBMS follow the path on the left side of the figure. A user

or an application program initiates some action, using the data manipulation language (DML). This command does not affect the schema of the database but may affect the content of the database (if the action is a modification command) or will extract data from the database (if the action is a query). DML statements are handled by two separate subsystems:

1. *Answering the query.* The query is parsed and optimized by a query compiler. The resulting query execution plan (query plan for short), or sequence of actions the DBMS will perform to answer the query, is passed to the execution engine. The execution engine issues a sequence of requests for small pieces of data, typically records or tuples of a relation, to a resource manager that knows about data files (holding relations), the format and size of records in those files, and index files, which help find elements of data files quickly. The requests for data are passed to the buffer manager. The buffer manager's task is to bring appropriate portions of the data from secondary storage (disk) where it is kept permanently, to the main-memory buffers. Normally, the page or "disk block" is the unit of transfer between buffers and disk. The buffer manager communicates with a storage manager to get data from disk. The storage manager might involve operating-system commands, but more typically, the DBMS issues commands directly to the disk controller.
2. *Transaction Processing.* Queries and other DML actions are grouped into transactions, which are units that must be executed atomically and in isolation from one another. Any query or modification action can be a transaction by itself. In addition, the execution of transactions must be durable, meaning that the effect of any completed transaction must be preserved even if the system fails in some way right after completion of the transaction. We divide the transaction processor into two major parts:
    a) *Concurrency control manager* or *scheduler*: responsible for assuring atomicity and isolation of transactions.
    b) *Logging and recovery manager*: responsible for the atomicity and durability of transactions.

Transaction

The transaction is the unit of execution of database operations, consisting of DML statements, and having the following properties:

- ***A**tomicity*: the all-or-nothing execution of transactions (the database operations in a transaction are either fully executed or not at all). Either all relevant data has to be changed in the database or none at all. This means that if one part of a transaction fails, the entire transaction fails, and the database state is left unchanged.
- ***C**onsistency preservation*: transactions are expected to preserve the consistency of the database, i.e., after the execution of a transaction, all consistency (or integrity) constraints (expectations about data elements and the relationships among them) defined in the database should be satisfied.
- ***I**solation*: the fact that each transaction must appear to be executed as if no other transaction were executing at the same time.
- ***D**urability*: the condition that the effect on the database of a transaction must never be lost, once the transaction has completed.

These are the *ACID properties* of transactions. From the DBMS's point of view, consistency preservation is always considered satisfied (see later: correctness principle), the other three properties, however, must be forced by the DBMS, although, sometimes we set aside some of them. For example, if we are issuing ad-hoc commands to a SQL system, then each query or database modification statement (plus any resulting trigger actions) is a transaction. When using an embedded SQL interface, the programmer controls the extent of a transaction, which may include several queries or modifications, as well as operations performed

in the host language. In the typical embedded SQL system, transactions begin as soon as operations on the database are executed and end with an explicit `COMMIT` or `ROLLBACK` ("abort") statement.

Transaction Processing

The transaction processor provides concurrent access to data and supports resilience (i.e., data integrity after a system failure) by executing transactions correctly. The transaction manager therefore accepts transaction commands from an application, which tell the transaction manager when transactions begin and end, as well as information about the expectations of the application (some may not wish to require atomicity, for example). The transaction processor performs the following tasks:

1. *Logging*: In order to assure durability, every change in the database is logged separately on disk. The log manager follows one of several policies designed to assure that no matter when a system failure or "crash" occurs, a recovery manager will be able to examine the log of changes and restore the database to some consistent state. The log manager initially writes the log in buffers and negotiates with the buffer manager to make sure that buffers are written to disk (where data can survive a crash) at appropriate times.
2. *Concurrency control*: Transactions must appear to execute in isolation. But in most systems, there will in truth be many transactions executing at once. Thus, the scheduler (concurrency control manager) must assure that the individual actions of multiple transactions are executed in such an order that the net effect is the same as if the transactions had in fact executed in their entirety, one-at-a-time. A typical scheduler does its work by maintaining locks on certain pieces of the database. These locks prevent two transactions from accessing the same piece of data in ways that interact badly. Locks are generally stored in a main-memory lock table, as suggested by the above figure. The scheduler affects the execution of queries and other database operations by forbidding the execution engine from accessing locked parts of the database.
3. *Deadlock resolution*: As transactions compete for resources through the locks that the scheduler grants, they can get into a situation where none can proceed because each needs something another transaction has. The transaction manager has the responsibility to intervene and cancel ("rollback" or "abort") one or more transactions to let the others proceed.

Coping with System Failures

This chapter deals with techniques for supporting the goal of *resilience*, that is, integrity of the data when the system fails in some way. (Data must not be corrupted simply because several error-free queries or database modifications are being done at once, either. This matter is addressed by concurrency control.)

The principal technique for supporting resilience is a *log*, which records securely the history of database changes. We shall discuss three different styles of logging, called "undo," "redo," and "undo/redo." We also discuss *recovery*, the process whereby the log is used to reconstruct what has happened to the database when there has been a failure. An important aspect of logging and recovery is avoidance of the situation where the log must be examined into the distant past. Thus, we shall learn about "*checkpointing*," which limits the length of log that must be examined during recovery.

We also discuss "*archiving*," which allows the database to survive not only temporary system failures but situations where the entire database is lost. Then, we must rely on a recent copy of the database (the archive) plus whatever log information survives to reconstruct the database as it existed at some point in the recent past. Finally, we shall learn about Oracle's logging and recovery management.

Failure Modes

There are many things that can go wrong as a database is queried and modified. Problems range from the keyboard entry of incorrect data to an explosion in the room where the database is stored on disk. The following items are a catalog of the most important failure modes and what the DBMS can do about them.

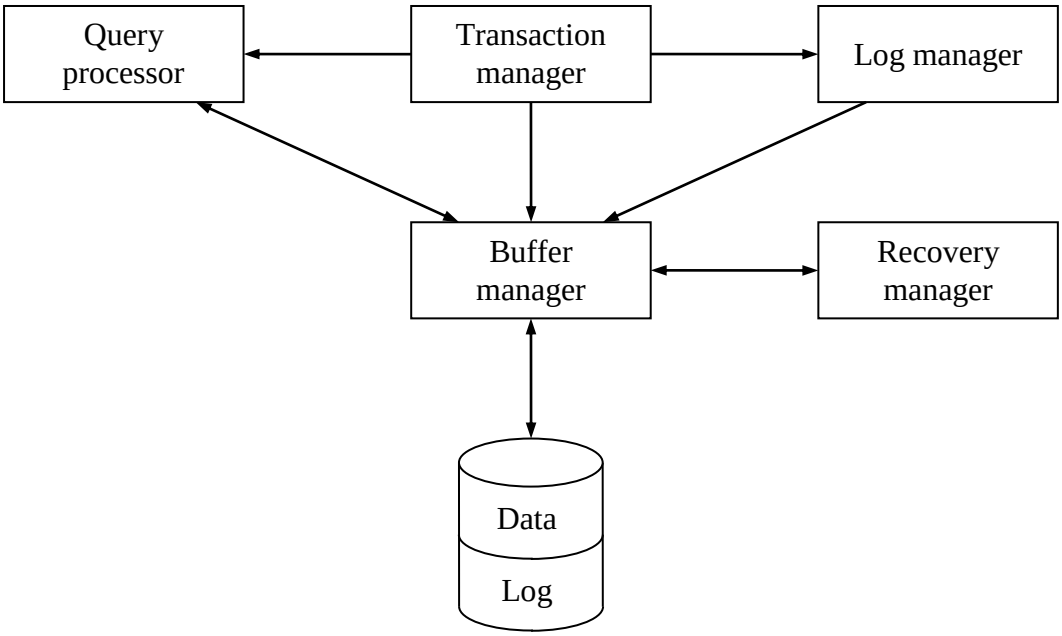
- *Erroneous data entry*: Some data errors are impossible to detect. For example, if a clerk mistypes one digit of your phone number, the data will still look like a phone number that could be yours. On the other hand, if the clerk omits a digit from your phone number, then the data is evidently in error, since it does not have the form of a phone number. The principal technique for addressing data entry errors is to write constraints and triggers that detect data believed to be erroneous. Triggers are program codes that execute automatically, typically in case of modifications of a certain type (such as inserting a row into a relation) in order to check if the new data satisfy the constraints defined by the designer of the database.
- *Media failures*: A local failure of a disk, one that changes only a bit or a few bits, can normally be detected by parity checks associated with the sectors of the disk. Head crashes, where the entire disk becomes unreadable, are generally handled by one or more of the following approaches:
    1. Use one of the *RAID* (Redundant Array of Independent Disks) schemes, so the lost disk can be restored.
    2. Maintain an *archive*, a copy of the database on a medium such as tape or optical disk. The archive is periodically created, either fully or incrementally, and stored at a safe distance from the database itself.
    3. Instead of an archive, one could keep *redundant copies of the database on-line*, distributed among several sites. In this case, consistency of the copies must be enforced.
- *Catastrophic failures*: In this category are a number of situations in which the media holding the database is completely destroyed. Examples include explosions, fires, or vandalism at the site of the database, as well as viruses. RAID will not help, since all the data disks and their parity check disks become useless simultaneously. However, the other approaches that can be used to protect against media failure — archiving and redundant, distributed copies — will also protect against a catastrophic failure.
- *System failures*: Each transaction has a *state*, which represents what has happened so far in the transaction. The state includes the current place in the transaction's code being executed and the values of any local variables of the transaction that will be needed later on. System failures are problems that cause the state of a transaction to be lost. Typical system failures are power loss and software errors. Since main memory is "volatile," a power failure will cause the contents of main memory to disappear, along with the result of any transaction step that was kept only in main memory, rather than on (nonvolatile) disk. Similarly, a software error may overwrite part of main memory, possibly including values that were part of the state of the program. When main memory is lost, the transaction state is lost; that is, we can no longer tell what parts of the transaction, including its database modifications, were made. Running the transaction again may not fix the problem. For example, if the transaction must add 1 to a value in the database, we do not know whether to repeat the addition of 1 or not. The principal remedy for the problems that arise due to a system error is logging of all database changes in a separate, nonvolatile log, coupled with recovery when necessary. However, the mechanisms whereby such logging can be done in a fail-safe manner are surprisingly intricate.

The Log Manager and the Transaction Manager

Assuring that transactions are executed correctly is the job of a *transaction manager*, a subsystem that performs several functions, including:

- issuing signals to the log manager (described below) so that necessary information in the form of "log records" can be stored on the log;
- assuring that concurrently executing transactions do not interfere with each other in ways that introduce errors (scheduling).

The transaction manager and its interactions are suggested by the following figure:



The transaction manager will send messages about actions of transactions to the log manager, to the buffer manager about when it is possible or necessary to copy the buffer back to disk, and to the query processor to execute the queries and other database operations that comprise the transaction.

The log manager maintains the log. It must deal with the buffer manager, since space for the log initially appears in main-memory buffers, and at certain times, these buffers must be copied to disk. The log, as well as the data, occupies space on the disk, as we suggest in the figure.

When there is a crash, the recovery manager is activated. It examines the log and uses it to repair the data if necessary. As always, access to the disk is through the buffer manager.

Correct Execution of Transactions

Before we can deal with correcting system errors, we need to understand what it means for a transaction to be executed "correctly." To begin, we assume that the database is composed of "elements." A *database element* is a functional unit of data stored in the physical database, whose value can be read or updated by transactions. A relation (or its object-oriented counterpart, a class extent), a tuple (or its OO counterpart, an object), or a disk block (or page) can all be considered as a database element. However, there are several good reasons in practice to use disk blocks or pages as the database element. In this way, buffer contents become single elements, allowing us to avoid some serious problems with logging and transactions. Avoiding database elements that are bigger than disk blocks also prevents a situation where part but not all of an element has been placed in nonvolatile storage (disk) when a crash occurs.

A database has a *state*, which is a value for each of its elements. Intuitively, we regard certain states as *consistent*, and others as *inconsistent*. Consistent states satisfy all constraints of the database schema, such as key constraints and constraints on values. *Explicit constraints* are enforced by the database, so any transaction that violates them will be rejected by the system and not change the database at all. However, consistent states must also satisfy *implicit constraints* that are in the mind of the database designer. The implicit constraints may be maintained by triggers that are part of the database schema, but they might also be maintained only by policy statements concerning the database, or warnings associated with the user interface through which updates are made. Implicit constraints cannot be characterized exactly under any circumstances. Our position is that if someone is given authority to modify the database, then they also have the authority to judge what the implicit constraints are.

A fundamental assumption about transactions is the *correctness principle*: If a transaction executes in the absence of any other transactions or system errors, and it starts with the database in a consistent state, then the database is also in a consistent state when the transaction ends (isolation + atomicity –> consistency preservation). There is a converse to the correctness principle that forms the motivation for both the logging techniques and the concurrency control mechanisms. This converse involves two points:

- A transaction is atomic; that is, it must be executed as a whole or not at all. If only part of a transaction executes, then the resulting database state may not be consistent.
- Transactions that execute simultaneously are likely to lead to an inconsistent state unless we take steps to control their interactions.

The Primitive Operations of Transactions

Let us now consider in detail how transactions interact with the database. There are three address spaces that interact in important ways:

1. the space of disk blocks holding the database elements;
2. the virtual or main memory address space that is managed by the buffer manager;
3. the local address space of the transaction.

For a transaction to read a database element, that element must first be brought to a main-memory buffer or buffers, if it is not already there. Then, the contents of the buffer(s) can be read by the transaction into its own address space. Writing a new value for a database element by a transaction follows the reverse route. The new value is first created by the transaction in its own space. Then, this value is copied to the appropriate buffer(s). It is very important to see that transactions may not write a new value for a database element directly on the disk.

The buffer may or may not be copied to disk immediately; that decision is the responsibility of the buffer manager in general. As we shall soon see, one of the principal tools for assuring resilience is forcing the buffer manager to write the block in a buffer back to disk at appropriate times. However, in order to reduce the number of disk I/O's, database systems can and will allow a change to exist only in volatile main-memory storage, at least for certain periods of time and under the proper set of conditions.

In order to study the details of logging algorithms and other transaction management algorithms, we need a notation that describes all the operations that move data between address spaces. The primitives we shall use are:

1. `INPUT(X)`: Copy the disk block containing database element `X` to a memory buffer.
2. `READ(X,t)`: Copy the database element `X` to the transaction's local variable `t`. More precisely, if the block containing database element `X` is not in a memory buffer, then first execute `INPUT(X)`. Next, assign the value of `X` to local variable `t`.

3. `WRITE(X,t)`: Copy the value of local variable `t` to database element `X` in a memory buffer. More precisely, if the block containing database element `X` is not in a memory buffer, then execute `INPUT(X)`. Next, copy the value of `t` to `X` in the buffer.
4. `OUTPUT(X)`: Copy the block containing `X` from its buffer to disk.

The above operations make sense as long as database elements reside within a single disk block and therefore within a single buffer. If a database element occupies several blocks, we shall imagine that each block-sized portion of the element is an element by itself. The logging mechanism to be used will assure that the transaction cannot complete without the write of `X` being atomic; i.e., either all blocks of `X` are written to disk, or none are. Thus, we shall assume for the entire discussion of logging that a database element is no larger than a single block.

Different DBMS components issue the various commands we just introduced. `READ` and `WRITE` are issued by transactions. `INPUT` and `OUTPUT` are normally issued by the buffer manager. `OUTPUT` can also be initiated by the log manager under certain conditions, as we shall see.

Example. To see how the above primitive operations relate to what a transaction might do, let us consider a database that has two elements, `A` and `B`, with the constraint that they must be equal in all consistent states. Transaction `T` consists logically of the following two steps:

```
A := A*2;
B := B*2;
```

If `T` starts in a consistent state (i.e., `A` = `B`) and completes its activities without interference from another transaction or system error, then the final state must also be consistent. That is, `T` doubles two equal elements to get new, equal elements.

Execution of `T` involves reading `A` and `B` from disk, performing arithmetic in the local address space of `T`, and writing the new values of `A` and `B` to their buffers. The relevant steps of `T` are thus:

```
READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t);
```

In addition, the buffer manager will eventually execute the `OUTPUT` steps to write these buffers back to disk. The following table shows the primitive steps of `T`, followed by the two `OUTPUT` commands from the buffer manager. We assume that initially `A` = `B` = 8. The values of the memory and disk copies of `A` and `B` and the local variable `t` in the address space of transaction `T` are indicated for each step:

Action	*t*	*M*–A	*M*–B	*D*–A	*D*–B
`READ(A,t)`	8	8		8	8
`t := t*2`	16	8		8	8
`WRITE(A,t)`	16	16		8	8
`READ(B,t)`	8	16	8	8	8
`t := t*2`	16	16	8	8	8
`WRITE(B,t)`	16	16	16	8	8
`OUTPUT(A)`	16	16	16	16	8
`OUTPUT(B)`	16	16	16	16	16

At the first step, `T` reads `A`, which generates an `INPUT(A)` command for the buffer manager if `A`'s block is not already in a buffer. The value of `A` is also copied by the `READ` command into local variable `t` of `T`'s address space. The second step doubles `t`; it has no effect on `A`, either in a buffer or on disk. The third step writes `t` into `A` in the buffer; it does not affect `A` on disk. The next three steps do the same for `B`, and the last two steps copy `A` and `B` to disk.

Observe that as long as all these steps execute, consistency of the database is preserved. If a system error occurs before `OUTPUT(A)` is executed, then there is no effect to the database stored on disk; it is as if `T`

never ran, and consistency is preserved. However, if there is a system error after `OUTPUT(A)` but before `OUTPUT(B)`, then the database is left in an inconsistent state. We cannot prevent this situation from ever occurring, but we can arrange that when it does occur, the problem can be repaired — either both `A` and `B` will be reset to 8, or both will be advanced to 16.

Example. Suppose that the consistency constraint on the database is $0 \leq A \leq B$. Tell whether each of the following transactions preserves consistency.

a) `A := A + B; B := A + B;`
b) `B := A + B; A := A + B;`
c) `A := B + 1; B := A + 1;`

Undo Logging

A *log* is a file of *log records*, each telling something about what some transaction has done. If log records appear in nonvolatile storage, we can use them to restore the database to a consistent state after a system crash. Our first style of logging — *undo logging* — makes repairs to the database state by undoing the effects of transactions that may not have completed before the crash.

Additionally, in this chapter we introduce the basic idea of log records, including the commit (successful completion of a transaction) action and its effect on the database state and log. We shall also consider how the log itself is created in main memory and copied to disk by a "flush-log" operation. Finally, we examine the undo log specifically, and learn how to use it in recovery from a crash. In order to avoid having to examine the entire log during recovery, we introduce the idea of "checkpointing," which allows old portions of the log to be thrown away.

Log Records

Imagine the log as a file opened for appending only. As transactions execute, the log manager has the job of recording in the log each important event. One block of the log at a time is filled with log records, each representing one of these events. Log blocks are initially created in main memory and are allocated by the buffer manager like any other blocks that the DBMS needs. The log blocks are written to nonvolatile storage on disk as soon as it is feasible.

There are several forms of log record that are used with each of the types of logging we discuss. These are:

1. `<START T>`: This record indicates that transaction `T` has begun.
2. `<COMMIT T>`: Transaction `T` has completed successfully and will make no more changes to database elements. Any changes to the database made by `T` should appear on disk. However, because we cannot control when the buffer manager chooses to copy blocks from memory to disk, we cannot in general be sure that the changes are already on disk when we see the `<COMMIT T>` log record. If we insist that the changes already be on disk, this requirement must be enforced by the log manager (as is the case for undo logging).
3. `<ABORT T>`: Transaction `T` could not complete successfully. If transaction `T` aborts, no changes it made can have been copied to disk, and it is the job of the transaction manager to make sure that such changes never appear on disk, or that their effect on disk is canceled if they do. We shall discuss the matter of repairing the effect of aborted transactions later. There can be several reasons for a transaction to abort. The simplest is when there is some error condition in the code of the transaction itself, e.g., an attempted division by zero. The DBMS may also abort a transaction for one of several reasons. For instance, a transaction may be involved in a deadlock, where it and one or more other transactions each

hold some resource that the other needs. Then, one or more transactions must be forced by the system to abort (see later).

4. <T,X,v>: This is the *update record*. The meaning of this record is: transaction T has changed database element X, and its former value was v. The change reflected by an update record normally occurs in memory, not disk; i.e., the log record is a response to a WRITE action into memory, not an OUTPUT action to disk. Notice also that an undo log does not record the new value of a database element, only the old value. As we shall see, should recovery be necessary in a system using undo logging, the only thing the recovery manager will do is cancel the possible effect of a transaction on disk by restoring the old value.

The Undo Logging Rules

An undo log is sufficient to allow recovery from a system failure, provided transactions and the buffer manager obey two rules:

U_1: If transaction T modifies database element X, then the log record of the form <T,X,v> must be written to disk *before* the new value of X is written to disk (*write-ahead logging* or WAL).

U_2: If a transaction commits, then its COMMIT log record must be written to disk only *after* all database elements changed by the transaction have been written to disk, but as soon thereafter as possible.

To summarize rules U_1 and U_2, material associated with one transaction must be written to disk in the following order:

1. the log records indicating changed database elements;
2. the changed database elements themselves;
3. the COMMIT log record.

However, the order of the first two steps applies to each database element individually, not to the group of update records for a transaction as a whole.

In order to force log records to disk, the log manager needs a flush-log command that tells the buffer manager to copy to disk any log blocks that have not previously been copied to disk or that have been changed since they were last copied. In sequences of actions, we shall show FLUSH LOG explicitly. The transaction manager also needs to have a way to tell the buffer manager to perform an OUTPUT action on a database element. We shall continue to show the OUTPUT action in sequences of transaction steps.

Example. Let us reconsider the previously investigated transaction in the light of undo logging. We shall expand our table to show the log entries and flush-log actions that have to take place along with the actions of transaction T:

Step	*Action*	*t*	*M*–A	*M*–B	*D*–A	*D*–B	*Log*
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T,A,8>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T,B,8>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)	OUTPUT(B)	16	16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

In line (1) of the table, transaction T begins. The first thing that happens is that the <START T> record is written to the log. Line (2) represents the read of A by T. Line (3) is the local change to t, which affects neither the database stored on disk nor any portion of the database in a memory buffer. Neither lines (2) nor (3) require any log entry, since they have no effect on the database.

Line (4) is the write of the new value of A to the buffer. This modification to A is reflected by the log entry <T,A,8>, which says that A was changed by T and its former value was 8. Note that the new value, 16, is not mentioned in an undo log.

Lines (5) through (7) perform the same three steps with B instead of A. At this point, T has completed and must commit. The changed A and B must migrate to disk, but in order to follow the two rules for undo logging, there is a fixed sequence of events that must happen.

First, A and B cannot be copied to disk until the log records for the changes are on disk. Thus, at step (8) the log is flushed, assuring that these records appear on disk. Then, steps (9) and (10) copy A and B to disk. The transaction manager requests these steps from the buffer manager in order to commit T.

Now, it is possible to commit T, and the <COMMIT T> record is written to the log, which is step (11). Finally, we must flush the log again at step (12) to make sure that the <COMMIT T> record of the log appears on disk. Notice that without writing this record to disk, we could have a situation where a transaction has committed, but for a long time a review of the log does not tell us that it has committed. That situation could cause strange behavior if there were a crash, because a transaction that appeared to the user to have completed long ago would then be undone and effectively aborted.

As we look at a sequence of actions and log entries like in the table above, it is tempting to imagine that these actions occur in isolation. However, the DBMS may be processing many transactions simultaneously. Thus, the four log records for transaction T may be interleaved on the log with records for other transactions. Moreover, if one of these transactions flushes the log, then the log records from T may appear on disk earlier than is implied by the flush-log actions. There is no harm if log records reflecting a database modification appear earlier than necessary. The essential policy for undo logging is that we don't write the <COMMIT T> record until the OUTPUT actions for T are completed.

A trickier situation occurs if two database elements A and B share a block. Then, writing one of them to disk writes the other as well. In the worst case, we can violate rule U_1 by writing one of these elements prematurely. It may be necessary to adopt additional constraints on transactions in order to make undo logging work. For instance, we might use a locking scheme where database elements are disk blocks, as described later, to prevent two transactions from accessing the same block at the same time. This and other problems that appear when database elements are fractions of a block motivate our suggestion that blocks be the database elements.

Recovery Using Undo Logging

Suppose now that a system failure occurs. It is possible that certain database changes made by a given transaction were written to disk, while other changes made by the same transaction never reached the disk. If so, the transaction was not executed atomically, and there may be an inconsistent database state. The recovery manager must use the log to restore the database to some consistent state.

First, we consider only the simplest form of recovery manager, one that looks at the entire log, no matter how long, and makes database changes as a result of its examination. Later, we consider a more sensible approach, where the log is periodically "checkpointed," to limit the distance back in history that the recovery manager must go.

The first task of the recovery manager is to divide the transactions into committed and uncommitted transactions. If there is a log record <COMMIT T>, then by undo rule U_2 all changes made by transaction

T were previously written to disk. Thus, T by itself could not have left the database in an inconsistent state when the system failure occurred.

However, suppose that we find a <START T> record on the log but no <COMMIT T> or <ABORT T> record. Then there could have been some changes to the database made by T that were written to disk before the crash, while other changes by T either were not made, or were made in the main-memory buffers but not copied to disk. In this case, T is an *incomplete transaction* and must be undone. That is, whatever changes T made must be reset to their previous value. Fortunately, rule U_1 assures us that if T changed X on disk before the crash, then there will be a <T,X,v> record on the log, and that record will have been copied to disk before the crash. Thus, during the recovery, we must write the value v for database element X. Note that this rule raises the question whether X had value v in the database anyway; we don't even bother to check.

Since there may be several uncommitted transactions in the log, and there may even be several uncommitted transactions that modified X, we have to be systematic about the order in which we restore values. Thus, the recovery manager must scan the log from the end (i.e., from the most recently written record to the earliest written). As it travels, it remembers all those transactions T for which it has seen a <COMMIT T> record or an <ABORT T> record. Also as it travels backward, if it sees a record <T,X,v>, then:

- if T is a transaction whose COMMIT record has been seen, then do nothing, as T is committed and must not be undone (T is completed);
- if an ABORT record has been seen for transaction T, then again do nothing, as T has already been recovered (T is completed);
- otherwise, T is an incomplete transaction, so the recovery manager must change the value of X in the database to v, in case X had been altered just before the crash.

After making these changes, the recovery manager must write a log record <ABORT T> for each incomplete transaction T, and then flush the log. Now, normal operation of the database may resume, and new transactions may begin executing.

Example. Let us consider the sequence of actions from the above example. There are several different times that the system crash could have occurred; let us consider each significantly different one.

1. The crash occurs after step (12). Then the <COMMIT T> record reached disk before the crash. When we recover, we do not undo the results of T, and all log records concerning T are ignored by the recovery manager.
2. The crash occurs between steps (11) and (12). It is possible that the log record containing the COMMIT got flushed to disk; for instance, the buffer manager may have needed the buffer containing the end of the log for another transaction, or some other transaction may have asked for a log flush. If so, then the recovery is the same as in case (1) as far as T is concerned. However, if the COMMIT record never reached disk, then the recovery manager considers T incomplete. When it scans the log backward, it comes first to the record <T,B,8>. It therefore stores 8 as the value of B on disk. It then comes to the record <T,A,8> and makes A have value 8 on disk. Finally, the record <ABORT T> is written to the log, and the log is flushed.
3. The crash occurs between steps (10) and (11). Now, the COMMIT record surely was not written, so T is incomplete and is undone as in case (2).
4. The crash occurs between steps (8) and (10). Again, T is undone. In this case, the change to A and/or B may not have reached disk. Nevertheless, the proper value, 8, is restored for each of these database elements.

5. The crash occurs prior to step (8). Now, it is not certain whether any of the log records concerning T have reached disk. However, we know by rule U_1 that if the change to A and/or B reached disk, then the corresponding log record reached disk. Therefore, if there were changes to A and/or B made on disk by T, then the corresponding log record will cause the recovery manager to undo those changes.

Suppose the system again crashes while we are recovering from a previous crash. Because of the way undo log records are designed, giving the old value rather than, say, the change in the value of a database element, the recovery steps are idempotent, that is, repeating them many times has exactly the same effect as performing them once. We already observed that if we find a record <T,X,v>, it does not matter whether the value of X is already v — we may write v for X regardless. Similarly, if we repeat the recovery process, it does not matter whether the first recovery attempt restored some old values; we simply restore them again. The same reasoning holds for the other logging methods we discuss. Since the recovery operations are idempotent, we can recover a second time without worrying about changes made the first time.

Checkpointing

As we observed, recovery requires that the entire log be examined, in principle. When logging follows the undo style, once a transaction has its COMMIT log record written to disk, the log records of that transaction are no longer needed during recovery. We might imagine that we could delete the log prior to a COMMIT, but sometimes we cannot. The reason is that often many transactions execute at once. If we truncated the log after one transaction committed, log records pertaining to some other active transaction T might be lost and could not be used to undo T if recovery were necessary.

The simplest way to untangle potential problems is to *checkpoint* the log periodically. There are two kinds of checkpoints: simple and nonquiescent. In a simple checkpoint, we:

1. stop accepting new transactions;
2. wait until all currently active transactions commit or abort and have written a COMMIT or ABORT record on the log;
3. flush the log to disk;
4. write a log record <CKPT>, and flush the log again;
5. resume accepting transactions.

Any transaction that executed prior to the checkpoint will have finished, and by rule U_2, its changes will have reached the disk. Thus, there will be no need to undo any of these transactions during recovery. During a recovery, we scan the log backwards from the end, identifying incomplete transactions. However, when we find a <CKPT> record, we know that we have seen all the incomplete transactions. Since no transactions may begin until the checkpoint ends, we must have seen every log record pertaining to the incomplete transactions already. Thus, there is no need to scan prior to the <CKPT>, and in fact, the log before that point can be deleted or overwritten safely (unless it is needed for some other reason).

Example. Consider the following log:

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>
<T_2,C,15>
<T_1,D,20>
<COMMIT T_1>
<COMMIT T_2>
<CKPT>
<START T_3>

<T_3,E,25>
<T_3,F,30>

Suppose we decide to do a checkpoint after the fourth entry. Since T_1 and T_2 are the active (incomplete) transactions, we shall have to wait until they complete before writing the <CKPT> record on the log. Suppose a crash occurs at the end of the listing. Scanning the log from the end, we identify T_3 as the only incomplete transaction and restore E and F to their former values 25 and 30, respectively. When we reach the <CKPT> record, we know there is no need to examine prior log records and the restoration of the database state is complete.

The question may arise how to find the last log record? It is common to recycle blocks of the log file on disk, since checkpoints allow us to drop old portions of the log. However, if we overwrite old log records, then we need to keep a serial number, which may only increase, as suggested by the following figure:

~~1~~	~~2~~	~~3~~	4	5	6	7	8
9	10	11					

Then, we can find the record whose serial number is greater than that of the next record; this record will be the current end of the log, and the entire log is found by ordering the current records by their present serial numbers. In practice, a large log may be composed of many files, with a "top" file whose records indicate the files that comprise the log. Then, to recover, we find the last record of the top file, go to the file indicated, and find the last record there.

Nonquiescent Checkpointing

A problem with the checkpointing technique described above is that effectively we must shut down the system while the checkpoint is being made. Since the active transactions may take a long time to commit or abort, the system may appear to users to be stalled. Thus, a more complex technique known as *nonquiescent checkpointing*, which allows new transactions to enter the system during the checkpoint, is usually preferred. The steps in a nonquiescent checkpoint are:

1. Write a log record <START CKPT(T_1,…,T_k)> and flush the log. Here, T_1,…,T_k are the names or identifiers for all the active transactions (i.e., transactions that have not yet committed and written their changes to disk).
2. Wait until all of T_1,…,T_k commit or abort, but do not prohibit other transactions from starting.
3. When all of T_1,…,T_k have completed, write a log record <END CKPT> and flush the log.

With a log of this type, we can recover from a system crash as follows. As usual, we scan the log from the end, finding all incomplete transactions as we go, and restoring old values for database elements changed by these transactions. There are two cases, depending on whether, scanning backwards, we first meet an <END CKPT> record or a <START CKPT(T_1,…,T_k)> record:

- If we first meet an <END CKPT> record, then we know that all incomplete transactions began after the previous <START CKPT(T_1,…,T_k)> record. We may thus scan backwards as far as the next START CKPT and then stop; previous log is useless and may as well have been discarded.
- If we first meet a record <START CKPT(T_1,…,T_k)>, then the crash occurred during the checkpoint. However, the only incomplete transactions are those we met scanning backwards before we reached the START CKPT and those of T_1,…,T_k that did not complete before the crash. Thus, we need scan no further back than the start of the earliest of these incomplete transactions. The previous START CKPT record with a corresponding END CKPT is certainly prior to any of these transaction starts, but often we shall find the starts of the incomplete transactions long before we reach the previous checkpoint. If the previous START CKPT has no corresponding END CKPT record, then it means that another crash also occurred during a checkpoint. Such incomplete checkpoints must be ignored. Moreover, if we use

pointers to chain together the log records that belong to the same transaction, then we need not search the whole log for records belonging to active transactions; we just follow their chains back through the log.

As a general rule, once an <END CKPT> record has been written to disk, we can delete the log prior to the previous START CKPT record.

Example. Consider the following log:

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>
<START CKPT(T_1,T_2)>
<T_2,C,15>
<START T_3>
<T_1,D,20>
<COMMIT T_1>
<T_3,E,25>
<COMMIT T_2>
<END CKPT>
<T_3,F,30>

Now, we decide to do a nonquiescent checkpoint after the fourth entry. Since T_1 and T_2 are the active (incomplete) transactions at this time, we write the fifth log record. Suppose that while waiting for T_1 and T_2 to complete, another transaction, T_3, initiates.

Suppose that at the end of the listing, there is a system crash. Examining the log from the end, we find that T_3 is an incomplete transaction and must be undone. The final log record tells us to restore database element F to the value 30. When we find the <END CKPT> record, we know that all incomplete transactions began after the previous START CKPT. Scanning further back, we find the record <T_3,E,25>, which tells us to restore E to value 25. Between that record and the START CKPT, there are no other transactions that started but did not commit, so no further changes to the database are made.

Now suppose the crash occurs during the checkpoint, and the end of the log after the crash is the <T_3,E,25> record. Scanning backwards, we identify T_3 and then T_2 as incomplete transactions and undo changes they have made. When we find the <START CKPT(T_1,T_2)> record, we know that the only other possible incomplete transaction is T_1. However, we have already scanned the <COMMIT T_1> record, so we know that T_1 is not incomplete. Also, we have already seen the <START T_3> record. Thus, we need only to continue backwards until we meet the START record for T_2, restoring database element B to value 10 as we go.

Redo Logging

Undo logging has a potential problem that we cannot commit a transaction without first writing all its changed data to disk. Sometimes, we can save disk I/O's if we let changes to the database reside only in main memory for a while. As long as there is a log to fix things up in the event of a crash, it is safe to do so.

The requirement for immediate backup of database elements to disk can be avoided if we use a logging mechanism called *redo logging*. The principal differences between redo and undo logging are:

- While undo logging cancels the effect of incomplete transactions and ignores committed ones during recovery, redo logging ignores incomplete transactions and repeats the changes made by committed transactions.
- While undo logging requires us to write changed database elements to disk before the COMMIT log record reaches disk, redo logging requires that the COMMIT record appear on disk before any changed values reach disk.
- While the old values of changed database elements are exactly what we need to recover when the undo rules U_1 and U_2 are followed, to recover using redo logging, we need the new values instead.

The Redo Logging Rule

In redo logging, an update log record is formally the same as in undo logging: <T,X,v>, but here it means "transaction T wrote new value v for database element X." There is no indication of the old value of X in this record. Every time a transaction T modifies a database element X, a record of the form <T,X,v> must be written to the log.

For redo logging, the order in which data and log entries reach disk can be described by a single "redo rule:"

R_1: Before modifying any database element X on disk, it is necessary that all log records pertaining to this modification of X, including both the update record <T,X,v> and the <COMMIT T> record, must appear on disk.

The COMMIT record for a transaction can only be written to the log when the transaction completes, so the commit record must follow all the update log records. Thus, when redo logging is in use, the order in which material associated with one transaction gets written to disk is:

1. the log records indicating changed database elements;
2. the COMMIT log record;
3. the changed database elements themselves.

Example. Let us consider transaction T defined earlier using redo logging:

Step	*Action*	*t*	*M*–A	*M*–B	*D*–A	*D*–B	*Log*
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T,A,16>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T,B,16>
8)							<COMMIT T>
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

The major differences between using undo and redo logging are as follows. First, we note in lines (4) and (7) that the log records reflecting the changes have the new values of A and B, rather than the old values. Second, we see that the <COMMIT T> record comes earlier, at step (8). Then, the log is flushed, so all log records involving the changes of transaction T appear on disk. Only then can the new values of A and B be written to disk. We show these values written immediately, at steps (10) and (11), although in practice, they might occur later.

Recovery with Redo Logging

An important consequence of the redo rule R_1 is that unless the log has a <COMMIT T> record, we know that no changes to the database made by transaction T have been written to disk. Thus, incomplete transactions may be treated during recovery as if they had never occurred. However, the committed transactions present a problem, since we do not know which of their database changes have been written to disk. Fortunately, the redo log has exactly the information we need: the new values, which we may write to disk regardless of whether they were already there. To recover, using a redo log, after a system crash, we do the following:

1. Identify the committed transactions.
2. Scan the log forward from the beginning. For each log record <T,X,v> encountered:
   a) If T is not a committed transaction, do nothing.
   b) If T is committed, write value v for database element X.
3. For each incomplete transaction T, write an <ABORT T> record to the log and flush the log.

Example. Let us consider the log written in the above example and see how recovery would be performed if the crash occurred after different steps in that sequence of actions:

1. If the crash occurs any time after step (9), then the <COMMIT T> record has been flushed to disk. The recovery system identifies T as a committed transaction. When scanning the log forward, the log records <T,A,16> and <T,B,16> cause the recovery manager to write values 16 for A and B. Notice that if the crash occurred between steps (10) and (11), then the write of A is redundant, but the write of B had not occurred and changing B to 16 is essential to restore the database state to consistency. If the crash occurred after step (11), then both writes are redundant but harmless.

2. If the crash occurs between steps (8) and (9), then although the record <COMMIT T> was written to the log, it may not have gotten to disk (depending on whether the log was flushed for some other reason). If it did get to disk, then the recovery proceeds as in case (1), and if it did not get to disk, then recovery is as in case (3), below.

3. If the crash occurs prior to step (8), then <COMMIT T> surely has not reached disk. Thus, T is treated as an incomplete transaction. No changes to A or B on disk are made on behalf of T, and eventually an <ABORT T> record is written to the log.

Since several committed transactions may have written new values for the same database element X, we have required that during a redo recovery, we scan the log from earliest to latest. Thus, the final value of X in the database will be the one written last, as it should be. Similarly, when describing undo recovery, we required that the log be scanned from latest to earliest. Thus, the final value of X will be the value that it had before any of the incomplete transactions changed it.

Checkpointing a Redo Log

Redo logs present a checkpointing problem that we do not see with undo logs. Since the database changes made by a committed transaction can be copied to disk much later than the time at which the transaction commits, we cannot limit our concern to transactions that are active at the time we decide to create a checkpoint. Regardless of whether the checkpoint is quiescent or nonquiescent, between the start and end of the checkpoint, we must write to disk all database elements that have been modified by committed transactions. To do so requires that the buffer manager keep track of which buffers are *dirty*, that is, they have been changed but not written to disk. It is also required to know which transactions modified which buffers.

On the other hand, we can complete the checkpoint without waiting for the active transactions to commit or abort, since they are not allowed to write their pages to disk at that time anyway. The steps to perform a nonquiescent checkpoint of a redo log are as follows:

1. Write a log record <START CKPT(T_1,…,T_k)>, where T_1,…,T_k are all the active (uncommitted) transactions, and flush the log.
2. Write to disk all database elements that were written to buffers but not yet to disk by transactions that had already committed when the START CKPT record was written to the log.
3. Write an <END CKPT> record to the log and flush the log.

Example. Consider the following log:

<START T_1>
<T_1,A,5>
<START T_2>
<COMMIT T_1>
<T_2,B,10>
<START CKPT(T_2)>
<T_2,C,15>
<START T_3>
<T_3,D,20>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

When we start the checkpoint, only T_2 is active, but the value of A written by T_1 may have reached disk. If not, then we must copy A to disk before the checkpoint can end. We suggest the end of the checkpoint occurring after several other events have occurred: T_2 wrote a value for database element C, and a new transaction T_3 started and wrote a value of D. After the end of the checkpoint, the only things that happen are that T_2 and T_3 commit.

Recovery with a Checkpointed Redo Log

As for an undo log, the insertion of records to mark the start and end of a checkpoint helps us limit our examination of the log when a recovery is necessary. Also as with undo logging, there are two cases, depending on whether the last checkpoint record is START or END:

- Suppose first that the last checkpoint record on the log before a crash is <END CKPT>. Now, we know that every value written by a transaction that committed before the corresponding <START CKPT(T_1,…,T_k)> has had its changes written to disk, so we need not concern ourselves with recovering the effects of these transactions. However, any transaction that is either among the T_i's or that started after the beginning of the checkpoint can still have changes it made not yet migrated to disk, even though the transaction has committed. Thus, we must perform recovery as described earlier but may limit our attention to the transactions that are either one of the T_i's mentioned in the last <START CKPT(T_1,…,T_k)> or that started after that log record appeared in the log. In searching the log, we do not have to look further back than the earliest of the <START T_i> records. Notice, however, that these START records could appear prior to any number of checkpoints. Linking backwards all the log records for a given transaction helps us to find the necessary records, as it did for undo logging.
- Now, suppose the last checkpoint record on the log is <START CKPT(T_1,…,T_k)>. We cannot be sure that committed transactions prior to the start of this checkpoint had their changes written to disk. Thus, we must search back to the previous <END CKPT> record, find its matching

<START CKPT(S_1,…,S_m)> record, and redo all those committed transactions that either started after that START CKPT or are among the S_i's.

Example. Consider again the log from the previous example. If a crash occurs at the end, we search backwards, finding the <END CKPT> record. We thus know that it is sufficient to consider as candidates to redo all those transactions that either started after the <START CKPT(T_2)> record was written or that are on its list (i.e., T_2). Thus, our candidate set is (T_2, T_3). We find the records <COMMIT T_2> and <COMMIT T_3>, so we know that each must be redone. We search the log as far back as the <START T_2> record, and find the update records <T_2,B,10>, <T_2,C,15>, and <T_3,D,20> for the committed transactions. Since we don't know whether these changes reached disk, we rewrite the values 10, 15, and 20 for B, C, and D, respectively.

Now, suppose the crash occurred between the records <COMMIT T_2> and <COMMIT T_3>. The recovery is similar to the above, except that T_3 is no longer a committed transaction. Thus, its change <T_3,D,20> must not be redone, and no change is made to D during recovery, even though that log record is in the range of records that is examined. Also, we write an <ABORT T_3> record to the log after recovery.

Finally, suppose that the crash occurs just prior to the <END CKPT> record. In principal, we must search back to the next-to-last START CKPT record (with a corresponding <END CKPT>) and get its list of active transactions. However, in this case there is no previous checkpoint, and we must go all the way to the beginning of the log. Thus, we identify T_1 as the only committed transaction, redo its action <T_1,A,5>, and write records <ABORT T_2> and <ABORT T_3> to the log after recovery.

Since transactions may be active during several checkpoints, it is convenient to include in the <START CKPT(T_1,…,T_k)> records not only the names of the active transactions but pointers to the place on the log where they started. By doing so, we know when it is safe to delete early portions of the log. When we write an <END CKPT>, we know that we shall never need to look back further than the earliest of the <START T_i> records for the active transactions T_i. Thus, anything prior to that START record may be deleted.

Undo/Redo logging

We have seen two different approaches to logging, differentiated by whether the log holds old values or new values when a database element is updated. Each has certain drawbacks:

- Undo logging requires that data be written to disk immediately after a transaction finishes, perhaps increasing the number of disk I/O's that need to be performed.
- On the other hand, redo logging requires us to keep all modified blocks in buffers until the transaction commits and the log records have been flushed, perhaps increasing the average number of buffers required by transactions.
- Both undo and redo logs may put contradictory requirements on how buffers are handled during a checkpoint, unless the database elements are complete blocks or sets of blocks. For instance, if a buffer contains one database element A that was changed by a committed transaction and another database element B that was changed in the same buffer by a transaction that has not yet had its COMMIT record written to disk, then we are required to copy the buffer to disk because of A but also forbidden to do so, because rule R_1 applies to B.

We shall now see a kind of logging called *undo/redo logging*, which provides increased flexibility to order actions, at the expense of maintaining more information on the log.

The Undo/Redo Rules

An undo/redo log has the same sorts of log records as the other kinds of log, with one exception. The update log record that we write when a database element changes value has four components. Record <T,X,v,w> means that transaction T changed the value of database element X; its former value was v, and its new value is w. The constraints that an undo/redo logging system must follow are summarized by the following rule:

UR_1: Before modifying any database element X on disk because of changes made by some transaction T, it is necessary that the update record <T,X,v,w> appear on disk.

Rule UR_1 for undo/redo logging thus enforces only the constraints enforced by both undo logging and redo logging. In particular, the <COMMIT T> log record can precede or follow any of the changes to the database elements on disk.

Example. Let us consider again transaction T defined earlier using undo/redo logging:

Step	*Action*	*t*	*M*–A	*M*–B	*D*–A	*D*–B	*Log*
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T,A,8,16>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T,B,8,16>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)							<COMMIT T>
11)	OUTPUT(B)	16	16	16	16	16	

Notice that the log records for updates now have both the old and the new values of A and B. In this sequence, we have written the <COMMIT T> log record in the middle of the output of database elements A and B to disk. Step (10) could also have appeared before step (8) or step (9), or after step (11).

Recovery with Undo/Redo Logging

When we need to recover using an undo/redo log, we have the information in the update records either to undo a transaction T by restoring the old values of the database elements that T changed, or to redo T by repeating the changes it has made. The undo/redo recovery policy is:

1. Redo all the committed transactions in the order earliest-first, and
2. undo all the incomplete transactions in the order latest-first.

Notice that it is necessary for us to do both. Because of the flexibility allowed by undo/redo logging regarding the relative order in which COMMIT log records and the database changes themselves are copied to disk, we could have either a committed transaction with some or all of its changes not on disk, or an uncommitted transaction with some or all of its changes on disk.

Example. Here are the different ways that recovery would take place on the assumption that there is a crash at various points in the sequence:

1. Suppose the crash occurs after the <COMMIT T> record is flushed to disk. Then T is identified as a committed transaction. We write the value 16 for both A and B to the disk. Because of the actual order of events, A already has the value 16, but B may not, depending on whether the crash occurred before or after step (11).

2. If the crash occurs prior to the <COMMIT T> record reaching disk, then T is treated as an incomplete transaction. The previous values of A and B, 8 in each case, are written to disk. If the crash occurs between steps (9) and (10), then the value of A was 16 on disk, and the restoration to value 8 is necessary. In this example, the value of B does not need to be undone, and if the crash occurs before step (9), then neither does the value of A. However, in general we cannot be sure whether restoration is necessary, so we always perform the undo operation.

Like undo logging, a system using undo/redo logging can exhibit a behavior where a transaction appears to the user to have been completed (e.g., they booked an airline seat over the Web and disconnected), and yet because the <COMMIT T> record was not flushed to disk, a subsequent crash causes the transaction to be undone rather than redone. If this possibility is a problem, we suggest the use of an additional rule for undo/redo logging:

UR$_2$: A <COMMIT T> record must be flushed to disk as soon as it appears in the log.

For instance, we would add FLUSH LOG after step (10) in the example above.

You may have noticed that we did not specify whether undo's or redo's are done first during recovery using an undo/redo log. In fact, whether we perform the redo's or undo's first, we are open to the following situation: a transaction T has committed and is redone; however, T wrote a value X written also by some transaction U that has not committed and is undone. The problem is not whether we redo first, and leave X with its value prior to U, or we undo first and leave X with its value written by T. The situation makes no sense either way, because the final database state does not correspond to the effect of any sequence of atomic transactions.

In reality, the DBMS must do more than log changes. It must assure that such situations do not occur at all. We will later see a discussion about the means to isolate transactions like T and U, so the interaction between them through database element X cannot occur. We will explicitly address means for preventing this situation where T reads a "dirty" value of X — one that has not been committed.

Checkpointing an Undo/Redo Log

A nonquiescent checkpoint is somewhat simpler for undo/redo logging than for the other logging methods. We have only to do the following:

1. Write a <START CKPT(T_1,…,T_k)> record to the log, where T_1,…,T_k are all the active transactions, and flush the log.
2. Write to disk all the buffers that are dirty; i.e., they contain one or more changed database elements. Unlike redo logging, we flush all dirty buffers, not just those written by committed transactions.
3. Write an <END CKPT> record to the log, and flush the log.

Notice in connection with point (2) that, because of the flexibility undo/redo logging offers regarding when data reaches disk, we can tolerate the writing to disk of data written by incomplete transactions. Therefore, we can tolerate database elements that are smaller than complete blocks and thus may share buffers.

Example. Consider the following log:

<START T_1>
<T_1,A,4,5>
<START T_2>
<COMMIT T_1>
<T_2,B,9,10>
<START CKPT(T_2)>
<T_2,C,14,15>

<START T_3>
<T_3,D,19,20>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

The example is analogous to the example for redo logging. We have changed only the update records, giving them an old value as well as a new value. For simplicity, we have assumed that in each case, the old value is one less than the new value.

When the checkpoint begins, T_2 is identified as the only active transaction. Since this log is an undo/redo log, it is possible that T_2's new B-value 10 has been written to disk, which was not possible under redo logging. However, it is irrelevant whether or not that disk write has occurred. During the checkpoint, we shall surely flush B to disk if it is not already there, since we flush all dirty buffers. Likewise, we shall flush A, written by the committed transaction T_1, if it is not already on disk.

- If the crash occurs at the end of this sequence of events, then T_2 and T_3 are identified as committed transactions. Transaction T_1 is prior to the checkpoint. Since we find the <END CKPT> record on the log, T_1 is correctly assumed to have both completed and had its changes written to disk. We therefore redo both T_2 and T_3, and ignore T_1. However, when we redo a transaction such as T_2, we do not need to look prior to the <START CKPT(T_2)> record, even though T_2 was active at that time, because we know that T_2's changes prior to the start of the checkpoint were flushed to disk during the checkpoint.
- For another instance, suppose the crash occurs just before the <COMMIT T_3> record is written to disk. Then we identify T_2 as committed but T_3 as incomplete. We redo T_2 by setting C to 15 on disk; it is not necessary to set B to 10 since we know that change reached disk before the <END CKPT>. However, unlike the situation with a redo log, we also undo T_3; that is, we set D to 19 on disk. If T_3 had been active at the start of the checkpoint, we would have had to look as far back as the earliest <START T_i> record for all T_i listed in the START CKPT record to find if there were more actions by T_i (now T_2 and T_3) that may have reached disk and need to be undone. For redo recovery, however, we again do not need to look prior to the START CKPT record.
- If the crash occurs prior to the <END CKPT> record, then the last START CKPT record is ignored, and then we do the same as described above.

Protecting Against Media Failures

The log can protect us against system failures, where nothing is lost from disk, but temporary data in main memory is lost. However, more serious failures involve the loss of one or more disks. Theoretically, the database can be recovered after a media failure with the help of the log if

- the disk storing the log is different from the disk(s) containing the database;
- the log is never truncated after creating a checkpoint;
- the log is of type redo or undo/redo, containing the new values of database elements.

However, the log may increase in a faster rate than the database, so it is not a good practice to save the log forever. An archiving system, which we cover next, is needed to enable a database to survive losses involving disk-resident data.

The Archive

To protect against media failures, we are thus led to a solution involving *archiving* — maintaining a copy of the database separate from the database itself. If it were possible to shut down the database for a while, we could make a backup copy on some storage medium such as tape or optical disk and store the copy remote from the database, in some secure location. The backup would preserve the database state as it existed at the time of the backup, and if there were a media failure, the database could be restored to this state.

To advance to a more recent state, we could use the log, provided the log had been preserved since the archive copy was made, and the log itself survived the failure. In order to protect against losing the log, we could transmit a copy of the log, almost as soon as it is created, to the same remote site as the archive. Then, if the log as well as the data is lost, we can use the archive plus remotely stored log to recover, at least up to the point that the log was last transmitted to the remote site.

Since writing an archive is a lengthy process, we try to avoid copying the entire database at each archiving step. Thus, we distinguish between two levels of archiving:

- a *full dump*, in which the entire database is copied;
- an *incremental dump*, in which only those database elements changed since the previous full or incremental dump are copied.

It is also possible to have several levels of dump, with a full dump thought of as a "level 0" dump, and a "level *i*" dump copying everything changed since the last dump at a level less than or equal to *i*. After creating a new level *i* dump, dumps at higher levels can be deleted or ignored during a recovery.

We can restore the database from a full dump and its subsequent incremental dumps in a process much like the way a redo or undo/redo log can be used to repair damage due to a system failure. We copy the full dump back to the database, and then in an earliest-first order, make the changes recorded by the later incremental dumps. In case of multilevel dumps, dumps at levels higher than 0 are processed in an increasing order of levels, and dumps at the same level are processed in chronological order.

We might question the need for an archive, since we have to back up the log in a secure place anyway if we are not to be stuck at the state the database was in when the previous archive was made. While it may not be obvious, the answer lies in the typical rate of change of a large database. While only a small fraction of the database may change in a day, the changes, each of which must be logged, will over the course of a year become much larger than the database itself. If we never archived, then the log could never be truncated, and the cost of storing the log would soon exceed the cost of storing a copy of the database.

Nonquiescent Archiving

The problem with the simple view of archiving described above is that most databases cannot be shut down for the period of time (possibly hours) needed to make a backup copy. We thus need to consider *nonquiescent archiving*, which is analogous to nonquiescent checkpointing. Recall that a nonquiescent checkpoint attempts to make a copy on the disk of the (approximate) database state that existed when the checkpoint started. We can rely on a small portion of the log around the time of the checkpoint to fix up any deviations from that database state, due to the fact that during the checkpoint, new transactions may have started and written to disk.

Similarly, a nonquiescent dump tries to make a copy of the database that existed when the dump began, but database activity may change many database elements on disk during the minutes or hours that the dump takes. If it is necessary to restore the database from the archive, the log entries made during the dump can be used to sort things out and get the database to a consistent state. In other words, a checkpoint gets data

from memory to disk, and the log allows recovery from system failure, whereas a dump gets data from disk to archive, and the archive with the log allows recovery from media failure.

A nonquiescent dump copies the database elements in some fixed order, possibly while those elements are being changed by executing transactions. As a result, the value of a database element that is copied to the archive may or may not be the value that existed when the dump began. As long as the log for the duration of the dump is preserved, the discrepancies can be corrected from the log.

Example. For a very simple example, suppose that our database consists of four elements, A, B, C, and D, which have the values 1 through 4, respectively, when the dump begins. During the dump, A is changed to 5, C is changed to 6, and B is changed to 7. However, the database elements are copied in order, and the sequence of events are the following:

Disk	*Archive*
	Copy A
A := 5	
	Copy B
C := 6	
	Copy C
B := 7	
	Copy D

Then, although the database at the beginning of the dump has values (1,2,3,4), and the database at the end of the dump has values (5,7,6,4), the copy of the database in the archive has values (1,2,6,4), a database state that existed at no time during the dump.

In more detail, the process of making an archive can be broken into the following steps. We assume that the logging method is either redo or undo/redo; an undo log is not suitable for use with archiving (see the discussion after the example for more details).

1. Write a log record <START DUMP>.
2. Perform a checkpoint appropriate for whichever logging method is being used.
3. Perform a full or incremental dump of the data disk(s), as desired, making sure that the copy of the data has reached the secure, remote site.
4. Make sure that enough of the log has been copied to the secure, remote site so that at least the prefix of the log up to and including the checkpoint in item (2) will survive a media failure of the database.
5. Write a log record <END DUMP>.

At the completion of the dump, it is safe to throw away log that is not required according to the recovery rules concerning the checkpoint performed in item (2) above.

Example. Suppose that the changes to the simple database introduced above were caused by two transactions T_1 (which writes A and B) and T_2 (which writes C) that were active when the dump began. The following listing shows a possible undo/redo log of the events during the dump.

<START DUMP>
<START CKPT(T_1,T_2)>
<T_1,A,1,5>
<T_2,C,3,6>
<COMMIT T_2>
<T_1,B,2,7>
<END CKPT>
dump completes
<END DUMP>

Notice that we did not show T_1 committing. It would be unusual that a transaction remained active during the entire time a full dump was in progress, but that possibility doesn't affect the correctness of the recovery method that we discuss next.

Now, we can see why undo log cannot be used with nonquiescent archiving. Suppose a T_3 transaction starts after the <START CKPT(T_1,T_2)> record, which writes A, then B, and then completes, so a <COMMIT T_3> record is written to the log, but only after the <END CKPT> record, i.e., during backup. Since, in case of undo logging, OUTPUT actions can execute at any time after the update record is written to the log, it may happen that A is copied after its value is changed, and B is copied before its value is changed. During recovery, T_3 will be ignored, as its COMMIT record is found in the log. Thus, we get a result as if T_3 had not been executed atomically. Using redo logging, such situations may not happen, because OUTPUT actions may only execute after the COMMIT record is written to the log. This way, either no changes are made on disk (if there is no COMMIT record) or the transaction is redone (if there is a COMMIT record). In case of undo/redo logging, each transaction is undone (if there is no COMMIT record) or redone (if there is a COMMIT record), so there can be no nonatomic behavior.

Recovery Using an Archive and Log

Suppose that a media failure occurs, and we must reconstruct the database from the most recent archive and whatever prefix of the log has reached the remote site and has not been lost in the crash. We perform the following steps:

1. Restore the database from the archive:
    a) Find the most recent full dump and reconstruct the database from it (i.e., copy the archive into the database).
    b) If there are later incremental dumps, modify the database according to each, earliest first. In case of multilevel dumps, apply each dump of each level, beginning with level 1 (in order of levels, and in chronological order within one level).
2. Modify the database using the surviving log. Use the method of recovery appropriate to the log method being used.

Example. Suppose there is a media failure after the dump of the above example completes, and the log survives. Assume, to make the process interesting, that the surviving portion of the log does not include a <COMMIT T_1> record, although it does include the <COMMIT T_2> record. The database is first restored to the values in the archive, which is, for database elements A, B, C, and D, respectively, (1,2,6,4).

Now, we must look at the log. Since T_2 has completed, we redo the step that sets C to 6. In this example, C already had the value 6, but it might be that

- the archive for C was made before T_2 changed C, or
- the archive actually captured a later value of C, which may or may not have been written by a transaction whose commit record survived. Later in the recovery, C will be restored to the value found in the archive if the transaction was committed.

Since T_1 does not have a COMMIT record, we must undo T_1. We use the log records for T_1 to determine that A must be restored to value 1 and B to 2. It happens that they had these values in the archive, but the actual archive value could have been different if the modified A and/or B had been included in the archive. (It depends on the order of the update and the backup of these elements.)

The Logging and Backup System of Oracle Database

The following information comes from Oracle Database Administrator's Guide and Oracle Database Backup and Recovery User's Guide.

The Redo Log

After instance failure (system failure of a single instance), Oracle uses the online redo log files to perform automatic recovery of the database. *Instance recovery* occurs as soon as the instance starts up again after it has failed or shut down abnormally. The most crucial structure for recovery operations is the *redo log*, which stores all changes made to the database as they occur. Every instance of an Oracle Database has an associated redo log to protect the database in case of an instance failure. It consists of two parts: online and archived redo log.

An *online redo log* consists of two or more online redo log files, which are filled with *redo records*. A redo record, also called a *redo entry*, is made up of a group of *change vectors*, each of which is a description of a change made to a single block in the database. For example, if you change a salary value in a table containing employee-related data, you generate a redo record containing change vectors that describe changes to the data segment block for the table, the undo segment data block, and the transaction table of the undo segment (see later). Redo records are buffered in a circular fashion in the redo log buffer of the SGA (System Global Area) and are written to one of the redo log files by the Log Writer (LGWR) database background process. (The SGA holds also the buffers for database elements; those buffers are written to disk by the Database Writer background process.) Whenever a transaction is committed, LGWR writes the transaction redo records from the redo log buffer of the SGA to a redo log file, and assigns a *system change number* (SCN) to identify the redo records for each committed transaction. Only when all redo records associated with a given transaction are safely on disk in the online logs is the user process notified that the transaction has been committed. Redo records can also be written to a redo log file before the corresponding transaction is committed. If the redo log buffer fills, or another transaction commits, LGWR flushes all of the redo log entries in the redo log buffer to a redo log file, even though some redo records may not be committed. If necessary, the database can roll back these changes.

The online redo log for a database consists of two or more redo log files. Oracle Database uses only one redo log file at a time to store redo records written from the redo log buffer. The redo log file that LGWR is actively writing to is called the *current* redo log file. Redo log files that are required for instance recovery (i.e., not all changes recorded in them have been written to the data files yet) are called *active* redo log files. Redo log files that are no longer required for instance recovery (i.e., the changes recorded in them have been written to the data files) are called *inactive* redo log files. The database requires a minimum of two files to guarantee that one is always available for writing while the other is being archived (if the database is in ARCHIVELOG mode). LGWR writes to redo log files in a circular fashion. When the current redo log file fills, LGWR begins writing to the next available redo log file. When the last available redo log file is filled, LGWR returns to the first redo log file and writes to it, starting the cycle again. Filled redo log files are available to LGWR for reuse depending on whether archiving is enabled. If archiving is disabled (the database is in NOARCHIVELOG mode), a filled redo log file is available after it becomes inactive. If archiving is enabled (the database is in ARCHIVELOG mode), a filled redo log file is available to LGWR after it becomes inactive *and* the file has been archived by one of the archiver background processes (ARC).

A *log switch* is the point at which the database stops writing to one redo log file and begins writing to another. Normally, a log switch occurs when the current redo log file is completely filled and writing must continue to the next redo log file. However, you can configure log switches to occur at regular intervals, regardless of whether the current redo log file is completely filled. You can also force log switches manually. Oracle Database assigns each redo log file a new *log sequence number* every time a log switch occurs and LGWR begins writing to it. When the database archives redo log files, the archived log retains

its log sequence number. A redo log file that is cycled back for use is given the next available log sequence number. Each online or archived redo log file is uniquely identified by its log sequence number. During crash, instance, or media recovery, the database properly applies redo log files in ascending order by using the log sequence number of the necessary archived and online redo log files.

To protect against a failure involving the redo log itself, Oracle Database allows a *multiplexed* redo log, meaning that two or more identical copies of the redo log can be automatically maintained in separate locations. When redo log files are multiplexed, LGWR concurrently writes the same redo log information to multiple identical redo log files, thereby eliminating a single point of redo log failure. For the most benefit, these copies should be on separate disks. However, even if all copies of the redo log are on the same disk, the redundancy can help protect against I/O errors, file corruption, and so on.

As we mentioned, Oracle Database lets you save filled groups of redo log files to one or more offline destinations, known collectively as the *archived redo log*. The process of turning redo log files into archived redo log files is called *archiving*. This process is only possible if the database is running in ARCHIVELOG mode. You can choose automatic or manual archiving.

When you run your database in NOARCHIVELOG mode, you disable the archiving of the redo log. The database control file indicates that filled redo log files are not required to be archived. Therefore, when a filled redo log file becomes inactive after a log switch, the file is available for reuse by LGWR. NOARCHIVELOG mode protects a database from instance failure but not from media failure. Only the most recent changes made to the database, which are stored in the online redo log files, are available for instance recovery. If a media failure occurs while the database is in NOARCHIVELOG mode, you can only restore the database to the point of the most recent full database backup. You cannot recover transactions subsequent to that backup. In NOARCHIVELOG mode, you cannot perform online tablespace backups, nor can you use online tablespace backups taken earlier while the database was in ARCHIVELOG mode. To restore a database operating in NOARCHIVELOG mode, you can use only whole database backups taken while the database is closed. Therefore, if you decide to operate a database in NOARCHIVELOG mode, take whole database backups at regular, frequent intervals.

When you run a database in ARCHIVELOG mode, you enable the archiving of the redo log. The database control file indicates that a group of filled redo log files cannot be reused by LGWR until the group is archived. A filled group becomes available for archiving immediately after a redo log switch occurs. The archiving of filled groups has these advantages:

- A database backup, together with online and archived redo log files, guarantees that you can recover all committed transactions in the event of an operating system or disk failure.
- If you keep archived logs available, you can use a backup taken while the database is open and in normal system use.
- You can keep a standby database current with its original database by continuously applying the original archived redo log files to the standby.

Oracle Database uses the online redo log only for recovery. However, administrators can query online redo log files through an SQL interface in the Oracle *LogMiner* utility. Redo log files are a useful source of historical information about database activity.

Every Oracle Database has a *control file*, which is a small binary file that records the physical structure of the database. The control file includes:

- the database name,
- names and locations of associated data files and redo log files,
- the timestamp of the database creation,
- the current log sequence number,
- checkpoint information.

The control file must be available for writing by the Oracle Database server whenever the database is open. Without the control file, the database cannot be mounted and recovery is difficult. The control file of an Oracle Database is created at the same time as the database. By default, at least one copy of the control file is created during database creation. On some operating systems the default is to create multiple copies. Every Oracle Database should have at least two control files, each stored on a different physical disk (*multiplexed control file*). If a control file is damaged due to a disk failure, the associated instance must be shut down. Once the disk drive is repaired, the damaged control file can be restored using the intact copy of the control file from the other disk, and the instance can be restarted. In this case, no media recovery is required.

Undo Management

Oracle uses a special combination of undo and redo logging. As we have seen, information for redo recovery (the new values of database blocks) is stored in the redo log. Information for undo recovery, however, are stored in one or more *undo tablespaces* by default (or in *rollback segments* placed in other tablespaces; see later). This means that Oracle Database stores undo data inside the database rather than in external logs. Undo data is stored in blocks that are updated just like data blocks, with changes to these blocks generating redo records. In this way, Oracle Database can efficiently access undo data without needing to read external logs. Undo tablespaces record the old values of data that was changed by each transaction (whether or not committed). Oracle uses undo data to roll back an active transaction, recover a terminated transaction, provide read consistency, and perform some logical flashback operations.

An undo tablespace consists of *undo segments*, which consist of *undo records* or *undo entries*. The contents of an undo entry include the address of changed data column(s), the transaction operation performing the change, an SQL statement that undoes the effect of the change, and the old value(s) of each changed column. Undo entries are always written to disk before the corresponding modified data reach disk. Undo entries for each transaction are linked so that they can be located easily for undo activities.

Each undo segment of an undo tablespace has a corresponding *transaction table*, which holds the transaction identifiers of the transactions using the undo segment. The transaction table is made up of a fixed number of slots or entries. This figure depends on the size of a data block, which is defined by the operating system. Each of these slots is assigned to a transaction and contains information about that action. The slots are initially used in order but are reused in a round-robin fashion with one exception: a slot referencing an uncommitted transaction will not be reused. It is possible to fill up all slots with active transactions. If this occurs, the transaction waits until a slot becomes available.

After a transaction is committed, undo data is no longer needed for rollback or transaction recovery purposes. However, for consistent read purposes, long-running queries may require this old undo information for producing older images of data blocks (see chapter titled *Concurrency Control in Oracle Database*). Furthermore, the success of several Oracle Flashback features can also depend upon the availability of older undo information. For these reasons, it is desirable to retain the old undo information for as long as possible.

An auto-extending undo tablespace named UNDOTBS1 is automatically created when you create the database with Database Configuration Assistant (DBCA). You can also create an undo tablespace explicitly, using the CREATE DATABASE or CREATE UNDO TABLESPACE statement. When the database instance starts, the database automatically selects the first available undo tablespace. If no undo tablespace is available, then the instance starts without an undo tablespace and stores undo records in the SYSTEM tablespace. This is not recommended, and an alert message is written to the alert log file to warn that the system is running without an undo tablespace. If the database contains multiple undo tablespaces, then you can optionally specify at startup that you want to use a specific undo tablespace. This is done by setting the UNDO_TABLESPACE initialization parameter.

Instance Recovery Phases

As changes are made to the undo segments, these changes are also written to the online redo log. It is because undo tablespace is part of the database just like other tablespaces. As a result, the online redo log always contains the undo data for permanent objects. This means that every change to the database implies the creation of an undo entry with the old value of the changed column(s), a log record with the new value of the data block containing the modified data, and another log record with the new value of the data block containing the undo entry.

The first phase of instance recovery is called *cache recovery* or *rolling forward*, and involves reapplying all of the changes recorded in the online redo log to the data files. It is enough to reconstruct changes made after the most recent checkpoint. The checkpoint position guarantees that every committed change with an SCN lower than the checkpoint SCN is saved to the data files. Checkpoints occur in a variety of situations. For example, when the Database Writer process writes dirty buffers, it advances the checkpoint position. The resulting database state after a cache recovery is very likely to be inconsistent. After the roll forward, any changes that were not committed must be undone. Because rollback data is recorded in the online redo log, rolling forward also regenerates the corresponding undo segments. Oracle Database applies undo blocks to roll back uncommitted changes in data blocks that were written before the failure or introduced during cache recovery. This phase is called *rolling back* or *transaction recovery*.

Undo Management Modes

The database can run in *automatic* or *manual undo management mode*. With automatic undo management, the database automatically manages undo segments in undo tablespaces, and no user intervention is required. Automatic undo management is the default mode for a newly installed database. In manual mode, undo space is managed through *rollback segments* (user-managed undo segments), and no undo tablespace is used. Space management for rollback segments is complex and requires hard work from the DBA.

Backup and Recovery

The focus in Oracle Database backup and recovery is on the physical backup of database files, which permits you to reconstruct your database. RMAN, a command-line tool, is the method preferred by Oracle for efficiently backing up and recovering your Oracle database. The files protected by the backup and recovery facilities built into RMAN include data files, control files, server parameter files, and archived redo log files. With these files you can reconstruct your database. RMAN is designed to work intimately with the server, providing block-level corruption detection during backup and restore. RMAN optimizes performance and space consumption during backup with file multiplexing and backup set compression, and integrates with leading tape and storage media products. The backup mechanisms work at the physical level to protect against file damage, such as the accidental deletion of a data file or the failure of a disk drive. RMAN can also be used to perform point-in-time recovery to recover from logical failures when other techniques such as flashback cannot be used.

In NOARCHIVELOG mode, the filled redo log groups that become inactive can be reused. This mode protects the database against instance failure, but not against media failure. In ARCHIVELOG mode, filled groups of redo logs are archived. This mode protects the database from both instance and media failure, but may require additional hardware resources.

A *full backup* of a data file includes all used blocks of the data file. An *incremental backup* copies only those blocks in a data file that change between backups. A *level 0 incremental backup*, which copies all blocks in the data file, is used as a starting point for an incremental backup strategy. A *level 1 incremental backup* copies only images of blocks that have changed since the previous level 0 or level 1 incremental

backup. Level 1 backups can be *cumulative*, in which case all blocks changed since the most recent level 0 backup are included, or *differential*, in which case only blocks changed since the most recent level 0 or level 1 incremental backup are included. A typical incremental strategy makes level 1 backups at regular intervals such as once each day. During recovery, RMAN will automatically apply both incremental backups and redo logs as required, to recover the database to the exact point in time desired.

A backup is either *consistent* or *inconsistent*. A consistent backup occurs when the database is in a consistent state. A database is in a consistent state after being shut down with the SHUTDOWN NORMAL, SHUTDOWN IMMEDIATE, or SHUTDOWN TRANSACTIONAL commands. A consistent shutdown guarantees that all redo has been applied to the data files. If you mount the database and make a backup at this point, then you can restore the database backup later and open it without performing media recovery. But you will, of course, lose all transactions that occurred after the backup was created.

Any database backup that is not consistent is an inconsistent backup. A backup made when the database is open is inconsistent, as is a backup made after an instance failure or SHUTDOWN ABORT command. When a database is restored from an inconsistent backup, Oracle Database must perform *media recovery* before the database can be opened, applying changes from the redo logs that took place after the backup was created. Note that RMAN does not permit you to make inconsistent backups when the database is in NOARCHIVELOG mode. If the database runs in ARCHIVELOG mode, and you back up the archived redo logs and data files, inconsistent backups can be the foundation for a sound backup and recovery strategy. Inconsistent backups offer superior availability because you do not have to shut down the database to make backups that fully protect the database.

Media recovery requires a control file, data files (typically restored from backup), and online and archived redo log files containing changes since the time the data files were backed up. Media recovery is most often used to recover from media failure, such as the loss of a file or disk, or a user error, such as the deletion of the contents of a table.

Media recovery can be a *complete recovery* or a *point-in-time recovery*. Complete recovery can apply to individual data files, tablespaces, or the entire database. Point-in-time recovery applies to the whole database (and also sometimes to individual tablespaces, with automation help from RMAN). In a complete recovery, you restore backup data files and apply all changes from the archived and online redo log files to the data files. The database is returned to its state at the time of failure and can be opened with no loss of data. In a point-in-time recovery, you return a database to its contents at a user-selected time in the past. You restore a backup of data files created before the target time and a complete set of archived redo log files from backup creation through the target time. Recovery applies changes between the backup time and the target time to the data files. All changes after the target time are discarded.

Concurrency Control

Interactions among concurrently executing transactions can cause the database state to become inconsistent, even when the transactions individually preserve correctness of the state, and there is no system failure. Thus, the timing of individual steps of different transactions needs to be regulated in some manner. This regulation is the job of the *scheduler* component of the DBMS, and the general process of assuring that transactions preserve consistency when executing simultaneously is called *concurrency control*.

As transactions request reads and writes of database elements, these requests are passed to the scheduler. In most situations, the scheduler will execute the reads and writes directly, first calling on the buffer manager if the desired database element is not in a buffer. However, in some situations, it is not safe for the request to be executed immediately. The scheduler must delay the request; in some concurrency-control techniques, the scheduler may even abort the transaction that issued the request.

Serial and Serializable Schedules

Recall the "correctness principle": every transaction, if executed in isolation (without any other transactions running concurrently), will transform any consistent state to another consistent state. In practice, transactions often run concurrently with other transactions, so the correctness principle doesn't apply directly. This section introduces the notion of "schedules," the sequence of actions performed by transactions, and "serializable schedules," which produce the same result as if the transactions executed one-at-a-time.

Schedules

A *schedule* is a sequence of the important actions taken by one or more transactions, in which the order of actions of a particular transaction is the order given in the transaction. When studying concurrency control, the important read and write actions take place in the main-memory buffers, not the disk. That is, a database element `A` that is brought to a buffer by some transaction `T` may be read or written in that buffer not only by `T` but by other transactions that access `A`. In other words, from the point of view of concurrency control, only the order of `READ` and `WRITE` operations is considered, `INPUT` and `OUTPUT` operations are ignored.

Example. Let us consider two transactions and the effect on the database when their actions are executed in certain orders:

T_1	T_2
`READ(A,t)`	`READ(A,s)`
`t := t+100`	`s := s*2`
`WRITE(A,t)`	`WRITE(A,s)`
`READ(B,t)`	`READ(B,s)`
`t := t+100`	`s := s*2`
`WRITE(B,t)`	`WRITE(B,s)`

The variables `t` and `s` are local variables of T_1 and T_2, respectively; they are not database elements. We shall assume that the only consistency constraint on the database state is that `A` = `B`. Since T_1 adds 100 to both `A` and `B`, and T_2 multiplies both `A` and `B` by 2, we know that each transaction, run in isolation, will preserve consistency.

Serial Schedules

A schedule is *serial* if, for any two transactions T and T′ in the schedule, if there exists an action in T that precedes an action of T′, then all the actions of T precede all the actions of T′. In other words, the actions of the schedule consist of all the actions of one transaction, then all the actions of another transaction, and so on. No mixing of the actions is allowed.

Example. For the transactions above, there are two serial schedules, one in which T_1 precedes T_2 and the other in which T_2 precedes T_1. Let the initial state be A = B = 25. Then, the two schedules are the following:

T_1	T_2	A	B
READ(A,t)		25	
t := t+100			
WRITE(A,t)		125	
READ(B,t)			25
t := t+100			
WRITE(B,t)			125
	READ(A,s)	125	
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		125
	s := s*2		
	WRITE(B,s)		250

T_1	T_2	A	B
	READ(A,s)	25	
	s := s*2		
	WRITE(A,s)	50	
	READ(B,s)		25
	s := s*2		
	WRITE(B,s)		50
READ(A,t)		50	
t := t+100			
WRITE(A,t)		150	
READ(B,t)			50
t := t+100			
WRITE(B,t)			150

We shall take the convention that when displayed vertically, time proceeds down the page. Also, the values of A and B shown refer to their values in main-memory buffers, not necessarily to their values on disk.

Notice that the final values of A and B are different for the two schedules; they both have value 250 when T_1 goes first and 150 when T_2 goes first. In general, we would not expect the final state of a database to be independent of the order of transactions.

We can represent a serial schedule as in the figure above, listing each of the actions in the order they occur. However, since the order of actions in a serial schedule depends only on the order of the transactions themselves, we shall sometimes represent a serial schedule by the list of transactions as in (T_1, T_2) or (T_2, T_1).

Serializable Schedules

The correctness principle for transactions tells us that every serial schedule will preserve consistency of the database state. But are there any other schedules that also are guaranteed to preserve consistency? There are, as the following example shows. In general, we say a schedule S of some transactions is *serializable* if there is a serial schedule S′ of the same transactions such that for every initial database state, the effects of S and S′ are the same.

Example. Consider the following two schedules of the two transactions defined above:

T_1	T_2	A	B
READ(A,t)		25	
t := t+100			
WRITE(A,t)		125	
	READ(A,s)	125	
	s := s*2		
	WRITE(A,s)	250	
READ(B,t)			25
t := t+100			
WRITE(B,t)			125
	READ(B,s)		125
	s := s*2		
	WRITE(B,s)		250

T_1	T_2	A	B
READ(A,t)		25	
t := t+100			
WRITE(A,t)		125	
	READ(A,s)	125	
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		25
	s := s*2		
	WRITE(B,s)		50
READ(B,t)			50
t := t+100			
WRITE(B,t)			150

The first example shows a schedule that is serializable but not serial. In this schedule, T_2 acts on A after T_1 does, but before T_1 acts on B. However, we see that the effect of the two transactions scheduled in this manner is the same as for the serial schedule (T_1, T_2). To convince ourselves of the truth of this statement, we must consider not only the effect from the database state A = B = 25 but from any consistent database state. Since all consistent database states have A = B = c for some constant c, it is not hard to deduce that in the schedule, both A and B will be left with the value 2(c + 100), and thus consistency is preserved from any consistent state.

On the other hand, consider the schedule in the second example, which is not serializable. The reason we can be sure it is not serializable is that it takes the consistent state A = B = 25 and leaves the database in an inconsistent state, where A = 250 and B = 150. Notice that in this order of actions, where T_1 operates on A first, but T_2 operates on B first, we have in effect applied different computations to A and B, that is A := 2(A + 100) versus B := 2B + 100. This is the sort of behavior that concurrency control mechanisms must avoid.

The Effect of Transaction Semantics

In our study of serializability so far, we have considered in detail the operations performed by the transactions, to determine whether or not a schedule is serializable. The details of the transactions do matter, as we can see from the following example.

Example. Consider the following schedule, which differs from our last example only in the computation that T_2 performs. That is, instead of multiplying A and B by 2, T_2 multiplies each by 1:

T_1	T_2	A	B
READ(A,t)		25	
t := t+100			
WRITE(A,t)		125	
	READ(A,s)	125	
	s := s*1		
	WRITE(A,s)	125	
	READ(B,s)		25
	s := s*1		
	WRITE(B,s)		25
READ(B,t)			25
t := t+100			
WRITE(B,t)			125

One can easily check that regardless of the consistent initial state, the final state is the one that results from the serial schedule (T_1, T_2). Coincidentally, it also results from the other serial schedule, (T_2, T_1).

You may notice that T_2 is not meaningful. Actually, it could be replaced with any transaction that leaves A and B intact. T_2 could, for example, only print the values of A and B, or it could compute a factor F based on user input and multiply A and B by F, and for some user input, F could result in 1.

Unfortunately, it is not realistic for the scheduler to concern itself with the details of computation undertaken by transactions. Since transactions often involve code written in a general-purpose programming language as well as SQL or other high-level-language statements, it is impossible to say for certain what a transaction is doing. However, the scheduler does get to see the read and write requests from the transactions, so it can know what database elements each transaction reads, and what elements it *might* change. To simplify the job of the scheduler, it is conventional to assume that:

- Any database element A that a transaction T writes is given a value that depends on the database state in such a way that no arithmetic coincidences occur.

An example of a "coincidence" is that in our previous example, where (A + 100) * 1 = B * 1 + 100 whenever A = B, even though the two operations are carried out in different orders on the two variables. Put another way, if there is something that T could do to a database element to make the database state inconsistent, then T will do that.

A Notation for Transactions and Schedules

If we assume "no coincidences," then only the reads and writes performed by the transaction matter, not the actual values involved. Thus, we shall represent transactions and schedules by a shorthand notation, in which the actions are $r_T(X)$ and $w_T(X)$, meaning that transaction T reads, or respectively writes, database element X. Moreover, since we shall usually name our transactions T_1, T_2, …, we adopt the convention that $r_i(X)$ and $w_i(X)$ are synonyms for $r_{Ti}(X)$ and $w_{Ti}(X)$, respectively.

Example. The two transactions in the previous examples can be written the following way:

T_1: $r_1(A); w_1(A); r_1(B); w_1(B);$
T_2: $r_2(A); w_2(A); r_2(B); w_2(B);$

We did not mention the local variables t and s, and did not denote what happened to A and B after reading them. This notation is interpreted that we assume the "worst case" regarding the change of values of database elements.

As another example, consider the serializable schedule of transactions T_1 and T_2 presented earlier:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$

To make the notation precise:

1. An *action* is an expression of the form $r_i(X)$ or $w_i(X)$, meaning that transaction T_i reads or writes, respectively, the database element X.
2. A *transaction* T_i is a sequence of actions with subscript i.
3. A *schedule* S of a set of transactions *T* is a sequence of actions, in which for each transaction T_i in *T*, the actions of T_i appear in S in the same order that they appear in the definition of T_i itself. We say that S is an *interleaving* of the actions of the transactions of which it is composed.

Conflict-Serializability

Schedulers in commercial systems generally enforce a condition, called "conflict-serializability," that is stronger than the general notion of serializability. It is based on the idea of a *conflict*: a pair of consecutive actions in a schedule such that, if their order is interchanged, then the behavior of at least one of the transactions involved can change.

Conflicts

To begin, let us observe that most pairs of actions do not conflict. In what follows, we assume that T_i and T_j are different transactions; i.e., $i \neq j$.

1. $r_i(X); r_j(Y)$ is never a conflict, even if $X = Y$. The reason is that neither of these steps change the value of any database element.
2. $r_i(X); w_j(Y)$ is not a conflict provided $X \neq Y$. The reason is that should T_j write Y before T_i reads X, the value of X is not changed. Also, the read of X by T_i has no effect on T_j, so it does not affect the value T_j writes for Y.
3. $w_i(X); r_j(Y)$ is not a conflict if $X \neq Y$, for the same reason as (2).
4. Similarly, $w_i(X); w_j(Y)$ is not a conflict as long as $X \neq Y$.

On the other hand, there are three situations where we may not swap the order of actions:

a) Two actions of the same transaction, e.g., $r_i(X); w_i(Y)$, always conflict. The reason is that the order of actions of a single transaction are fixed and may not be reordered.
b) Two writes of the same database element by different transactions conflict. That is, $w_i(X); w_j(X)$ is a conflict. The reason is that as written, the value of X remains afterward as whatever T_j computed it to be. If we swap the order, as $w_j(X); w_i(X)$, then we leave X with the value computed by T_i. Our assumption of "no coincidences" tells us that the values written by T_i and T_j will be different, at least for some initial states of the database.
c) A read and a write of the same database element by different transactions also conflict. That is, $r_i(X); w_j(X)$ is a conflict, and so is $w_i(X); r_j(X)$. If we move $w_j(X)$ ahead of $r_i(X)$, then the value of X read by T_i will be that written by T_j, which we assume is not necessarily the same as the previous value of X. Thus, swapping the order of $r_i(X)$ and $w_j(X)$ affects the value T_i reads for X and could therefore affect what T_i does.

The conclusion we draw is that any two actions of different transactions may be swapped unless

1. they involve the same database element, and
2. at least one is a write.

Extending this idea, we may take any schedule and make as many nonconflicting swaps as we wish, with the goal of turning the schedule into a serial schedule. If we can do so, then the original schedule is serializable, because its effect on the database state remains the same as we perform each of the nonconflicting swaps.

We say that two schedules are *conflict-equivalent* if they can be turned one into the other by a sequence of nonconflicting swaps of adjacent actions. We shall call a schedule *conflict-serializable* if it is conflict-equivalent to a serial schedule. Note that conflict-serializability is a sufficient condition for serializability; i.e., a conflict-serializable schedule is a serializable schedule. Conflict-serializability is not required for a schedule to be serializable, but it is the condition that the schedulers in commercial systems generally use when they need to guarantee serializability.

Example. Consider the schedule

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$

We claim this schedule is conflict-serializable. Here is the sequence of swaps in which this schedule is converted to the serial schedule (T_1, T_2), where all of T_1's actions precede all those of T_2. We have underlined the pair of adjacent actions about to be swapped at each step.

$r_1(A); w_1(A); r_2(A); \underline{w_2(A); r_1(B);} w_1(B); r_2(B); w_2(B);$
$r_1(A); w_1(A); \underline{r_2(A); r_1(B);} w_2(A); w_1(B); r_2(B); w_2(B);$
$r_1(A); w_1(A); r_1(B); r_2(A); \underline{w_2(A); w_1(B);} r_2(B); w_2(B);$
$r_1(A); w_1(A); r_1(B); \underline{r_2(A); w_1(B);} w_2(A); r_2(B); w_2(B);$
$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B);$

You may ask why conflict-serializability is not necessary for serializability. We have already seen an example when serializability of a schedule could only be determined by considering the semantics of the transactions. We could see that the schedule was serializable because of the specific computations undertaken by T_2. That particular schedule, however, is *not* conflict-serializable, as A is written by T_1 first, while B is written by T_2 first. Since neither the write of A nor the write of B can be rearranged, there is no way for all of T_1's actions to precede those of T_2, nor vice versa.

There are serializable, but not conflict-serializable schedules that do not depend on the computations undertaken by the transactions. Consider three transactions T_1, T_2, and T_3 that each write a value for X. T_1 and T_2 also write values for Y before they write values for X. One possible schedule, which happens to be serial, is

S_1: $w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X);$

S_1 leaves X with the value written by T_3 and Y with the value written by T_2. However, so does the schedule

S_2: $w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X);$

Intuitively, the values of X written by T_1 and T_2 have no effect, since T_3 overwrites their values. Thus, X has the same value after either S_1 or S_2, and likewise Y has the same value after either S_1 or S_2. Since S_1 is serial, and S_2 has the same effect as S_1 on any database state, we know that S_2 is serializable. However, since we cannot swap $w_1(Y)$ with $w_2(Y)$, and we cannot swap $w_1(X)$ with $w_2(X)$, therefore we cannot convert S_2 to any serial schedule by swaps. That is, S_2 is serializable, but not conflict-serializable.

Precedence Graphs and a Test for Conflict-Serializability

It is relatively simple to examine a schedule S and decide whether or not it is conflict-serializable. When a pair of conflicting actions appears anywhere in S, the transactions performing those actions must appear in the same order in any conflict-equivalent serial schedule as the actions appear in S. Thus, conflicting pairs of actions put constraints on the order of transactions in the hypothetical, conflict-equivalent serial schedule. If these constraints are not contradictory, we can find a conflict-equivalent serial schedule. If they are contradictory, we know that no such serial schedule exists.

Given a schedule S, involving transactions T_1 and T_2 ($T_1 \neq T_2$), perhaps among other transactions, we say that T_1 *takes precedence over* T_2, written $T_1 <_S T_2$, if there are actions A_1 of T_1 and A_2 of T_2, such that

1. A_1 is ahead of A_2 in S,
2. both A_1 and A_2 involve the same database element, and
3. at least one of A_1 and A_2 is a write action.

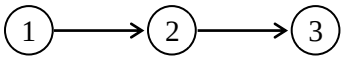
In other words, A_1 and A_2 would conflict if they were consecutive actions. Notice that these are exactly the conditions under which we cannot swap the order of A_1 and A_2. Thus, A_1 will appear before A_2 in any schedule that is conflict-equivalent to S. As a result, a conflict-equivalent serial schedule must have T_1 before T_2.

We can summarize these precedences in a *precedence graph*. The nodes of the precedence graph are the transactions of a schedule S. When the transactions are T_i for various i, we shall label the node for T_i by only the integer i. There is an arc from node i to node j if $T_i <_S T_j$.

Example. The following schedule S involves three transactions, T_1, T_2, and T_3:

S: $r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$

The precedence graph for schedule S is the following:



If we look at the actions involving A, we find several reasons why $T_2 <_S T_3$. For example, $r_2(A)$ comes ahead of $w_3(A)$ in S, and $w_2(A)$ comes ahead of both $r_3(A)$ and $w_3(A)$. Any one of these three observations is sufficient to justify the arc in the precedence graph from 2 to 3. Similarly, if we look at the actions involving B, we find that there are several reasons why $T_1 <_S T_2$. For instance, the action $r_1(B)$ comes before $w_2(B)$. Thus, the precedence graph for S also has an arc from 1 to 2. However, these are the only arcs we can justify from the order of actions in schedule S.

There is a simple rule to determine whether a schedule S is conflict-serializable:

- To tell whether a schedule S is conflict-serializable, construct the precedence graph for S and ask if there are any cycles. If so, then S is not conflict-serializable. But if the graph is acyclic, then S is conflict-serializable, and moreover, any topological order of the nodes is a conflict-equivalent serial order.

A topological order of an acyclic graph is any order of the nodes such that for every arc $a \rightarrow b$, node a precedes node b in the topological order. We can find a topological order for any acyclic graph by repeatedly removing nodes that have no predecessors among the remaining nodes.

Example. The precedence graph above is acyclic, so schedule S is conflict-serializable. There is only one order of the nodes or transactions consistent with the arcs of that graph: (T_1, T_2, T_3). Notice that it is indeed possible to convert S into the schedule in which all actions of each of the three transactions occur in this order; this serial schedule is:

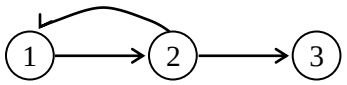
S': $r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B); r_3(A); w_3(A);$

To see that we can get from S to S' by swaps of adjacent elements, first notice we can move $r_1(B)$ ahead of $r_2(A)$ without conflict. Then, by three swaps we can move $w_1(B)$ just after $r_1(B)$, because each of the intervening actions involves A and not B. We can then move $r_2(B)$ and $w_2(B)$ to a position just after $w_2(A)$, moving through only actions involving A; the result is S'.

Example. Consider the following schedule:

S_1: $r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$

S_1 differs from S only in that action $r_2(B)$ has been moved forward three positions. Examination of the actions involving A still give us only the precedence $T_2 <_{S1} T_3$. However, when we examine B, we get not only $T_1 <_{S1} T_2$ (because $r_1(B)$ and $w_1(B)$ appear before $w_2(B)$) but also $T_2 <_{S1} T_1$ (because $r_2(B)$ appears before $w_1(B)$). Thus, we have the following precedence graph for schedule S_1:



This graph evidently has a cycle. We conclude that S_1 is not conflict-serializable. Intuitively, any conflict-equivalent serial schedule would have to have T_1 both ahead of and behind T_2, so therefore no such schedule exists.

Why the Precedence-Graph Test Works

If there is a cycle involving n transactions $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$, then in the hypothetical serial order, the actions of T_1 must precede those of T_2, which precede those of T_3, and so on, up to T_n. But the actions of T_n, which therefore come after those of T_1, are also required to precede those of T_1 because of the arc $T_n \rightarrow T_1$. Thus, if there is a cycle in the precedence graph, then the schedule is not conflict-serializable.

The converse is a bit harder. We must show that if the precedence graph has no cycles, then we can reorder the schedule's actions using legal swaps of adjacent actions, until the schedule becomes a serial schedule. If we can do so, then we have our proof that every schedule with an acyclic precedence graph is conflict-serializable. The proof is an induction on the number of transactions involved in the schedule:

Basis: If $n = 1$, i.e., there is only one transaction in the schedule, then the schedule is already serial, and therefore surely conflict-serializable.

Induction: Let the schedule S consist of the actions of n transactions T_1, T_2, …, T_n. We suppose that S has an acyclic precedence graph. If a finite graph is acyclic, then there is at least one node that has no arcs in; let the node i corresponding to transaction T_i be such a node. Since there are no arcs into node i, there can be no action A in S that

1. involves any transaction T_j other than T_i,
2. precedes some action of T_i, and
3. conflicts with that action.

For if there were, we should have put an arc from node j to node i in the precedence graph.

It is thus possible to swap all the actions of T_i, keeping them in order, but moving them to the front of S. The schedule has now taken the form

(actions of T_i) (actions of the other $n - 1$ transactions)

Let us now consider the tail of S — the actions of all transactions other than T_i. Since these actions maintain the same relative order that they did in S, the precedence graph for the tail is the same as the precedence graph for S, except that the node for T_i and any arcs out of that node are missing.

Since the original precedence graph was acyclic, and deleting nodes and arcs cannot make it cyclic, we conclude that the tail's precedence graph is acyclic. Moreover, since the tail involves $n - 1$ transactions, the inductive hypothesis applies to it. Thus, we know we can reorder the actions of the tail using legal swaps of adjacent actions to turn it into a serial schedule. Now, S itself has been turned into a serial schedule, with the actions of T_i first and the actions of the other transactions following in some serial order. The induction is complete, and we conclude that every schedule with an acyclic precedence graph is conflict-serializable.

Enforcing Serializability by Locks

In this section, we consider the most common architecture for a scheduler, one in which *locks* are maintained on database elements to prevent unserializable behavior. Intuitively, a transaction obtains locks on the database elements it accesses to prevent other transactions from accessing these elements at roughly the same time and thereby incurring the risk of unserializability.

First, we introduce the concept of locking with an (overly) simple locking scheme. In this scheme, there is only one kind of lock, which transactions must obtain on a database element if they want to perform any operation whatsoever on that element. Later, we shall learn more realistic locking schemes, with several kinds of lock, including the common shared/exclusive locks that correspond to the privileges of reading and writing, respectively.

Locks

Recall from the chapter introduction that the responsibility of the scheduler is to take requests from transactions and either allow them to operate on the database or block the transaction until such time as it is safe to allow it to continue. A *lock table* will be used to guide this decision in a manner that we shall discuss at length.

Ideally, a scheduler would forward a request if and only if its execution cannot possibly lead to an inconsistent database state after all active transactions commit or abort. A *locking scheduler*, like most types of scheduler, instead enforces conflict-serializability, which as we learned is a more stringent condition than correctness, or even than serializability.

When a scheduler uses locks, transactions must request and release locks, in addition to reading and writing database elements. The use of locks must be proper in two senses, one applying to the structure of transactions, and the other to the structure of schedules:

- *Consistency of Transactions*: Actions and locks must relate in the expected ways:
    1. A transaction can only read or write an element if it previously was granted a lock on that element and hasn't yet released the lock.
    2. If a transaction locks an element, it must later unlock that element.
- *Legality of Schedules*: Locks must have their intended meaning: no two transactions may have locked the same element without one having first released the lock.

We shall extend our notation for actions to include locking and unlocking actions:

$l_i(X)$: Transaction T_i *requests a lock* on database element X.
$u_i(X)$: Transaction T_i *releases* ("*unlocks*") *its lock* on database element X.

Thus, the consistency condition for transactions can be stated as:

- Whenever a transaction T_i has an action $r_i(X)$ or $w_i(X)$, then there is a previous action $l_i(X)$ with no intervening action $u_i(X)$, and there is a subsequent $u_i(X)$.

The legality of schedules is stated:

- If there are actions $l_i(X)$ followed by $l_j(X)$ in a schedule, then somewhere between these actions there must be an action $u_i(X)$.

Example. Let us consider the two transactions T_1 and T_2 that we introduced earlier. Recall that T_1 adds 100 to database elements A and B, while T_2 doubles them. Here are specifications for these transactions, in which we have included lock actions as well as arithmetic actions to help us remember what the transactions are doing (remember that the actual computations of the transaction usually are not represented in our

current notation, since they are not considered by the scheduler when deciding whether to grant or deny transaction requests):

T_1: $l_1(A); r_1(A); A := A+100; w_1(A); u_1(A); l_1(B); r_1(B); B := B+100; w_1(B); u_1(B);$
T_2: $l_2(A); r_2(A); A := A*2; w_2(A); u_2(A); l_2(B); r_2(B); B := B*2; w_2(B); u_2(B);$

Each of these transactions is consistent. They each release the locks on A and B that they take. Moreover, they each operate on A and B only at steps where they have previously requested a lock on that element and have not yet released the lock.

T_1	T_2	A	B
$l_1(A); r_1(A);$		25	
A := A+100;			
$w_1(A); u_1(A);$		125	
	$l_2(A); r_2(A);$	125	
	A := A*2;		
	$w_2(A); u_2(A);$	250	
	$l_2(B); r_2(B);$		25
	B := B*2;		
	$w_2(B); u_2(B);$		50
$l_1(B); r_1(B);$			50
B := B+100;			
$w_1(B); u_1(B);$			150

The figure above shows one legal schedule of these two transactions. The schedule is legal because the two transactions never hold a lock on A at the same time, and likewise for B. Specifically, T_2 does not execute $l_2(A)$ until after T_1 executes $u_1(A)$, and T_1 does not execute $l_1(B)$ until after T_2 executes $u_2(B)$. As we see from the trace of the values computed, the schedule, although legal, is not serializable. We shall soon see the additional condition, "two-phase locking," that we need to assure that legal schedules are conflict-serializable.

The Locking Scheduler

It is the job of a scheduler based on locking to grant requests if and only if the request will result in a legal schedule. If a request is not granted, the requesting transaction is delayed; it waits until the scheduler grants its request at a later time. To aid its decisions, the scheduler has a *lock table* that tells, for every database element, the transaction (if any) that currently holds a lock on that element. We shall later discuss the structure of a lock table in more detail. However, when there is only one kind of lock, as we have assumed so far, the table may be thought of as a relation Locks(element, transaction), consisting of pairs (X,T) such that transaction T currently has a lock on database element X. The scheduler has only to query and modify this relation using simple INSERT and DELETE statements.

Example. The schedule in the previous example is legal, as we mentioned, so the locking scheduler would grant every request in the order of arrival shown. However, sometimes it is not possible to grant requests. Here are T_1 and T_2 with simple but important changes, in which T_1 and T_2 each lock B before releasing the lock on A:

T_1: $l_1(A); r_1(A); A := A+100; w_1(A); l_1(B); u_1(A); r_1(B); B := B+100; w_1(B); u_1(B);$
T_2: $l_2(A); r_2(A); A := A*2; w_2(A); l_2(B); u_2(A); r_2(B); B := B*2; w_2(B); u_2(B);$

T_1	T_2	A	B
$l_1(A)$; $r_1(A)$;		25	
A := A+100;			
$w_1(A)$; $l_1(B)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;	125	
	A := A*2;		
	$w_2(A)$;	250	
	$l_2(B)$; **denied**		
$r_1(B)$; B := B+100;			25
$w_1(B)$; $u_1(B)$;			125
	$l_2(B)$; $u_2(A)$; $r_2(B)$;		125
	B := B*2;		
	$w_2(B)$; $u_2(B)$;		250

As you can see in the figure, when T_2 requests a lock on B, the scheduler must deny the lock, because T_1 still holds a lock on B. Thus, T_2 is delayed, and the next actions are from T_1. Eventually, T_1 executes $u_1(B)$, which unlocks B. Now, T_2 can get its lock on B, which is executed at the next step. Notice that because T_2 was forced to wait, it wound up multiplying B by 2 after T_1 added 100, resulting in a consistent database state.

Two-Phase Locking

There is a surprising condition, called *two-phase locking* (or *2PL*) under which we can guarantee that a legal schedule of consistent transactions is conflict-serializable:

- In every transaction, all lock actions precede all unlock actions.

The "two phases" referred to by 2PL are thus the first phase, where locks are obtained, and the second phase, where locks are relinquished. Two-phase locking is a condition, like consistency, on the order of actions in a transaction. A transaction that obeys the 2PL condition is said to be a *two-phase-locked transaction*, or *2PL transaction*.

Example. In our first example, the transactions do not obey the two-phase locking rule. For instance, T_1 unlocks A before it locks B. However, the versions of the transactions found in the second example do obey the 2PL condition. Notice that T_1 locks both A and B within the first five actions and unlocks them within the next five actions; T_2 behaves similarly. If we compare the two figures, we see how the 2PL transactions interact properly with the scheduler to assure consistency, while the non-2PL transactions allow non-conflict-serializable behavior.

Why Two-Phase Locking Works

Intuitively, each two-phase-locked transaction may be thought to execute in its entirety at the instant it issues its first unlock request. Thus, there is always at least one conflict-equivalent serial schedule for a schedule S of 2PL transactions: the one in which the transactions appear in the same order as their first unlocks.

We shall show how to convert any legal schedule S of consistent, two-phase-locked transactions to a conflict-equivalent serial schedule. The conversion is best described as an induction on n, the number of transactions in S. In what follows, it is important to remember that the issue of conflict-equivalence refers to the read and write actions only. As we swap the order of reads and writes, we ignore the lock and unlock actions. Once we have the read and write actions ordered serially, we can place the lock and unlock actions around them as the various transactions require. Since each transaction releases all locks before its end, we know that the serial schedule is legal.

Basis: If $n = 1$, i.e., the schedule consists of only one transaction, there is nothing to do; S is already a serial schedule.

Induction: Suppose S involves n transactions T_1, T_2, …, T_n, and let T_i be the transaction with the first unlock action in the entire schedule S, say $u_i(X)$. We claim it is possible to move all the read and write actions of T_i forward to the beginning of the schedule without passing any conflicting reads or writes.

Consider some action of T_i, say $w_i(Y)$. Could it be preceded in S by some conflicting action, say $w_j(Y)$? If so, then in schedule S, actions $u_j(Y)$ and $l_i(Y)$ must intervene, in a sequence of actions

…; $w_j(Y)$; …; $u_j(Y)$; …; $l_i(Y)$; …; $w_i(Y)$; …

Since T_i is the first to unlock, $u_i(X)$ precedes $u_j(Y)$ in S; that is, S might look like this:

…; $w_j(Y)$; …; $u_i(X)$; …; $u_j(Y)$; …; $l_i(Y)$; …; $w_i(Y)$; …

$u_i(X)$ could even appear before $w_j(Y)$. In any case, $u_i(X)$ appears before $l_i(Y)$, which means that T_i is not two-phase-locked, as we assumed. While we have only argued the nonexistence of conflicting pairs of writes, the same argument applies to any pair of potentially conflicting actions, one from T_i and the other from another T_j.

We conclude that it is indeed possible to move all the actions of T_i forward to the beginning of S, using swaps of nonconflicting read and write actions, followed by restoration of the lock and unlock actions of T_i. That is, S can be written in the form

(actions of T_i) (actions of the other $n - 1$ transactions)

The tail of $n - 1$ transactions is still a legal schedule of consistent, 2PL transactions, so the inductive hypothesis applies to it. We convert the tail to a conflict-equivalent serial schedule, and now all of S has been shown conflict-serializable.

A Risk of Deadlock

One problem that is not solved by two-phase locking is the potential for *deadlocks*, where several transactions are forced by the scheduler to wait "forever" for a lock held by another transaction. For instance, consider our familiar 2PL transactions, but with T_2 changed to work on B first:

T_1: $l_1(A); r_1(A); A := A+100; w_1(A); l_1(B); u_1(A); r_1(B); B := B+100; w_1(B); u_1(B);$
T_2: $l_2(B); r_2(B); B := B*2; w_2(B); l_2(A); u_2(B); r_2(A); A := A*2; w_2(A); u_2(A);$

A possible interleaving of the actions of these transactions is:

T_1	T_2	A	B
$l_1(A); r_1(A);$		25	
	$l_2(B); r_2(B);$		25
A := A+100;			
	B := B*2;		
$w_1(A);$		125	
	$w_2(B);$		50
$l_1(B);$ **denied**	$l_2(A);$ **denied**		

Now, neither transaction can proceed, and they wait forever. You can observe that it is not possible to allow both transactions to proceed, since if we do so, the final database state cannot possibly have $A = B$.

When a deadlock exists, it is generally impossible to repair the situation so that all transactions involved can proceed. Thus, at least one of the transactions will have to be aborted and restarted.

There are two broad approaches to dealing with deadlocks. We can detect deadlocks and fix them (*deadlock detection*), or we can manage transactions in such a way that deadlocks are never able to form (*deadlock prevention*).

The simplest way to detect and resolve deadlocks is with a *timeout*. Put a limit on how long a transaction may be active, and if a transaction exceeds this time, roll it back. For example, in a simple transaction system, where typical transactions execute in milliseconds, a timeout of one minute would affect only transactions that are caught in a deadlock.

Notice that when one deadlocked transaction times out and rolls back, it releases its locks or other resources. Thus, there is a chance that the other transactions involved in the deadlock will complete before reaching their timeout limits. However, since transactions involved in a deadlock are likely to have started at approximately the same time (or else, one would have completed before another started), it is also possible that spurious timeouts of transactions that are no longer involved in a deadlock will occur.

A more sophisticated way for deadlock detection is by a *waits-for graph*, indicating which transactions are waiting for locks held by another transaction. This graph can be used either to detect deadlocks after they have formed or to prevent deadlocks from ever forming. We shall assume the latter, which requires us to maintain the waits-for graph at all times, refusing to allow an action that creates a cycle in the graph.

We will see that a lock table maintains for each database element X a list of the transactions that are waiting for locks on X, as well as transactions that currently hold locks on X. The waits-for graph has a node for each transaction that currently holds any lock or is waiting for one. There is an arc from node (transaction) T to node U if there is some database element A such that

1. U holds a lock on A,
2. T is waiting for a lock on A, and
3. T cannot get a lock on A unless U first releases its lock on A.

If there are no cycles in the waits-for graph, then each transaction can complete eventually. There will be at least one transaction waiting for no other transaction, and this transaction surely can complete. At that time, there will be at least one other transaction that is not waiting, which can complete, and so on.

However, if there is a cycle, then no transaction in the cycle can ever make progress, so there is a deadlock. Thus, a strategy for deadlock avoidance is to roll back any transaction that makes a request that would cause a cycle in the waits-for graph.

Example. Suppose we have the following four transactions, each of which reads one element and writes another:

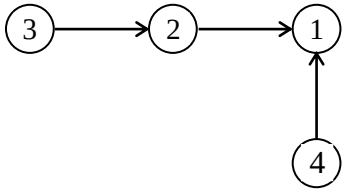
T_1: $l_1(A); r_1(A); l_1(B); w_1(B); u_1(A); u_1(B);$
T_2: $l_2(C); r_2(C); l_2(A); w_2(A); u_2(C); u_2(A);$
T_3: $l_3(B); r_3(B); l_3(C); w_3(C); u_3(B); u_3(C);$
T_4: $l_4(D); r_4(D); l_4(A); w_4(A); u_4(D); u_4(A);$

Step	T_1	T_2	T_3	T_4
1)	$l_1(A); r_1(A);$			
2)		$l_2(C); r_2(C);$		
3)			$l_3(B); r_3(B);$	
4)				$l_4(D); r_4(D);$
5)		$l_2(A);$ **denied**		
6)			$l_3(C);$ **denied**	
7)				$l_4(A);$ **denied**
8)	$l_1(B);$ **denied**			

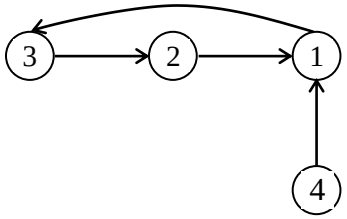
In the figure above, you can see the beginning of a schedule of these four transactions. In the first four steps, each transaction obtains a lock on the element it wants to read. At step (5), T_2 tries to lock A, but the

request is denied because T_1 already has a lock on A. Thus, T_2 waits for T_1, and we draw an arc from the node for T_2 to the node for T_1.

Similarly, at step (6) T_3 is denied a lock on C because of T_2, and at step (7), T_4 is denied a lock on A because of T_1. The waits-for graph at this point is as follows:

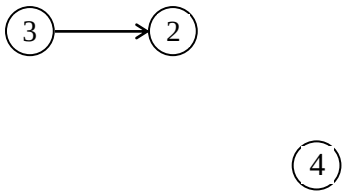


There is no cycle in this graph. At step (8), T_1 must wait for the lock on B held by T_3. If we allow T_1 to wait, there is a cycle in the waits-for graph involving T_1, T_2, and T_3, as seen in the following figure:



Since each of these transactions is waiting for another to finish, none can make progress, and therefore, there is a deadlock involving these three transactions. Incidentally, T_4 cannot finish either, although it is not in the cycle, because T_4's progress depends on T_1 making progress.

Since we roll back any transaction that causes a cycle, T_1 must be rolled back, yielding the following waits-for graph:



T_1 relinquishes its lock on A, which may be given to either T_2 or T_4. Suppose it is given to T_2. Then T_2 can complete, whereupon it relinquishes its locks on A and C. Now T_3, which needs a lock on C, and T_4, which needs a lock on A, can both complete. At some time, T_1 is restarted, but it cannot get locks on A and B until T_2, T_3, and T_4 have completed.

Locking Systems With Several Lock Modes

The locking scheme described previously illustrates the important ideas behind locking, but it is too simple to be a practical scheme. The main problem is that a transaction T must take a lock on a database element X even if it only wants to read X and not write it. We cannot avoid taking the lock, because if we didn't, then another transaction might write a new value for X while T was active and cause unserializable behavior. On the other hand, there is no reason why several transactions could not read X at the same time, as long as none is allowed to write X.

Shared and Exclusive Locks

The lock we need for writing is "stronger" than the lock we need to read, since it must prevent both reads and writes. Let us therefore consider a locking scheduler that uses two different kinds of locks: *shared locks* (*read locks*) and *exclusive locks* (*write locks*). For any database element X, there can be either one exclusive lock on X, or no exclusive locks but any number of shared locks. If we want to write X, we need to have an exclusive lock on X, but if we wish only to read X, we may have either a shared or an exclusive lock on X. If we want to read X but not write it, it is better to take only a shared lock.

We shall use $sl_i(X)$ to mean "transaction T_i requests a shared lock on database element X" and $xl_i(X)$ for "T_i requests an exclusive lock on X." We continue to use $u_i(X)$ to mean that T_i unlocks X; i.e., it relinquishes whatever lock(s) it has on X.

The three kinds of requirements — consistency and 2PL for transactions, and legality for schedules — each have their counterpart for a shared/exclusive lock system. We summarize these requirements here:

1. *Consistency of transactions*: A transaction may not write without holding an exclusive lock and may not read without holding some lock. More precisely, in any transaction T_i,

   a) a read action $r_i(X)$ must be preceded by $sl_i(X)$ or $xl_i(X)$, with no intervening $u_i(X)$;
   b) a write action $w_i(X)$ must be preceded by $xl_i(X)$, with no intervening $u_i(X)$.

   All locks must be followed by an unlock of the same element.

2. *Two-phase locking of transactions*: Locking must precede unlocking. To be more precise, in any two-phase locked transaction T_i, no action $sl_i(X)$ or $xl_i(X)$ can be preceded by an action $u_i(Y)$, for any Y.

3. *Legality of schedules*: An element may either be locked exclusively by one transaction or by several in shared mode, but not both. More precisely:

   a) If $xl_i(X)$ appears in a schedule, then there cannot be a following $xl_j(X)$ or $sl_j(X)$, for some j other than i, without an intervening $u_i(X)$.
   b) If $sl_i(X)$ appears in a schedule, then there cannot be a following $xl_j(X)$, for $j \neq i$, without an intervening $u_i(X)$.

Note that we do allow one transaction to request and hold both shared and exclusive locks on the same element, provided its doing so does not conflict with the lock(s) of other transactions. If transactions know in advance their needs for locks, then only the exclusive lock would have to be requested, but if lock needs are unpredictable, then it is possible that one transaction would request both shared and exclusive locks at different times.

Example. Let us examine a possible schedule of the following two transactions, using shared and exclusive locks:

T_1: $sl_1(A); r_1(A); xl_1(B); r_1(B); w_1(B); u_1(A); u_1(B);$
T_2: $sl_2(A); r_2(A); sl_2(B); r_2(B); u_2(A); u_2(B);$

Both T_1 and T_2 read A and B, but only T_1 writes B. Neither writes A.

T_1	T_2
$sl_1(A);\ r_1(A);$	
	$sl_2(A);\ r_2(A);$
	$sl_2(B);\ r_2(B);$
$xl_1(B);$ **denied**	
	$u_2(A);\ u_2(B);$
$xl_1(B);\ r_1(B);\ w_1(B);$	
$u_1(A);\ u_1(B);$	

In the figure above is an interleaving of the actions of T_1 and T_2 in which T_1 begins by getting a shared lock on A. Then, T_2 follows by getting shared locks on both A and B. Now, T_1 needs an exclusive lock on B, since it will both read and write B. However, it cannot get the exclusive lock because T_2 already has a shared lock on B. Thus, the scheduler forces T_1 to wait. Eventually, T_2 releases the lock on B. At that time, T_1 may complete.

Notice that the resulting schedule is conflict-serializable. The conflict-equivalent serial order is (T_2, T_1), even though T_1 started first. The argument we gave earlier to show that legal schedules of consistent, 2PL transactions are conflict-serializable applies to systems with shared and exclusive locks as well. In the figure, T_2 unlocks before T_1, so we would expect T_2 to precede T_1 in the serial order.

Compatibility Matrices

If we use several lock modes, then the scheduler needs a policy about when it can grant a lock request, given the other locks that may already be held on the same database element. A *compatibility matrix* is a convenient way to describe lock-management policies. It has a row and column for each lock mode. The rows correspond to a lock that is already held on an element X by another transaction, and the columns correspond to the mode of a lock on X that is requested. The rule for using a compatibility matrix for lock-granting decisions is:

- We can grant the lock on X in mode C if and only if for every row R such that there is already a lock on X in mode R by some other transaction, there is a "yes" in column C.

Example. The following figure shows the compatibility matrix for shared (S) and exclusive (X) locks:

	S	X
S	yes	no
X	no	no

The column for S says that we can grant a shared lock on an element if the only locks held on that element currently are shared locks. The column for X says that we can grant an exclusive lock only if there are no other locks held currently. As you can see, these rules summarize the definition of the legality of schedules for this locking scheme.

Upgrading Locks

A transaction T that takes a shared lock on X is being "friendly" toward other transactions, since they are allowed to read X at the same time T is. Thus, we might wonder whether it would be friendlier still if a transaction T that wants to read and write a new value of X were first to take a shared lock on X, and only later, when T was ready to write the new value, *upgrade* the lock to exclusive (i.e., request an exclusive lock on X in addition to its already held shared lock on X). There is nothing that prevents a transaction from issuing requests for locks on the same database element in different modes. We adopt the convention that $u_i(X)$ releases all locks on X held by transaction T_i, although we could introduce mode-specific unlock actions if there were a use for them.

To be more precise: We say that transaction T *upgrades* its lock in mode L_1 to mode L_2 on database element X if

- T holds a lock in mode L_1 on X,
- T requests a lock in mode L_2 on X, and
- L_2 dominates L_1.

A lock mode L_2 *dominates* lock mode L_1 if in the compatibility matrix, L_2's row and column each have "no" in whatever positions L_1's row or column, respectively, has "no." For instance, X dominates S in case of SX locking scheme. (Actually, X dominates all lock modes in all locking schemes, because both its row and its column contain "no" in each position.)

Example. In the following example, transaction T_1 is able to perform its computation concurrently with T_2, which would not be possible had T_1 taken an exclusive lock on B initially. The two transactions are:

T_1: $sl_1(A); r_1(A); sl_1(B); r_1(B); xl_1(B); w_1(B); u_1(A); u_1(B);$
T_2: $sl_2(A); r_2(A); sl_2(B); r_2(B); u_2(A); u_2(B);$

Here, T_1 reads A and B and performs some (possibly lengthy) calculation with them, eventually using the result to write a new value of B. Notice that T_1 takes a shared lock on B first, and later, after its calculation involving A and B is finished, requests an exclusive lock on B. Transaction T_2 only reads A and B and does not write.

T_1	T_2
$sl_1(A); r_1(A);$	
	$sl_2(A); r_2(A);$
	$sl_2(B); r_2(B);$
$sl_1(B); r_1(B);$	
$xl_1(B);$ **denied**	
	$u_2(A); u_2(B);$
$xl_1(B); w_1(B);$	
$u_1(A); u_1(B);$	

The figure shows a possible schedule of actions. T_2 gets a shared lock on B before T_1 does, but on the fourth line, T_1 is also able to lock B in shared mode. Thus, T_1 has both A and B and can perform its computation using their values. It is not until T_1 tries to upgrade its lock on B to exclusive that the scheduler must deny the request and force T_1 to wait until T_2 releases its lock on B. At that time, T_1 gets its exclusive lock on B, writes B, and finishes.

Notice that had T_1 asked for an exclusive lock on B initially, before reading B, then the request would have been denied, because T_2 already had a shared lock on B. T_1 could not perform its computation without reading B, and so T_1 would have more to do after T_2 releases its locks. As a result, T_1 would finish later if it used only an exclusive lock on B than it does using the upgrading strategy.

Example. Unfortunately, indiscriminate use of upgrading introduces a new and potentially serious source of deadlocks. Suppose that T_1 and T_2 each read database element A and write a new value for A. If both transactions use an upgrading approach, first getting a shared lock on A and then upgrading it to exclusive, the sequence of events suggested in the following figure will happen whenever T_1 and T_2 initiate at approximately the same time:

T_1	T_2
$sl_1(A);$	
	$sl_2(A);$
$xl_1(A);$ **denied**	
	$xl_2(A);$ **denied**

T_1 and T_2 are both able to get shared locks on A. Then, they each try to upgrade to exclusive, but the scheduler forces each to wait because the other has a shared lock on A. Thus, neither can make progress, and they will each wait forever, or until the system discovers that there is a deadlock, aborts one of the two transactions, and gives the other the exclusive lock on A.

Update Locks

We can avoid the deadlock problem described above with a third lock mode, called *update locks*. An update lock $ul_i(X)$ gives transaction T_i only the privilege to read X, not to write X. However, only the update lock can be upgraded to a write lock later; a read lock cannot be upgraded. We can grant an update lock on X when there are already shared locks on X, but once there is an update lock on X, we prevent additional locks of any kind — shared, update, or exclusive — from being taken on X. The reason is that if we don't deny such locks, then the updater might never get a chance to upgrade to exclusive, since there would always be other locks on X (thus, update locks solve not only the above deadlock problem but also the starvation problem).

This rule leads to an asymmetric compatibility matrix, because the update (U) lock looks like a shared lock when we are requesting it and looks like an exclusive lock when we already have it. Thus, the columns for U and S locks are the same, and the rows for U and X locks are the same:

	S	X	U
S	yes	no	yes
X	no	no	no
U	no	no	no

Remember, however, that there is an additional condition regarding legality of schedules that is not reflected by this matrix: a transaction holding a shared lock but not an update lock on an element X cannot be given an exclusive lock on X, even though we do not in general prohibit a transaction from holding multiple locks on an element.

Example. The use of update locks would have no effect on the first example of the previous page. As its third action, T_1 would take an update lock on B, rather than a shared lock. But the update lock would be granted, since only shared locks are held on B, and the same sequence of actions would occur.

However, update locks fix the deadlock problem shown in the second example of the previous page. Now, both T_1 and T_2 first request update locks on A and only later take exclusive locks. Possible descriptions of T_1 and T_2 are:

T_1: $ul_1(A); r_1(A); xl_1(A); w_1(A); u_1(A);$
T_2: $ul_2(A); r_2(A); xl_2(A); w_2(A); u_2(A);$

The sequence of events corresponding to the one described there is the following:

T_1	T_2
$ul_1(A);\ r_1(A);$	
	$ul_2(A);$ **denied**
$xl_1(A);\ w_1(A);\ u_1(A);$	
	$ul_2(A);\ r_2(A);$
	$xl_2(A);\ w_2(A);\ u_2(A);$

Now, T_2, the second to request an update lock on A, is denied. T_1 is allowed to finish, and then T_2 may proceed. The lock system has effectively prevented concurrent execution of T_1 and T_2, but in this example, any significant amount of concurrent execution will result in either a deadlock or an inconsistent database state.

Increment Locks

Another interesting kind of lock that is useful in some situations is an *increment lock*. Many transactions operate on the database only by incrementing or decrementing stored values. For example, consider a transaction that transfers money from one bank account to another.

The useful property of increment actions is that they commute with each other, since if two transactions add constants to the same database element, it does not matter which goes first. On the other hand, incrementation commutes with neither reading nor writing; if you read A before or after it is incremented, you leave different values, and if you increment A before or after some other transaction writes a new value for A, you get different values of A in the database.

Let us introduce as a possible action in transactions the *increment action*, written INC(A,c). Informally, this action adds constant c to database element A, which we assume is a single number. Note that c could be negative, in which case we are really decrementing A. In practice, we might apply INC to a component of a tuple, while the tuple itself, rather than one of its components, is the lockable element. More formally, we use INC(A,c) to stand for the atomic execution of the following steps: READ(A,t); t := t+c; WRITE(A,t);.

Corresponding to the increment action, we need an *increment lock*. We shall denote the action of T_i requesting an increment lock on X by $il_i(X)$. We also use shorthand $inc_i(X)$ for the action in which transaction T_i increments database element X by some constant; the exact constant doesn't matter.

The existence of increment actions and locks requires us to make several modifications to our definitions of consistent transactions, conflicts, and legal schedules. These changes are:

a) A consistent transaction can only have an increment action on X if it holds an increment (or exclusive) lock on X at the time. An increment lock does not enable either read or write actions, however.

b) The action $inc_i(X)$ conflicts with both $r_j(X)$ and $w_j(X)$, for $j \neq i$, but does not conflict with $inc_j(X)$.

c) In a legal schedule, any number of transactions can hold an increment lock on X at any time. However, if an increment lock on X is held by some transaction, then no other transaction can hold either a shared or exclusive lock on X at the same time. These requirements are expressed by the compatibility matrix, where I represents a lock in increment mode:

	S	X	I
S	yes	no	no
X	no	no	no
I	no	no	yes

Example. Consider two transactions, each of which reads database element A and then increments B:

T_1: $sl_1(A); r_1(A); il_1(B); inc_1(B); u_1(A); u_1(B);$
T_2: $sl_2(A); r_2(A); il_2(B); inc_2(B); u_2(A); u_2(B);$

Notice that the transactions are consistent, since they only perform an incrementation while they have an increment lock, and they only read while they have a shared lock. A possible interleaving of T_1 and T_2 is the following:

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$il_2(B)$; $inc_2(B)$;
$il_1(B)$; $inc_1(B)$;	
	$u_2(A)$; $u_2(B)$;
$u_1(A)$; $u_1(B)$;	

T_1 reads A first, but then T_2 both reads A and increments B. However, T_1 is then allowed to get its increment lock on B and proceed. Notice that the scheduler did not have to delay any requests. Suppose, for instance, that T_1 increments B by A, and T_2 increments B by 2A. They can execute in either order, since the value of A does not change, and the incrementations may also be performed in either order. Put another way, we may look at the sequence of nonlock actions in the schedule; they are:

S: $r_1(A); r_2(A); inc_2(B); inc_1(B);$

We may move the last action, $inc_1(B)$, to the second position, since it does not conflict with another increment of the same element, and surely does not conflict with a read of a different element. This sequence of swaps shows that S is conflict-equivalent to the following serial schedule:

$r_1(A); inc_1(B); r_2(A); inc_2(B);$

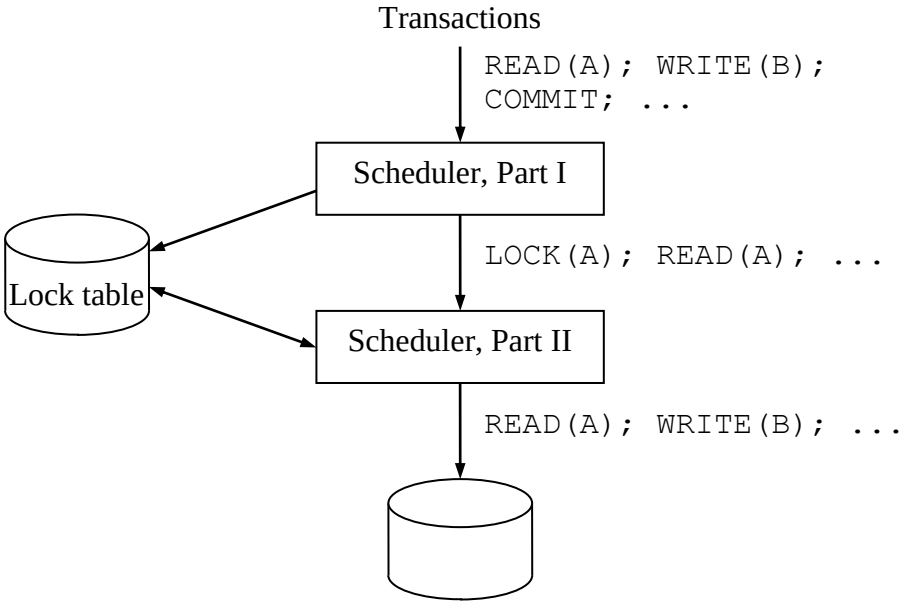
Similarly, we can move the first action, $r_1(A)$, to the third position by swaps, giving a serial schedule in which T_2 precedes T_1.

An Architecture for a Locking Scheduler

Having seen a number of different locking schemes, we next consider how a scheduler that uses one of these schemes operates. We shall consider here only a simple scheduler architecture based on several principles:

1. The transactions themselves do not request locks, or cannot be relied upon to do so. It is the job of the scheduler to insert lock actions into the stream of reads, writes, and other actions that access data.
2. Transactions do not release locks. Rather, the scheduler releases the locks when the transaction manager tells it that the transaction will commit or abort.

A Scheduler That Inserts Lock Actions



The figure shows a two-part scheduler that accepts requests such as read, write, commit, and abort, from transactions. The scheduler maintains a lock table, which, although it is shown as secondary-storage data, may be partially or completely in main memory. Normally, the main memory used by the lock table is not part of the buffer pool that is used for query execution and logging. Rather, the lock table is just another component of the DBMS and will be allocated space by the operating system like other code and internal data of the DBMS.

Actions requested by a transaction are generally transmitted through the scheduler and executed on the database. However, under some circumstances a transaction is *delayed*, waiting for a lock, and its requests are not (yet) transmitted to the database. The two parts of the scheduler perform the following actions:

1. Part I takes the stream of requests generated by the transactions and inserts appropriate lock actions ahead of all database-access operations, such as read, write, increment, or update. Part I of the scheduler must select an appropriate lock mode from whatever set of lock modes the scheduler is using. The database-access and lock actions are then transmitted to Part II.
2. Part II takes the sequence of lock and database-access actions passed to it by Part I. It determines whether the issuing transaction `T` is already delayed, because a lock has not been granted. If so, then the action is itself delayed and added to a list of actions that must eventually be performed for transaction `T`. If `T` is *not* delayed (i.e., all locks it previously requested have been granted already), then Part II checks the type of action to be executed.
   a) If the action is a database access, it is transmitted to the database and executed.
   b) If a lock action is received by Part II, it examines the lock table to see if the lock can be granted. If so, the lock table is modified to include the lock just granted. If not, then an entry must be made in the lock table to indicate that the lock has been requested. Part II of the scheduler then delays transaction `T` until such time as the lock is granted.
3. When a transaction `T` commits or aborts, Part I is notified by the transaction manager and releases all locks held by `T`. If any transactions are waiting for any of these locks, Part I notifies Part II.
4. When Part II is notified that a lock on some database element `X` is available, it determines the next transaction or transactions that can now be given a lock on `X`. The transaction(s) that receive a lock are

allowed to execute as many of their delayed actions as can execute, until they either complete or reach another lock request that cannot be granted.

Example. If there is only one kind of lock, then the task of Part I of the scheduler is simple. If it sees any action on database element X, and it has not already inserted a lock request on X for that transaction, then it inserts the request. When a transaction commits or aborts, Part I can forget about that transaction after releasing its locks, so the memory required for Part I does not grow indefinitely.

When there are several kinds of locks, the scheduler may require advance notice of what future actions on the same database element will occur. Let us reconsider the case of shared-exclusive-update locks, using the transactions of the example seen at lock upgrading, which we now write without any locks at all:

T_1: $r_1(A); r_1(B); w_1(B);$
T_2: $r_2(A); r_2(B);$

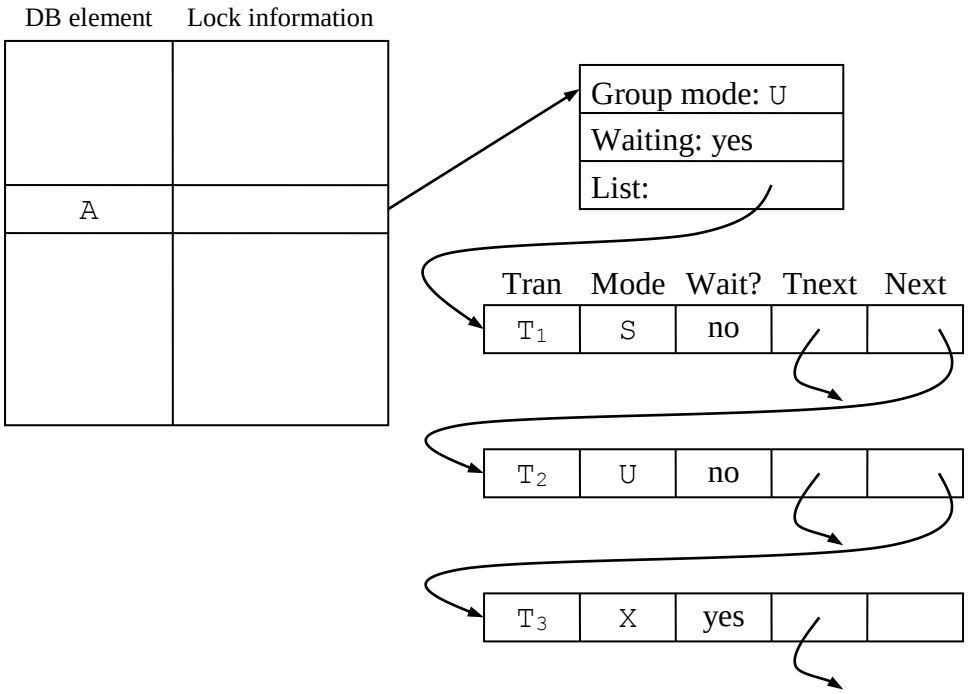
The messages sent to Part I of the scheduler must include not only the read or write request, but an indication of future actions on the same element. In particular, when $r_1(B)$ is sent, the scheduler needs to know that there will be a later $w_1(B)$ action (or might be such an action). There are several ways the information might be made available. For example, if the transaction is a query, we know it will not write anything. If the transaction is a SQL database modification command, then the query processor can determine in advance the database elements that might be both read and written. If the transaction is a program with embedded SQL, then the compiler has access to all the SQL statements (which are the only ones that can access the database) and can determine the potential database elements written.

In our example, suppose that events occur in the order suggested by the example at lock upgrading. Then T_1 first issues $r_1(A)$. Since there will be no future upgrading of this lock, the scheduler inserts $sl_1(A)$ ahead of $r_1(A)$. Next, the requests from T_2 — $r_2(A)$ and $r_2(B)$ — arrive at the scheduler. Again, there is no future upgrade, so the sequence of actions $sl_2(A); r_2(A); sl_2(B); r_2(B);$ are issued by Part I.

Then, the action $r_1(B)$ arrives at the scheduler, along with a warning that this lock may be upgraded. The scheduler Part I thus emits $ul_1(B); r_1(B);$ to Part II. The latter consults the lock table and finds that it can grant the update lock on B to T_1, because there are only shared locks on B.

When the action $w_1(B)$ arrives at the scheduler, Part I emits $xl_1(B); w_1(B);$. However, Part II cannot grant the $xl_1(B)$ request, because there is a shared lock on B for T_2. This and any subsequent actions from T_1 are delayed, stored by Part II for future execution. Eventually, T_2 commits, and Part I releases the locks on A and B that T_2 held. At that time, it is found that T_1 is waiting for a lock on B. Part II of the scheduler is notified, and it finds the lock $xl_1(B)$ is now available. It enters this lock into the lock table and proceeds to execute stored actions from T_1 to the extent possible. In this case, T_1 completes.

The Lock Table



Abstractly, the lock table is a relation that associates database elements with locking information about that element, as suggested by the figure. The table might, for instance, be implemented with a hash table, using (addresses of) database elements as the hash key. Any element that is not locked does not appear in the table, so the size is proportional to the number of locked elements only, not to the size of the entire database.

In the figure is an example of the sort of information we would find in a lock-table entry. This example structure assumes that the shared-exclusive-update lock scheme is used by the scheduler. The entry shown for a typical database element A is a tuple with the following components:

1. The *group mode* is a summary of the most stringent conditions that a transaction requesting a new lock on A faces, i.e., the group mode is the most dominant lock mode currently held on A. Rather than comparing the lock request with every lock held by another transaction on the same element, we can simplify the grant/deny decision by comparing the request with only the group mode. (The lock manager must, however, deal with the possibility that the requesting transaction already has a lock in another mode on the same element. For instance, in the SXU lock system discussed, the lock manager may be able to grant an X-lock request if the requesting transaction is the one that holds a U lock on the same element. For systems that do not support multiple locks held by one transaction on one element, the group mode always tells what the lock manager needs to know.) For the shared-exclusive-update (SXU) lock scheme, the rule is simple:

   The group mode

   a) S means that only shared locks are held;
   b) U means that there is one update lock and perhaps one or more shared locks;
   c) X means there is one exclusive lock and no other locks.

   For other lock schemes, there is usually an appropriate system of summaries by a group mode.

2. The *waiting bit* tells whether there is at least one transaction waiting for a lock on A.

3. A list describing all those transactions that either currently hold locks on A or are waiting for a lock on A. Useful information that each list entry might include:

a) the name of the transaction holding or waiting for a lock;
b) the mode of this lock;
c) whether the transaction is holding or waiting for the lock.

We also show in the figure two links for each entry. One links the entries for a particular database element, and the other (*Tnext* in the figure) links all entries for a particular transaction. The latter link would be used when a transaction commits or aborts, so that we can easily find all the locks that must be released.

Handling Lock Requests

Suppose transaction T requests a lock on A. If there is no lock-table entry for A, then surely there are no locks on A, so the entry is created and the request is granted. If the lock-table entry for A exists, we use it to guide the decision about the lock request. We find the group mode, which in the figure is U (update). Once there is an update lock on an element, no other lock can be granted (except in the case that T itself holds the U lock, and other locks are compatible with T's request). Thus, this request by T is denied, and an entry will be placed on the list saying T requests a lock (in whatever mode was requested), and the waiting bit is set to true.

If the group mode had been X (exclusive), then the same thing would happen, but if the group mode were S (shared), then another shared or update lock could be granted. In that case, the entry for T on the list would have the waiting bit false, and the group mode would be changed to U if the new lock were an update lock; otherwise, the group mode would remain S. Whether or not the lock is granted, the new list entry is linked properly, through its *Tnext* and *Next* fields. Notice that whether or not the lock is granted, the entry in the lock table tells the scheduler what it needs to know without having to examine the list of locks.

Handling Unlocks

Now suppose transaction T unlocks A. T's entry on the list for A is deleted. If the lock held by T is not the same as the group mode (e.g., T held an S lock, while the group mode was U), then there is no reason to change the group mode. On the other hand, if T's lock is in the group mode, we may have to examine the entire list to find the new group mode. In the example of the figure, we know there can be only one U lock on an element, so if that lock is released, the new group mode could be only S (if there are shared locks remaining) or nothing (if no other locks are currently held). (We would never actually see a group mode of "nothing," since if there are no locks and no lock requests on an element, then there is no lock-table entry for that element. Moreover, there may not be a lock request without a granted lock on an element.) If the group mode is X, we know there are no other locks, and if the group mode is S, we need to determine whether there are other shared locks.

If the waiting bit is true, then we need to grant one or more locks from the list of requested locks. There are several different approaches, each with its advantages:

1. *First-come-first-served*: Grant the lock request that has been waiting the longest. This strategy guarantees no *starvation*, the situation where a transaction can wait forever for a lock.
2. *Priority to shared locks*: First grant all the shared locks waiting. Then, grant one update lock, if there are any waiting. Only grant an exclusive lock if no others are waiting. This strategy can allow starvation, if a transaction is waiting for a U or X lock.
3. *Priority to upgrading*: If there is a transaction with a U lock waiting to upgrade it to an X lock, grant that first. Otherwise, follow one of the other strategies mentioned.

Hierarchies of Database Elements

Let us now return to the exploration of different locking schemes. In particular, we shall focus on two problems that come up when there is a tree structure to our data:

1. The first kind of tree structure we encounter is a hierarchy of lockable elements (lock units). We shall discuss in this section how to allow locks on both large elements, e.g., relations, and smaller elements contained within these, such as blocks holding several tuples of the relation, or individual tuples.
2. The second kind of hierarchy that is important in concurrency-control systems is data that is itself organized in a tree. A major example is B-tree indexes. We may view nodes of the B-tree as database elements, but if we do, then as we shall see in the next section, the locking schemes studied so far perform poorly, and we need to use a new approach.

Locks With Multiple Granularity

Recall that the term "database element" was purposely left undefined, because different systems use different sizes of database elements to lock, such as tuples, pages or blocks, and relations. Some applications benefit from small database elements, such as tuples, while others are best off with large elements.

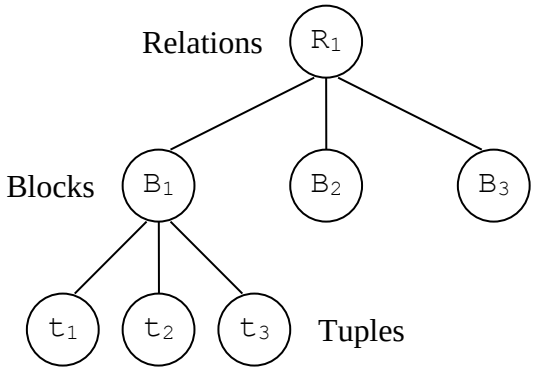
Example. Consider a database for a bank. If we treated relations as database elements, and therefore had only one lock for an entire relation such as the one giving account balances, then the system would allow very little concurrency. Since most transactions will change an account balance either positively or negatively, most transactions would need an exclusive lock on the accounts relation. Thus, only one deposit or withdrawal could take place at any time, no matter how many processors we had available to execute these transactions. A better approach is to lock individual pages or data blocks. Thus, two accounts whose tuples are on different blocks can be updated at the same time, offering almost all the concurrency that is possible in the system. The extreme would be to provide a lock for every tuple, so any set of accounts whatsoever could be updated at once, but this fine a grain of locks is probably not worth the extra effort.

In contrast, consider a database of documents. These documents may be edited from time to time, but most transactions will retrieve whole documents. The sensible choice of database element is a complete document. Since most transactions are read-only (i.e., they do not perform any write actions), locking is only necessary to avoid the reading of a document that is in the middle of being edited. Were we to use smaller-granularity locks, such as paragraphs, sentences, or words, there would be essentially no benefit but added expense. The only activity a smaller-granularity lock would support is the ability for two people to edit different parts of a document simultaneously.

Some applications could use both large- and small-grained locks. For instance, the bank database discussed in the example clearly needs block- or tuple-level locking, but might also at some time need a lock on the entire accounts relation in order to audit accounts (e.g., check that the sum of the accounts is correct). However, permitting a shared lock on the accounts relation, in order to compute some aggregation on the relation, while at the same time there are exclusive locks on individual account tuples, can lead easily to unserializable behavior. The reason is that the relation is actually changing while a supposedly frozen copy of it is being read by the aggregation query.

Warning Locks

The solution to the problem of managing locks at different granularities involves a new kind of lock called a "*warning*." These locks are useful when the database elements form a nested or hierarchical structure, as suggested in the following figure:



Here, we see three levels of database elements:

1. relations are the largest lockable elements;
2. each relation is composed of one or more blocks or pages, on which its tuples are stored;
3. each block contains one or more tuples.

The rules for managing locks on a hierarchy of database elements constitute the *warning protocol*, which involves both "ordinary" locks and "warning" locks. We shall describe the lock scheme where the ordinary locks are S and X (shared and exclusive). The warning locks will be denoted by prefixing I (for "intention to") to the ordinary locks; for example, IS represents the intention to obtain a shared lock on a subelement. The rules of the warning protocol are:

1. To place an ordinary S or X lock on any element, we must begin at the root of the hierarchy.
2. If we are at the element that we want to lock, we need look no further. We request an S or X lock on that element.
3. If the element we wish to lock is further down the hierarchy, then we place a warning at this node; that is, if we want to get a shared lock on a subelement, we request an IS lock at this node, and if we want an exclusive lock on a subelement, we request an IX lock on this node. When the lock on the current node is granted, we proceed to the appropriate child (the one whose subtree contains the node we wish to lock). We then repeat step (2) or step (3), as appropriate, until we reach the desired node.

In order to decide whether or not one of these locks can be granted, we use the following compatibility matrix:

	IS	IX	S	X
IS	yes	yes	yes	no
IX	yes	yes	no	no
S	yes	no	yes	no
X	no	no	no	no

To see why this matrix makes sense, consider first the IS column. When we request an IS lock on a node `N`, we intend to read a descendant of `N`. The only time this intent could create a problem is if some other transaction has already claimed the right to write a new copy of the entire database element represented by `N`; thus we see "no" in the row for X. Notice that if some other transaction plans to write only a subelement, indicated by an IX lock at `N`, then we can afford to grant the IS lock at `N`, and allow the conflict to be resolved at a lower level, if indeed the intent to write and the intent to read happen to involve a common element.

Now consider the column for IX. If we intend to write a subelement of node `N`, then we must prevent either reading or writing of the entire element represented by `N`. Thus, we see "no" in the entries for lock modes S and X. However, per our discussion of the IS column, another transaction that reads or writes a subelement can have potential conflicts dealt with at that level, so IX does not conflict with another IX at `N` or with an IS at `N`.

Next, consider the column for S. Reading the element corresponding to node N cannot conflict with either another read lock on N or a read lock on some subelement of N, represented by IS at N. Thus, we see "yes" in the rows for both S and IS. However, either an X or an IX means that some other transaction will write at least a part of the element represented by N. Thus, we cannot grant the right to read all of N, which explains the "no" entries in the column for S.

Finally, the column for X has only "no" entries. We cannot allow writing of all of node N if any other transaction already has the right to read or write N, or to acquire that right on a subelement.

Example. Consider the following relation:

```
Movie(title, year, length, studioName)
```

Let us postulate a lock on the entire relation and locks on individual tuples. Let transaction T_1 consist of the following query:

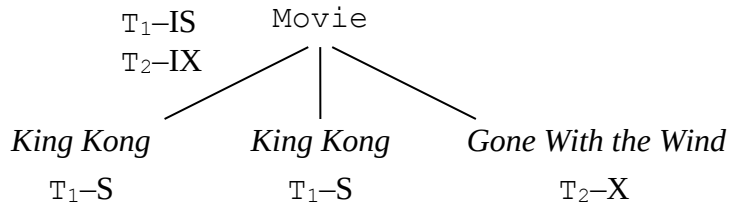
```
SELECT * FROM Movie WHERE title = 'King Kong';
```

T_1 starts by getting an IS lock on the entire relation. It then moves to the individual tuples (there are two movies with the title "King Kong"), and gets S locks on each of them.

Now, suppose that while we are executing the first query, transaction T_2, which changes the year component of a tuple, begins:

```
UPDATE Movie SET year = 1939 WHERE title = 'Gone With the Wind';
```

T_2 needs an IX lock on the relation, since it plans to write a new value for one of the tuples. T_1's IS lock on the relation is compatible, so the lock is granted. When T_2 goes to the tuple for "Gone With the Wind", it finds no lock there, and so gets its X lock and rewrites the tuple. Had T_2 tried to write a new value in the tuple for one of the "King Kong" movies, it would have had to wait until T_1 released its S lock, since S and X are not compatible. The collection of locks is suggested by the following figure:



Group Modes for Intention Locks

The compatibility matrix depicted above exhibits a situation we have not seen before regarding the power of lock modes. In prior lock schemes, whenever it was possible for a database element to be locked in two distinct modes at the same time, one of these modes dominated the other. For example, in SXU lock scheme, we see that U dominates S, and X dominates both S and U. An advantage of knowing that there is always one dominant lock on an element is that we can summarize the effect of many locks with a "group mode."

As we see from the compatibility matrix containing intention locks, neither of modes S and IX dominate the other. Moreover, it is possible for an element to be locked in both modes S and IX at the same time, provided the locks are requested by the same transaction (recall that the "no" entries in a compatibility matrix only apply to locks held by some other transaction). A transaction might request both locks if it wanted to read an entire element and then write a few of its subelements. If a transaction has both S and IX locks on an element, then it restricts other transactions to the extent that either lock does. That is, we can imagine another lock mode SIX, whose row and column have "no" everywhere except in the entry for IS.

The lock mode SIX serves as the group mode if there is a transaction with locks in S and IX modes, but not X mode.

Incidentally, we might imagine that the same situation occurs in the matrix for increment locks. That is, one transaction could hold locks in both S and I modes. However, this situation is equivalent to holding a lock in X mode, so we could use X as the group mode in that situation.

Nonrepeatable Read and Phantoms

Consider a transaction T_1 executing a query that selects some rows from a relation. Then, another transaction T_2 modifies or deletes some of the rows satisfying T_1's search condition and commits. If T_1 then attempts to reread the rows with the same search condition, it will receive modified values or discover that some rows have been deleted. This phenomenon is called *nonrepeatable read* or *fuzzy read*. The problem with nonrepeatable reads is that the second execution of the same query may give different result from its first execution. However, a transaction may expect that executing the same query more than one time gives the same result each time.

A similar situation is when transaction T_2 inserts some rows satisfying T_1's search condition into the relation (instead of modifying or deleting existing rows). Again, running the query the second time will give different result from the first time. The reason for this is that during the second execution, new rows appear that did not even exist during the first execution. Such rows are called *phantoms*.

These two phenomena (nonrepeatable read and phantom read) occur so rarely in real life that some DBMSs do not even prevent them by default, although both result in unserializable behavior. However, the user may request that nonrepeatable reads and/or phantom reads should not occur when executing a particular transaction. This is done by changing the transaction's *isolation level* (see later).

Preventing nonrepeatable reads is simple: a shared lock must be requested by T_1 on each row selected by the query. This way, T_2 cannot lock them in exclusive mode until T_1 commits or aborts. There is also a simple way for preventing phantom reads if we use locks with multiple granularity: T_2 must lock the entire relation in X mode before inserting new rows. Since T_1 previously locked the relation in IS mode, this request will be denied first by the scheduler and granted only after T_1 commits, thus preventing unserializable behavior.

The Tree Protocol

Like the previous section, this section deals with data in the form of a tree. However, here, the nodes of the tree do not form a hierarchy based on containment. Rather, database elements are disjoint pieces of data, but the only way to get to a node is through its parent; B-trees are an important example of this sort of data. Knowing that we must traverse a particular path to an element gives us some important freedom to manage locks differently from the two-phase locking approaches we have seen so far.

Motivation for Tree-Based Locking

Let us consider a B-tree index in a system that treats individual nodes (i.e., blocks) as lockable database elements. The node is the right level of lock granularity, because treating smaller pieces as elements offers no benefit, and treating the entire B-tree as one database element prevents the sort of concurrent use of the index that can be achieved via the mechanisms that form the subject of this section.

If we use a standard set of lock modes, like shared, exclusive, and update locks, and we use two-phase locking, then concurrent use of the B-tree is almost impossible. The reason is that every transaction using

the index must begin by locking the root node of the B-tree. If the transaction is 2PL, then it cannot unlock the root until it has acquired all the locks it needs, both on B-tree nodes and other database elements. Moreover, since in principle any transaction that inserts or deletes could wind up rewriting the root of the B-tree, the transaction needs at least an update lock on the root node, or an exclusive lock if update mode is not available. Thus, only one transaction that is not read-only can access the B-tree at any time.

However, in most situations, we can deduce almost immediately that a B-tree node will not be rewritten, even if the transaction inserts or deletes a tuple. For example, if the transaction inserts a tuple, but the child of the root that we visit is not completely full, then we know the insertion cannot propagate up to the root. Similarly, if the transaction deletes a single tuple, and the child of the root we visit has more than the minimum number of keys and pointers, then we can be sure the root will not change.

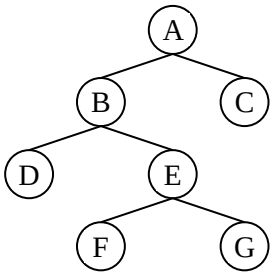
Thus, as soon as a transaction moves to a child of the root and observes the (quite usual) situation that rules out a rewrite of the root, we would like to release the lock on the root. The same observation applies to the lock on any interior node of the B-tree. Unfortunately, releasing the lock on the root early will violate 2PL, so we cannot be sure that the schedule of several transactions accessing the B-tree will be serializable. The solution is a specialized protocol for transactions that access tree-structured data such as B-trees. The protocol violates 2PL, but uses the fact that accesses to elements must proceed down the tree to assure serializability.

Rules for Access to Tree-Structured Data

The following restrictions on locks form the *tree protocol*. We assume that there is only one kind of lock, represented by lock requests of the form $l_i(X)$, but the idea generalizes to any set of lock modes. We assume that transactions are consistent, and schedules must be legal (i.e., the scheduler will enforce the expected restrictions by granting locks on a node only when they do not conflict with locks already on that node), but there is no two-phase locking requirement on transactions.

1. A transaction's first lock may be at any node of the tree. (In the B-tree example, the first lock would always be at the root because a search in a B-tree always starts at the root.)
2. Subsequent locks may only be acquired if the transaction currently has a lock on the parent node.
3. Nodes may be unlocked at any time.
4. A transaction may not relock a node on which it has released a lock, even if it still holds a lock on the node's parent.

Example. The following figure shows a hierarchy of nodes, and the table indicates the actions of three transactions on this data:



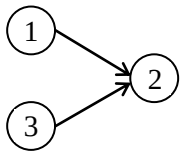
T_1	T_2	T_3
$l_1(A)$; $r_1(A)$;		
$l_1(B)$; $r_1(B)$;		
$l_1(C)$; $r_1(C)$;		
$w_1(A)$; $u_1(A)$;		
$l_1(D)$; $r_1(D)$;		
$w_1(B)$; $u_1(B)$;		
	$l_2(B)$; $r_2(B)$;	
		$l_3(E)$; $r_3(E)$;
$w_1(D)$; $u_1(D)$;		
$w_1(C)$; $u_1(C)$;		
	$l_2(E)$; **denied**	
		$l_3(F)$; $r_3(F)$;
		$w_3(F)$; $u_3(F)$;
		$l_3(G)$; $r_3(G)$;
		$w_3(E)$; $u_3(E)$;
	$l_2(E)$; $r_2(E)$;	
		$w_3(G)$; $u_3(G)$;
	$w_2(B)$; $u_2(B)$;	
	$w_2(E)$; $u_2(E)$;	

T_1 starts at the root A and proceeds downward to B, C, and D. T_2 starts at B and tries to move to E, but its move is initially denied because of the lock by T_3 on E. Transaction T_3 starts at E and moves to F and G. Notice that T_1 is not a 2PL transaction, because the lock on A is relinquished before the lock on D is acquired. Similarly, T_3 is not a 2PL transaction, although T_2 happens to be 2PL.

Why the Tree Protocol Works

The tree protocol implies a conflict-serializable order on consistent transactions involved in a legal schedule. We can define an order of precedence as follows. Say that T_i *takes precedence over* T_j ($T_i <_S T_j$) if in schedule S, the transactions T_i and T_j lock a node in common, and T_i locks the node first.

Example. In the schedule S of the example above, we find T_1 and T_2 lock B in common, and T_1 locks it first. Thus, $T_1 <_S T_2$. We also find that T_2 and T_3 lock E in common, and T_3 locks it first; thus $T_3 <_S T_2$. However, there is no precedence between T_1 and T_3, because they lock no node in common. Thus, the precedence graph derived from these precedence relations is as shown in the following figure:



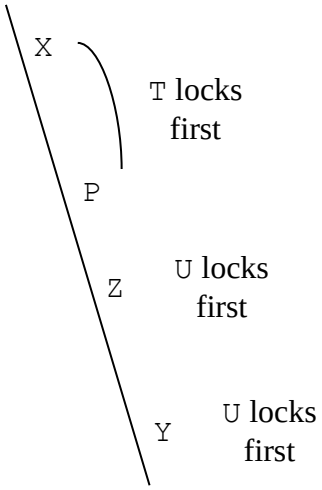
If the precedence graph drawn from the precedence relations that we defined above has no cycles, then we claim that any topological order of the transactions is an equivalent serial schedule. In our example, either (T_1, T_3, T_2) or (T_3, T_1, T_2) is an equivalent serial schedule. The reason is that in such a serial schedule, all nodes are touched in the same order as they are in the original schedule.

To understand why the precedence graph described above must always be acyclic if the tree protocol is obeyed, observe the following:

- If two transactions lock several elements in common, then they are all locked in the same order.

To see why, consider some transactions T and U, which lock two or more items in common. First, notice that each transaction locks a set of elements that form a tree, and the intersection of two trees is itself a tree. Since now T and U lock some elements in common, this intersection cannot be an empty tree. Thus, there is some one highest element X that both T and U lock. Suppose that T locks X first, but that there is some

other element Y that U locks before T. Then there is a path in the tree of elements from X to Y, and both T and U must lock each element along the path, because neither can lock a node without having a lock on its parent.



Consider the first element along this path, say Z, that U locks first, as suggested by the figure above. Then T locks the parent P of Z before U does. But then T is still holding the lock on P when it locks Z, so U has not yet locked P when it locks Z. It cannot be that Z is the first element U locks in common with T, since they both lock ancestor X (which could also be P, but not Z). Thus, U cannot lock Z until after it has acquired a lock on P, which is after T locks Z. We conclude that T precedes U at every node they lock in common.

Now, consider an arbitrary set of transactions T_1, T_2, …, T_n that obey the tree protocol and lock some of the nodes of a tree according to schedule S. First, among those that lock the root, they do so in some order, and by the rule just observed:

- If T_i locks the root before T_j, then T_i locks every node in common with T_j before T_j does. That is, $T_i <_S T_j$, but not $T_j <_S T_i$.

We can show by induction on the number of nodes of the tree that there is some serial order equivalent to S for the complete set of transactions.

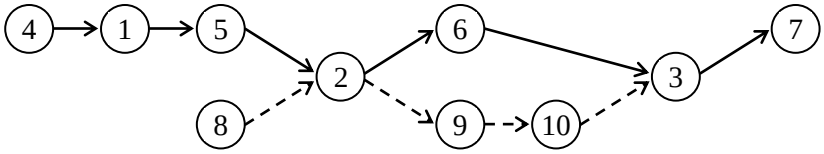
Basis: If there is only one node, the root, then as we just observed, the order in which the transactions lock the root serves.

Induction: If there is more than one node in the tree, consider for each subtree of the root the set of transactions that lock one or more nodes in that subtree. Note that transactions locking the root may belong to more than one subtree, but a transaction that does not lock the root will belong to only one subtree. For instance, among the transactions of the table above, only T_1 locks the root, and it belongs to both subtrees — the tree rooted at B and the tree rooted at C. However, T_2 and T_3 belong only to the tree rooted at B.

By the inductive hypothesis, there is a serial order for all the transactions that lock nodes in any one subtree. We have only to blend the serial orders for the various subtrees. Since the only transactions these lists of transactions have in common are the transactions that lock the root, and we established that these transactions lock every node in common in the same order that they lock the root, it is not possible that two transactions locking the root appear in different orders in two of the sublists. Specifically, if T_i and T_j appear on the list for some child C of the root, then they lock C in the same order as they lock the root and therefore appear on the list in that order. Thus, we can build a serial order for the full set of transactions by starting with the transactions that lock the root, in their appropriate order, and interspersing those transactions that do not lock the root in any order consistent with the serial order of their subtrees.

Example. Suppose there are 10 transactions T_1, T_2, …, T_{10}, and of these, T_1, T_2, and T_3 lock the root in that order. Suppose also that there are two children of the root, the first locked by T_1 through T_7 and the second locked by T_2, T_3, T_8, T_9, and T_{10}. Hypothetically, let the serial order for the first subtree be (T_4, T_1, T_5, T_2, T_6, T_3, T_7); note that this order must include T_1, T_2, and T_3 in that order. Also, let the serial order for the second subtree be (T_8, T_2, T_9, T_{10}, T_3). As must be the case, the transactions T_2 and T_3, which locked the root, appear in this sequence in the order in which they locked the root.

The constraints imposed on the serial order of these transactions are as shown in the following figure:



Solid lines represent constraints due to the order at the first child of the root, while dashed lines represent the order at the second child. (T_4, T_8, T_1, T_5, T_2, T_9, T_6, T_{10}, T_3, T_7) is one of the many topological sorts of this graph.

Concurrency Control by Timestamps

Next, we shall consider two methods other than locking that are used in some systems to assure serializability of transactions:

1. *Timestamping* (*timestamp ordering* — TO): Assign a "timestamp" to each transaction. Record the timestamps of the transactions that last read and write each database element, and compare these values with the transactions' timestamps, to assure that the serial schedule according to the transactions' timestamps is equivalent to the actual schedule of the transactions.
2. *Validation*: Examine timestamps of the transaction and the database elements when a transaction is about to commit; this process is called "validation" of the transaction. The serial schedule that orders transactions according to their validation time must be equivalent to the actual schedule.

Both these approaches are *optimistic*, in the sense that they assume that no unserializable behavior will occur and only fix things up when a violation is apparent. In contrast, all locking methods assume that things will go wrong unless transactions are prevented in advance from engaging in nonserializable behavior. The optimistic approaches differ from locking in that the only remedy when something does go wrong is to abort and restart a transaction that tries to engage in unserializable behavior. In contrast, locking schedulers delay transactions, but do not abort them. (That is not to say that systems using a locking scheduler will never abort a transaction; sometimes they do, for instance, to fix deadlocks. However, a locking scheduler never uses a transaction abort simply as a response to a lock request that it cannot grant.) Generally, optimistic schedulers are better than locking when many of the transactions are read-only, since those transactions can never, by themselves, cause unserializable behavior.

Timestamps

To use timestamping as a concurrency-control method, the scheduler needs to assign to each transaction T a unique number, its *timestamp* $TS(T)$. Timestamps must be issued in ascending order, at the time that a transaction first notifies the scheduler that it is beginning. Two approaches to generating timestamps are:

a) We can use the system clock as the timestamp, provided the scheduler does not operate so fast that it could assign timestamps to two transactions on one tick of the clock.

b) The scheduler can maintain a counter. Each time a transaction starts, the counter is incremented by 1, and the new value becomes the timestamp of the transaction. In this approach, timestamps have nothing to do with "time," but they have the important property that we need for any timestamp-generating system: a transaction that starts later has a higher timestamp than a transaction that starts earlier.

Whatever method of generating timestamps is used, the scheduler must maintain a table of currently active transactions and their timestamps.

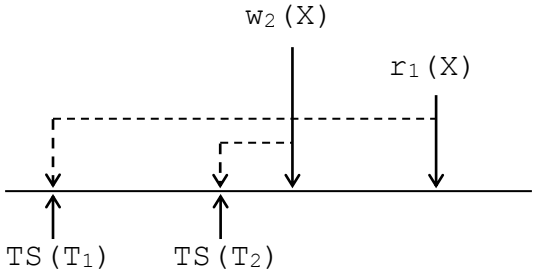
To use timestamps as a concurrency-control method, we need to associate with each database element X two timestamps and perhaps an additional bit:

1. RT(X): the *read time* of X, which is the highest timestamp of a transaction that has read X.
2. WT(X): the *write time* of X, which is the highest timestamp of a transaction that has written X.
3. C(X): the *commit bit* for X, which is true if and only if the most recent transaction to write X has already committed. This bit is not essential, its purpose is to avoid a situation where one transaction T reads data written by another transaction U, and U then aborts. This problem, where T makes a "dirty read" of uncommitted data, certainly can cause the database state to become inconsistent, and any scheduler needs a mechanism to prevent dirty reads. (Although commercial systems generally give the user an option to allow dirty reads, as suggested by the SQL isolation level "read uncommitted" — see later.)

Physically Unrealizable Behaviors

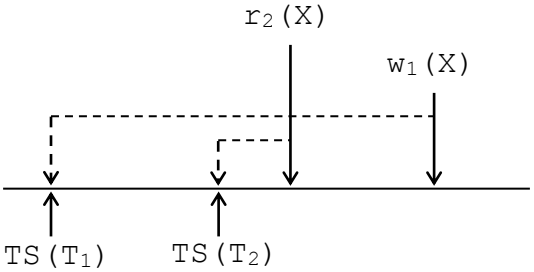
In order to understand the architecture and rules of a timestamp scheduler, we need to remember that the scheduler assumes the timestamp order of transactions is also the serial order in which they must appear to execute. Thus, the job of the scheduler, in addition to assigning timestamps and updating RT, WT, and C for the database elements, is to check that whenever a read or write occurs, what happens in real time could have happened if each transaction had executed instantaneously at the moment of its timestamp. If not, we say the behavior is *physically unrealizable*. There are two kinds of problems that can occur:

1. *Read too late*: Transaction T_1 tries to read database element X, but the write time of X indicates that the current value of X was written after T_1 theoretically executed; that is, $TS(T_1) < WT(X)$. The following figure illustrates the problem:



   The horizontal axis represents the real time at which events occur. Dashed lines link the actual events to the times at which they theoretically occur — the timestamp of the transaction that performs the event. Thus, we see a transaction T_2 that started after transaction T_1, but wrote a value for X before T_1 reads X. T_1 should not be able to read the value written by T_2, because theoretically, T_2 executed after T_1 did. However, T_1 has no choice, because T_2's value of X is the one that T_1 now sees. The solution is to abort T_1 when the problem is encountered.

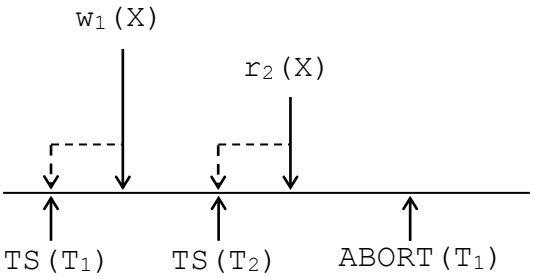
2. *Write too late*: Transaction T_1 tries to write database element X. However, the read time of X indicates that some other transaction should have read the value written by T_1, but read some other value instead. That is, $TS(T_1) < RT(X)$. The problem is shown in the following figure:



Here, we see a transaction T_2 that started after T_1, but read X before T_1 got a chance to write X. When T_1 tries to write X, we find $RT(X) > TS(T_1)$, meaning that X has already been read by a transaction T_2 that theoretically executed later than T_1.

Problems With Dirty Data

There is a class of problems that the commit bit is designed to solve. One of these problems, a "dirty read," is suggested in the following figure:

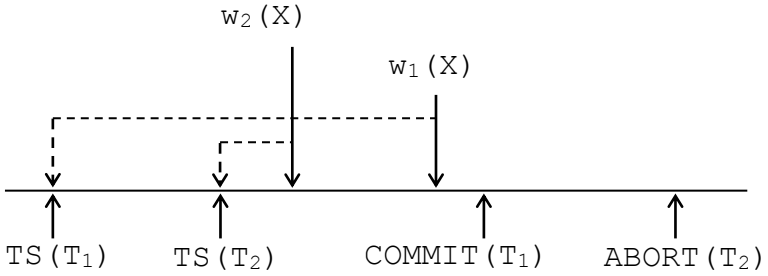


Here, transaction T_2 reads X, and X was last written by T_1. The timestamp of T_1 is less than that of T_2, and the read by T_2 occurs after the write by T_1 in real time, so the event seems to be physically realizable. However, it is possible that after T_2 reads the value of X written by T_1, transaction T_1 will abort; perhaps T_1 encounters an error condition in its own data, such as a division by 0, or the scheduler forces T_1 to abort because it tries to do something physically unrealizable. Thus, although there is nothing physically unrealizable about T_2 reading X, it is better to delay T_2's read until T_1 commits or aborts, or else our schedule may become non-conflict-serializable. We can tell that T_1 is not committed because the commit bit $C(X)$ will be false.

We can solve the dirty read problem also without the commit bit: Whenever a transaction T is aborted, we check whether there are any other transactions that read one or more database elements written by T. If so, they too must be aborted. These aborts may imply yet other aborts, and so on. Such situations are called *cascading rollbacks*. This solution, however, leads to a lower degree of concurrency than the commit bit and delays. Moreover, it may result in a *nonrecoverable schedule*, which occurs if one of the transactions to be aborted has already been committed.

A very simple, though drastic, solution to the dirty read problem is to abort each transaction that is about to read dirty data. Finally, multiversion timestamping also gives a remedy for this problem (see later).

A second potential problem is suggested by the following figure:



Here, T_2, a transaction with a later timestamp than T_1, has written X first. When T_1 tries to write, the appropriate action is to do nothing. Evidently, no other transaction T_3 that should have read T_1's value of X got T_2's value instead, because if T_3 tried to read X, it would have aborted because of a too-late read. Future reads of X will want T_2's value or a later value of X, not T_1's value. This idea, that writes can be skipped when a write with a later write time is already in place, is called the *Thomas' write rule*.

There is a potential problem with the Thomas' write rule, however. If T_2 later aborts, as is suggested in the figure, then its value of X should be removed and the previous value and write time restored. Since T_1 is committed, it would seem that the value of X should be the one written by T_1 for future reading. However, we already skipped the write by T_1, and it is too late to repair the damage.

This is how this problem can be managed: When transaction T_1 writes X, and the scheduler sees that the write time of X is greater than T_1's timestamp (i.e., $TS(T_1) < WT(X)$) and that the transaction having written X (T_2 in the figure) is not yet committed (i.e., $C(X)$ is false), then T_1 is delayed until $C(X)$ becomes true, either because T_2 commits or because T_2 aborts. If T_2 commits, then T_1's write will be skipped, but if T_2 aborts, then T_1's write will proceed.

Another solution is to simply abort T_1, instead of delaying it, if the conditions described above are true. This solution apparently leads to a lower degree of concurrency than the commit bit and delays, and if dirty reads are also handled by rollbacks, then this abort increases the risk of cascading rollbacks and nonrecoverable schedules. And finally, multiversion timestamping can be a solution also in this case.

As you can see, when using basic timestamp ordering (with no commit bit and delays), deadlocks cannot occur, but cascading rollbacks and nonrecoverable schedules can.

The Rules for Timestamp-Based Scheduling

We can now summarize the rules that a scheduler using timestamps must follow to make sure that only conflict-serializable schedules may occur. Here, we consider a scheduler using the commit bit. The scheduler, in response to a read or write request from a transaction T, has the choice of:

a) granting the request,
b) aborting T (if T would "violate physical reality") and restarting T with a new timestamp (abort followed by restart is often called *rollback*), or
c) delaying T and later deciding what to do (if the request is a read, and the read might be dirty, or if the request is a write, and Thomas' write rule might be applied).

The rules are as follows:

1. Suppose the scheduler receives a request $r_T(X)$:
   a) If $TS(T) \geq WT(X)$, the read is physically realizable.
      i) If $C(X)$ is true or $TS(T) = WT(X)$, grant the request. If $TS(T) > RT(X)$, set $RT(X) := TS(T)$, otherwise do not change $RT(X)$.

ii) If C(X) is false and TS(T) > WT(X), delay T until C(X) becomes true (i.e., the latest transaction having written X commits or aborts).

b) If TS(T) < WT(X), the read is physically unrealizable. Rollback T, that is, abort T and restart it with a new, larger timestamp.

2. Suppose the scheduler receives a request $w_T(X)$:

a) If TS(T) ≥ RT(X) and TS(T) ≥ WT(X), the write is physically realizable and must be performed:

i) write the new value for X,
ii) set WT(X) := TS(T), and
iii) set C(X) := false.

b) If TS(T) ≥ RT(X), but TS(T) < WT(X), then the write is physically realizable, but there is already a later value in X.

i) If C(X) is true, then the previous writer of X is committed, and we simply ignore the write by T; we allow T to proceed and make no change to the database.
ii) However, if C(X) is false, then we must delay T as in point 1. a) ii).

c) If TS(T) < RT(X), then the write is physically unrealizable, and T must be rolled back.

3. Suppose the scheduler receives a request to commit T. It must find (using a list maintained by the scheduler) all the database elements X last written by T (WT(X) = TS(T)), and set C(X) := true. If, according to points 1. a) ii) and 2. b) ii), any transactions are waiting for X to be committed (found from another scheduler-maintained list), these transactions are allowed to proceed with another attempt to execute their delayed read or write actions.

4. Suppose the scheduler receives a request to abort T or decides to rollback T as in 1. b) or 2. c). It must undo all writes of T that involve a database element X such that WT(X) = TS(T). This means that the old values of X and WT(X) are restored that belong to the biggest write time, and C(X) is set to true if the transaction with a timestamp equal to this write time has committed. Additionally, the reads of T also have to be "undone", i.e., for all X such that RT(X) = TS(T), the biggest old value of RT(X) must also be restored. Then, any transaction that was waiting for an element X that T wrote (1. a) ii) and 2. b) ii)) must repeat its attempt to read or write, and see whether the action is now legal after T's writes are cancelled.

Example. The following figure shows a schedule of three transactions, T_1, T_2, and T_3 that access three database elements, A, B, and C:

T_1	T_2	T_3	A	B	C
200	150	175	RT = 0 WT = 0 C = true	RT = 0 WT = 0 C = true	RT = 0 WT = 0 C = true
$r_1(B)$;				RT = 200	
	$r_2(A)$;		RT = 150		
		$r_3(C)$;			RT = 175
$w_1(B)$;				WT = 200 C = false	
$w_1(A)$;			WT = 200 C = false		
	$w_2(C)$;				
	abort		RT = 0		
commit			C = true	C = true	
		$w_3(A)$;			

The real time at which events occur increases down the page, as usual. We have also indicated the timestamps of the transactions and the read and write times of the elements. At the beginning, each of the database elements has both a read and write time of 0. The timestamps of the transactions are acquired when they notify the scheduler that they are beginning. Notice that even though T_1 executes the first data access, it does not have the least timestamp. Presumably T_2 was the first to notify the scheduler of its start, and T_3 did so next, with T_1 last to start.

In the first action, T_1 reads B. Since the write time of B is less than the timestamp of T_1, this read is physically realizable and allowed to happen. The read time of B is set to 200, the timestamp of T_1. The second and third read actions similarly are legal and result in the read time of each database element being set to the timestamp of the transaction that read it.

At the fourth step, T_1 writes B. Since the read time of B is not bigger than the timestamp of T_1, the write is physically realizable. Since the write time of B is no larger than the timestamp of T_1, we must actually perform the write. When we do, the write time of B is raised to 200, the timestamp of the writing transaction T_1. Then, we do the same with A.

Next, T_2 tries to write C. However, C was already read by transaction T_3, which theoretically executed at time 175, while T_2 would have written its value at time 150. Thus, T_2 is trying to do something that's physically unrealizable, so T_2 must be rolled back.

The last step is the write of A by T_3. Since the read time of A, 150, is less than the timestamp of T_3, 175, the write is legal. However, there is already a later value of A stored in that database element, namely the value written by T_1, theoretically at time 200. Thus, T_3 is not rolled back, but neither does it write its value. (We suppose that T_1 has committed by this time.)

Multiversion Timestamping

An important variation of timestamping (called *multiversion timestamping*, *multiversion timestamp ordering* — MVTO, or *multiversion concurrency control* — MVCC) maintains old versions of database elements in addition to the current version that is stored in the database itself. The purpose is to allow reads $r_T(X)$ that otherwise would cause transaction T to abort (because the current version of X was written in T's future) to proceed by reading the version of X that is appropriate for a transaction with T's timestamp. The method is especially useful if database elements are disk blocks or pages, since then all that must be done is for the buffer manager to keep in memory certain blocks that might be useful for some currently active transaction.

Example. Consider the set of transactions accessing database element A shown in the following figure:

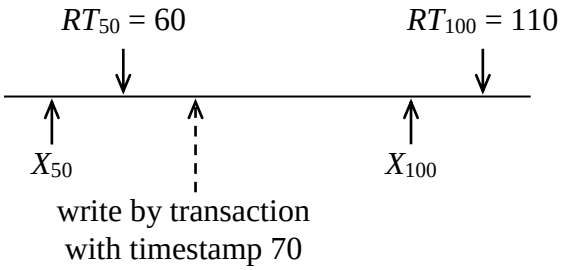
T_1	T_2	T_3	T_4	A
150	200	175	225	RT = 0 WT = 0
$r_1(A)$;				RT = 150
$w_1(A)$;				WT = 150
	$r_2(A)$;			RT = 200
	$w_2(A)$;			WT = 200
		$r_3(A)$;		
		abort		
			$r_4(A)$;	RT = 225

These transactions are operating under an ordinary timestamp-based scheduler, and when T_3 tries to read A, it finds WT(A) to be greater than its own timestamp and must abort. However, there is an old value of A written by T_1 and overwritten by T_2 that would have been suitable for T_3 to read; this version of A had a

write time of 150, which is less than T_3's timestamp of 175. If this old value of A were available, T_3 could be allowed to read it, even though it is not the "current" value of A.

A multiversion-timestamp scheduler differs from the scheduler described above in the following ways:

1. When a new write $w_T(X)$ occurs, if it is legal, then a new version of database element X is created. Its write time is TS(T), and we shall refer to it as X_t, where $t = TS(T)$.
2. When a read $r_T(X)$ occurs, the scheduler finds the version X_t of X such that $t \leq TS(T)$, but there is no other version $X_{t'}$ with $t < t' \leq TS(T)$. That is, the version of X written immediately before T theoretically executed is the version that T reads.
3. Write times are associated with versions of an element, and they never change.
4. Read times are also associated with versions. They are used to make it possible to accept certain writes, namely one whose time is greater or equal to the read time of the previous version. If we maintained only the read time of the latest version, then we would have to reject such writes. The following figure suggests the problem:



   X has versions X_{50} and X_{100}. X_{50} was read at time 60, and a new write by a transaction T with timestamp 70 occurs. This write is legal because $RT_{50} \leq TS(T)$. If we knew only the read time of the latest version (110), we couldn't determine whether this write is legal, so T would have to be aborted.

5. When a version X_t has a write time t such that no active transaction has a timestamp less than t, then we may delete any version of X previous to X_t.

Example. Let us reconsider the actions of the previous example if multiversion timestamping is used:

T_1	T_2	T_3	T_4	A_0	A_{150}	A_{200}
150	200	175	225	RT = 0		
$r_1(A)$;				read, RT = 150		
$w_1(A)$;					create, RT = 150	
	$r_2(A)$;				read, RT = 200	
	$w_2(A)$;					create, RT = 200
		$r_3(A)$;			read	
			$r_4(A)$;			read, RT = 225

First, there are three versions of A: A_0, which exists before these transactions start, A_{150}, written by T_1, and A_{200}, written by T_2. The figure shows the sequence of events, when the versions are created, and when they are read. Notice in particular that T_3 does not have to abort, because it can read an earlier version of A.

As we can see, multiversion timestamping eliminates too-late reads. What about dirty reads and the problem of Thomas' write rule? Dirty reads might occur in this case too, but now we can take measures against them

other than delaying or aborting the transaction. When a transaction T_1 wants to read a database element X, we look up the latest version of X that was created by T_1 itself or by a transaction T_2 that was *committed* when T_1 started. This way, dirty reads will never occur, and no transactions have to be delayed. Moreover, too-late writes may not occur either, because a transaction "writing too late" could not have been committed when the reading transaction started; therefore, the reading transaction does not need the value "written too late" anyway. This technique (also used by Oracle Database) is called *snapshot isolation*, the main drawback of which is that it does not guarantee serializability.

Thomas' write rule may not be applied in case of multiversion timestamping (at least in its original form); the "new" version of a database element is always created, even if it is older than the latest version.

Timestamps Versus Locking

Generally, timestamping is superior in situations where either most transactions are read-only, or it is rare that concurrent transactions will try to read and write the same element. In high-conflict situations, locking performs better. The argument for this rule-of-thumb is:

- Locking will frequently delay transactions as they wait for locks, and deadlocks can occur when some transactions have been waiting for a long time, in which case one of them needs to be rolled back.
- But if concurrent transactions frequently read and write elements in common, then rollbacks will be frequent in a timestamp scheduler, introducing even more delay than a locking system.

There is an interesting compromise used in several commercial systems. The scheduler divides the transactions into read-only transactions and read/write transactions. Read/write transactions are executed using two-phase locking, to keep all transactions from accessing the elements they lock. Read-only transactions are executed using multiversion timestamping. As the read/write transactions create new versions of a database element, those versions are managed as discussed above. A read-only transaction is allowed to read whatever version of a database element is appropriate for its timestamp. A read-only transaction thus never has to abort, and will only rarely be delayed.

Concurrency Control by Validation

Validation (*Kung–Robinson model*) is another type of optimistic concurrency control, where we allow transactions to access data without locks, and at the appropriate time, we check that the transaction has behaved in a serializable manner. Validation differs from timestamping principally in that the scheduler maintains a record of what active transactions are doing, rather than keeping read and write times for all database elements. Just before a transaction starts to write values of database elements, it goes through a "validation phase," where the sets of elements it has read and will write are compared with the write sets of other active transactions. Should there be a risk of physically unrealizable behavior, the transaction is rolled back.

Architecture of a Validation-Based Scheduler

When validation is used as the concurrency-control mechanism, the scheduler must be told for each transaction T the sets of database elements T reads and writes: the *read set*, `RS(T)`, and the *write set*, `WS(T)`, respectively. Transactions are executed in three phases:

1. *Read*. In the first phase, the transaction reads from the database all required elements in its read set. The transaction also computes in its local address space all the results it is going to write, thus creating its write set.

2. *Validate*. In the second phase, the scheduler validates the transaction by comparing its read and write sets with those of other transactions. We shall describe the validation process later. If validation fails, then the transaction is rolled back; otherwise it proceeds to the third phase.
3. *Write*. In the third phase, the transaction writes to the database its values for the elements in its write set.

Intuitively, we may think of each transaction that successfully validates as executing at the moment that it validates. Thus, the validation-based scheduler has an assumed serial order of the transactions to work with, and it bases its decision to validate or not on whether the transactions' behaviors are consistent with this serial order. To support the decision whether to validate a transaction, the scheduler maintains three sets:

1. *START*: the set of transactions that have started but not yet completed validation. For each transaction T in this set, the scheduler maintains START(T), the time at which T started.
2. *VAL*: the set of transactions that have been validated but not yet finished the writing of phase 3. For each transaction T in this set, the scheduler maintains both START(T) and VAL(T), the time at which T validated. Note that VAL(T) is also the time at which T is imagined to execute in the hypothetical serial order of execution.
3. *FIN*: the set of transactions that have completed phase 3. For these transactions T, the scheduler records START(T), VAL(T), and FIN(T), the time at which T finished. In principle, this set grows, but as we shall see, we do not have to remember transaction T if FIN(T) < START(U) for any active transaction U (i.e., for any $U \in START \cup VAL$). The scheduler may thus periodically purge the *FIN* set to keep its size from growing beyond bounds.

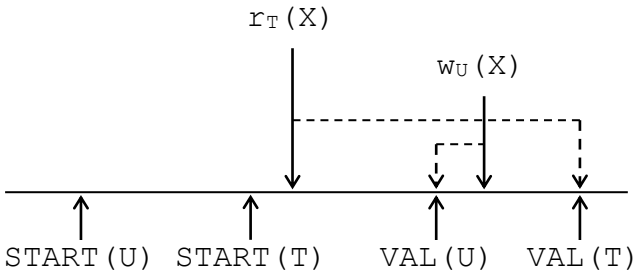
The Validation Rules

The information described above is enough for the scheduler to detect any potential violation of the assumed serial order of the transactions — the order in which the transactions validate. To understand the rules, let us first consider what can be wrong when we try to validate a transaction T.

1. Suppose there is a transaction U such that:

   a) $U \in VAL \cup FIN$, that is, U has validated.
   b) FIN(U) > START(T), that is, U did not finish before T started. (Note that if $U \in VAL$, then U has not yet finished when T validates. In that case, FIN(U) is technically undefined. However, we know it must be larger than START(T) in this case.)
   c) $RS(T) \cap WS(U) \neq \emptyset$, in particular, let it contain database element X.

   Then it is possible that U wrote X after T read X ("read too early"). In fact, U may not even have written X yet. A situation where U wrote X, but not in time is shown in the following figure:



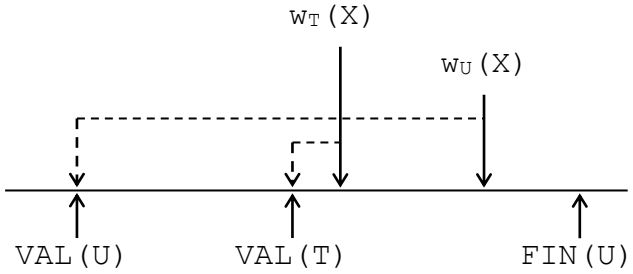
   To interpret the figure, note that the dashed lines connect the events in real time with the time at which they would have occurred had transactions been executed at the moment they validated. Since we don't

know whether or not T got to read U's value, we must rollback T to avoid a risk that the actions of T and U will not be consistent with the assumed serial order.

2. Suppose there is a transaction U such that:

   a) $U \in VAL$, i.e., U has successfully validated.
   b) $FIN(U) > VAL(T)$, that is, U did not finish before T entered its validation phase. (This condition is always true actually, because U surely has not finished yet.)
   c) $WS(T) \cap WS(U) \neq \emptyset$, in particular, let X be in both write sets.

   Then the potential problem is as shown in the following figure:

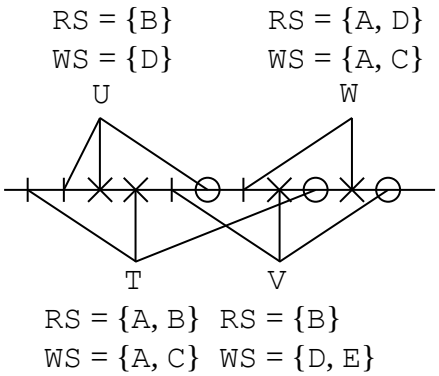


   T and U must both write values of X, and if we let T validate, it is possible that it will write X before U does ("write too early"). Since we cannot be sure, we rollback T to make sure it does not violate the assumed serial order in which it follows U.

The two problems described above are the only situations in which a read or write by T could be physically unrealizable. In the first case, if U finished before T started, then surely T would read the value of X that either U or some later transaction wrote. In the second case, if U finished before T validated, then surely U wrote X before T did. We may thus summarize these observations with the following rule for validating a transaction T:

- Check that $RS(T) \cap WS(U) = \emptyset$ for any previously validated U that did not finish before T started, i.e., if $U \in VAL \cup FIN$ and $FIN(U) > START(T)$.
- Check that $WS(T) \cap WS(U) = \emptyset$ for any previously validated U that did not finish before T validated, i.e., if $U \in VAL$ and $FIN(U) > VAL(T)$.

Example.



The figure shows a time line during which four transactions T, U, V, and W attempt to execute and validate. For each transaction, I denotes its start time, X its validation time, and O its finishing time. The read and write sets for each transaction are indicated on the diagram. T starts first, although U is the first to validate.

1. Validation of U: When U validates, there are no other validated transactions, so there is nothing to check. U validates successfully and writes a value for database element D.

2. Validation of T: When T validates, U is validated but not finished. Thus, we must check that neither the read nor the write set of T has anything in common with WS(U) = {D}. Since RS(T) = {A, B}, and WS(T) = {A, C}, both checks are successful, and T validates.

3. Validation of V: When V validates, U is validated and finished, and T is validated but not finished. Also, V started before U finished. Thus, we must compare both RS(V) and WS(V) against WS(T), but only RS(V) needs to be compared against WS(U). We find:

   - $RS(V) \cap WS(T) = \{B\} \cap \{A, C\} = \emptyset$;
   - $WS(V) \cap WS(T) = \{D, E\} \cap \{A, C\} = \emptyset$;
   - $RS(V) \cap WS(U) = \{B\} \cap \{D\} = \emptyset$.

   Thus, V also validates successfully.

4. Validation of W: When W validates, we find that U finished before W started, so no comparison between W and U is performed. T is finished before W validates but did not finish before W started, so we compare only RS(W) with WS(T). V is validated but not finished, so we need to compare both RS(W) and WS(W) with WS(V). These tests are:

   - $RS(W) \cap WS(T) = \{A, D\} \cap \{A, C\} = \{A\}$;
   - $RS(W) \cap WS(V) = \{A, D\} \cap \{D, E\} = \{D\}$;
   - $WS(W) \cap WS(V) = \{A, C\} \cap \{D, E\} = \emptyset$.

   Since the intersections are not all empty, W is not validated. Rather, W is rolled back and does not write values for A or C.

You may have been concerned with a tacit notion that validation takes place in a moment, or indivisible instant of time. For example, we imagine that we can decide whether a transaction U has already validated before we start to validate transaction T. Could U perhaps finish validating while we are validating T?

If we are running on a uniprocessor system, and there is only one scheduler process, we can indeed think of validation and other actions of the scheduler as taking place in an instant of time. The reason is that if the scheduler is validating T, then it cannot also be validating U, so all during the validation of T, the validation status of U cannot change.

If we are running on a multiprocessor, and there are several scheduler processes, then it might be that one is validating T while the other is validating U. If so, then we need to rely on whatever synchronization mechanism the multiprocessor system provides to make validation an atomic action.

Comparison of the Three Concurrency-Control Mechanisms

The three approaches to serializability that we have considered — locking, timestamping, and validation — each have their advantages. First, they can be compared for their storage utilization:

- *Locking*: Space in the lock table is proportional to the number of database elements locked.
- *Timestamping*: In a naive implementation, space is needed for read and write times with every database element, whether or not it is currently accessed. However, a more careful implementation will treat all timestamps that are prior to the earliest active transaction as "minus infinity" and not record them. In that case, we can store read and write times in a table analogous to a lock table, in which only those database elements that have been accessed recently are mentioned at all.
- *Validation*: Space is used for timestamps and read/write sets for each currently active transaction, plus a few more transactions that finished after some currently active transaction began.

Thus, the amounts of space used by each approach is approximately proportional to the sum over all active transactions of the number of database elements the transaction accesses. Timestamping and validation may use slightly more space because they keep track of certain accesses by recently committed transactions that a lock table would not record. A potential problem with validation is that the write set for a transaction must be known before the writes occur (but after the transaction's local computation has been completed).

We can also compare the methods for their effect on the ability of transactions to complete without delay. The performance of the three methods depends on whether interaction among transactions (the likelihood that a transaction will access an element that is also being accessed by a concurrent transaction) is high or low:

- Locking delays transactions but avoids rollbacks, even when interaction is high. Timestamping and validation do not delay transactions but can cause them to rollback, which is a more serious form of delay and also wastes resources.
- If interference is low, then neither timestamping nor validation will cause many rollbacks, and may be preferable to locking because they generally have lower overhead than a locking scheduler.
- When a rollback is necessary, timestamping catches some problems earlier than validation, which always lets a transaction do all its internal work before considering whether the transaction must rollback.

Concurrency Control in Oracle Database

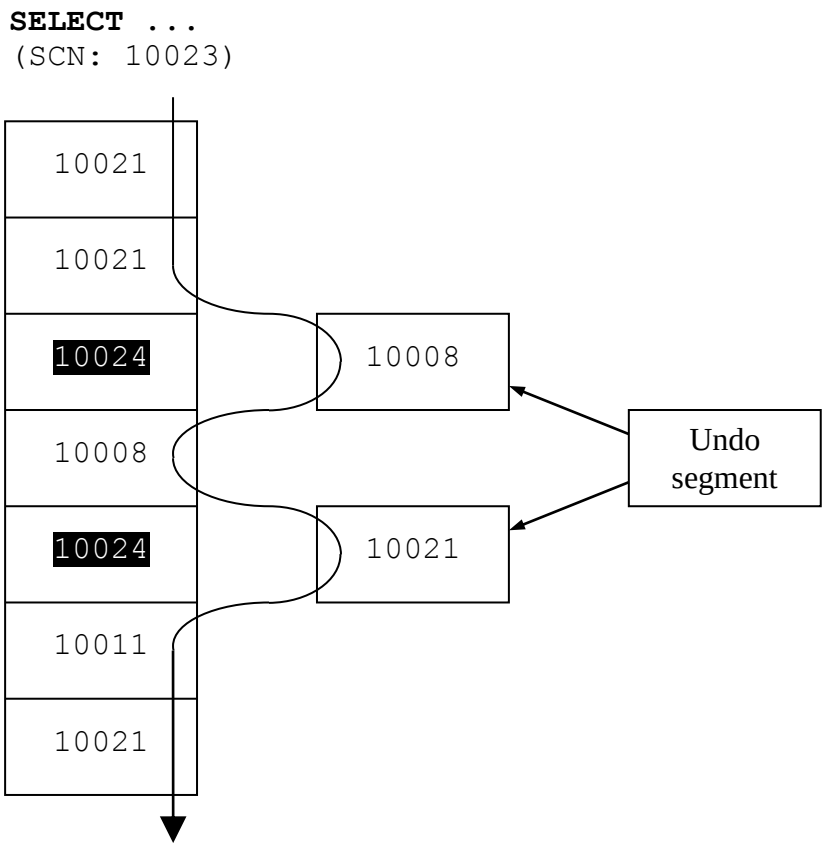
The following information comes from Oracle Database Concepts — Data Concurrency and Consistency.

Oracle Database uses a combination of two-phase locking and snapshot isolation for concurrency control. At user level, there are two lock units: the table and the row. Locks are automatically acquired and released by the scheduler, but users (applications) may also request locks.

Levels of Read Consistency

Oracle Database always enforces *statement-level read consistency*, which guarantees that data returned by a single query is committed and consistent for a single point in time (by default, the time at which the statement was opened). Thus, a query never reads dirty (uncommitted) data or data that commits while the query is in progress. Oracle Database can also provide read consistency to all queries in a transaction, known as *transaction-level read consistency*. We can achieve this by running the transaction in serializable or read-only mode (see later). In this case, each statement in a transaction sees data from the same point in time, which is the time at which the transaction began (except for data changed by the transaction itself).

To manage the multiversion read consistency model, the database must create a read-consistent set of data when a table is simultaneously queried and updated. Oracle Database achieves this goal through undo data. Whenever a user modifies data, Oracle Database creates undo entries, which it writes to undo (or rollback) segments. The undo segments contain the old values of data that have been changed by uncommitted or recently committed transactions. Thus, multiple versions of the same data, all at different points in time, can exist in the database. The database can use snapshots of data at different points in time to provide read-consistent views of the data and enable nonblocking queries (see later). As a query or transaction enters the execution phase, the database determines the *system change number* (SCN) recorded at the time the query or transaction began executing. An SCN can be considered as a timestamp associated with blocks as database elements. As the query reads through the data blocks, the database compares each block's SCN ("time" of last write) with that of the query and reads only those committed blocks whose SCN is not greater than the SCN of the query. In case of blocks with a greater SCN, it copies current data blocks to a new buffer and applies undo data to reconstruct previous versions of the blocks — versions that were created by a committed transaction and whose SCN is the greatest but less than the SCN of the query. These reconstructed data blocks are called *consistent read clones*. The following figure illustrates this process:



In some cases, the required old version of a block can no longer be reconstructed based on the undo data. When automatic undo management is enabled, there is always a current *undo retention period*, which is the minimum amount of time that Oracle Database attempts to retain old undo information before overwriting it. Old (committed) undo information that is older than the current undo retention period is said to be *expired* and its space is available to be overwritten by new transactions. Old undo information with an age that is less than the current undo retention period is said to be *unexpired* and is possibly retained for consistent read and Oracle Flashback operations (to restore a table to its state as of a previous point in time).

If the undo tablespace is configured with the AUTOEXTEND option, the database dynamically tunes the undo retention period to be somewhat longer than the longest-running active query on the system. When space is low, instead of overwriting unexpired undo information, the tablespace auto-extends. If the MAXSIZE clause is specified for an auto-extending undo tablespace, when the maximum size is reached, the database may begin to overwrite unexpired undo information. (The UNDOTBS1 tablespace that is automatically created by DBCA is auto-extending.)

If the undo tablespace is fixed size, the database dynamically tunes the undo retention period for the best possible retention for that tablespace size and the current system load. This best possible retention time is typically significantly greater than the duration of the longest-running active query. If you choose an undo tablespace size that is too small, then long-running queries could fail with a “snapshot too old” error, which means that there was insufficient undo data for read consistency.

To guarantee the success of long-running queries or Oracle Flashback operations, you can enable *retention guarantee*. If retention guarantee is enabled, the specified minimum undo retention is guaranteed; the database never overwrites unexpired undo data, even if it means that transactions fail due to lack of space in the undo tablespace. If retention guarantee is not enabled, the database can overwrite unexpired undo when space is low, thus lowering the undo retention for the system.

Transaction Isolation Levels

The SQL92 ANSI/ISO standard defines four levels of isolation in terms of the phenomena that a transaction running at a particular isolation level is permitted to experience. The preventable phenomena are:

- *dirty reads:* a transaction reads data that has been written by another transaction that has not been committed yet;
- *nonrepeatable (fuzzy) reads:* a transaction rereads data it has previously read and finds that another committed transaction has modified or deleted the data;
- *phantom reads:* a transaction reruns a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

The four transaction isolation levels are the following:

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
read uncommitted	possible	possible	possible
read committed	not possible	possible	possible
repeatable read	not possible	not possible	possible
serializable	not possible	not possible	not possible

Oracle Database provides the read committed and serializable transaction isolation levels, as well as a *read-only* mode, which is not part of the standard.

- *Read committed:* This is the default transaction isolation level. Every query executed by a transaction sees only data committed before the query — not the transaction — began. Dirty reads never occur. However, because the database does not prevent other transactions from modifying data read by a query, other transactions may change data between query executions. Thus, a transaction that runs the same query twice may experience fuzzy reads and phantoms. This level of isolation is appropriate for database environments in which few transactions are likely to conflict.
- *Serializable:* A transaction sees only changes committed at the time the transaction — not the query — began and changes made by the transaction itself using `INSERT`, `UPDATE`, and `DELETE` statements. Serializable transactions do not experience dirty reads, fuzzy reads, or phantom reads. Serializable isolation is suitable for environments with large databases and short transactions that update only a few rows, where the chance that two concurrent transactions will modify the same rows is relatively low, and where relatively long-running transactions are primarily read-only. Oracle Database permits a serializable transaction to modify a row only if changes to the row made by other transactions were already committed when the serializable transaction began. The database generates an error ("Cannot serialize access for this transaction") when a serializable transaction tries to update or delete data changed by a different transaction that committed after the serializable transaction began. **Remember that, despite its name, serializable isolation level actually uses snapshot isolation and does not guarantee serializability!**
- *Read-only:* The read-only isolation level is similar to the serializable isolation level, but read-only transactions do not permit data to be modified in the transaction unless the user is SYS. Thus, read-only transactions are not susceptible to the error described above. Read-only transactions are useful for generating reports in which the contents must be consistent with respect to the time when the transaction began.

You can set the isolation level of a transaction by using one of these statements at the beginning of the transaction:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET TRANSACTION READ ONLY;
```

The Locking System

Both read committed and serializable transactions use row-level locking, and both will wait if they try to change a row updated by an uncommitted concurrent transaction. The second transaction that tries to update a given row waits for the other transaction to commit or abort and release its lock. If that other transaction rolls back, the waiting transaction, regardless of its isolation mode, can proceed to change the previously locked row as if the other transaction had not existed. However, if the other blocking transaction commits and releases its locks, only a read committed transaction proceeds with its intended update. A serializable transaction fails with the "Cannot serialize access" error, because the other transaction has committed a change that was made since the serializable transaction began.

Oracle Database automatically obtains necessary locks when executing SQL statements. Users never need to lock any resource explicitly, although Oracle Database also enables users to lock data manually. Oracle Database automatically uses the lowest applicable level of restrictiveness to provide the highest degree of data concurrency, yet also provide fail-safe data integrity.

Usually, the database holds locks acquired by statements within a transaction for the duration of the transaction (two-phase locking). Oracle Database releases all locks acquired by the statements within a transaction when it commits or rolls back. Oracle Database also releases locks acquired after a savepoint when rolling back to the savepoint. However, only transactions not waiting for the previously locked resources can acquire locks on the now available resources. Waiting transactions continue to wait until after the original transaction commits or rolls back completely.

Types of Locks

Oracle Database locks are divided into the following categories:

- DML locks (data locks): protect data;
- DDL locks (data dictionary locks): protect the structure of schema objects (e.g., tables);
- system locks: protect internal database structures such as data files; managed entirely automatically.

DML locks exist in two levels: there are row-level locks (TX) and table-level locks (TM). Both of them are automatically acquired by DML statements. At row level, there is only one lock mode: exclusive. The combination of multiversion concurrency control and row-level locking means that users contend for data only when accessing the same rows. The following rules summarize the locking behavior of Oracle Database for readers and writers:

- A row is locked only when modified by a writer.
- A writer of a row blocks a concurrent writer of the same row.
- A reader never blocks a writer unless `SELECT … FOR UPDATE` is used, which is a special type of `SELECT` statement that *does* lock the row that it is reading.
- A writer never blocks a reader. When a row is being changed by a writer, the database uses undo data to provide readers with a consistent view of the row.

A query without a `FOR UPDATE` clause acquires no data locks. Therefore, other transactions can query and update a table being queried, including the specific rows being queried. Because queries lacking `FOR`

UPDATE clauses do not acquire any data locks to block other operations, such queries are often referred to in Oracle as *nonblocking queries*. On the other hand, a query does not have to wait for any data locks to be released; it can always proceed.

A transaction acquires a row lock (or TX lock) for each row modified by an INSERT, UPDATE, DELETE, MERGE, or SELECT … FOR UPDATE statement. If a transaction obtains a lock for a row, then the transaction also acquires a lock for the table containing the row. The table lock prevents conflicting DDL operations that would override data changes in the current transaction.

A table lock (or TM lock) is acquired by a transaction when a table is modified by an INSERT, UPDATE, DELETE, MERGE, SELECT … FOR UPDATE, or LOCK TABLE statement. A table lock can be held in any of the following modes: *row share* (RS) or *subshare* (SS), *row exclusive* (RX) or *subexclusive* (SX), *share* (S), *share row exclusive* (SRX) or *share-subexclusive* (SSX) and *exclusive* (X). The following table shows the table lock modes that statements acquire and lock modes with which they are compatible:

SQL statement	Lock mode	RS	RX	S	SRX	X
SELECT … FROM table	–	Y	Y	Y	Y	Y
INSERT INTO table	RX	Y	Y	N	N	N
UPDATE table	RX	Y*	Y*	N	N	N
MERGE INTO table	RX	Y	Y	N	N	N
DELETE FROM table	RX	Y*	Y*	N	N	N
SELECT … FROM table … FOR UPDATE	RX	Y*	Y*	N	N	N
LOCK TABLE table IN ROW SHARE MODE	RS	Y	Y	Y	Y	N
LOCK TABLE table IN ROW EXCLUSIVE MODE	RX	Y	Y	N	N	N
LOCK TABLE table IN SHARE MODE	S	Y	N	Y	N	N
LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE	SRX	Y	N	N	N	N
LOCK TABLE table IN EXCLUSIVE MODE	X	N	N	N	N	N

* Yes, if no conflicting row locks are held by another transaction. Otherwise, waits occur.

The description of each lock mode is the following:

- A row share table lock indicates that the transaction holding the lock on the table has locked rows in the table and intends to update them. An RS lock is the least restrictive mode of table lock, offering the highest degree of concurrency for a table.
- A row exclusive table lock generally indicates that the transaction holding the lock has updated table rows or issued SELECT … FOR UPDATE.
- A share table lock held by a transaction allows other transactions to query the table (without using SELECT … FOR UPDATE), but updates are allowed only if a single transaction holds the share table lock. Because multiple transactions may hold a share table lock concurrently, holding this lock is not sufficient to ensure that a transaction can modify the table.
- A share row exclusive table lock is more restrictive than a share table lock. Only one transaction at a time can acquire an SRX lock on a given table. An SRX lock held by a transaction allows other transactions to query the table (except for SELECT … FOR UPDATE) but not to update the table.
- An exclusive table lock is the most restrictive mode of table lock, allowing the transaction that holds the lock exclusive write access to the table. Only one transaction can obtain an X lock for a table.

A transaction containing a DML statement (INSERT, UPDATE, DELETE, MERGE, or SELECT … FOR UPDATE) acquires exclusive row locks on the rows modified by the statement. Therefore, other transactions cannot update or delete the locked rows until the locking transaction either commits or rolls back. In addition to these row locks, a transaction containing a DML statement that modifies data also requires at least a row exclusive table lock (RX) on the table that contains the affected rows. If the transaction already holds an S, SRX, or X table lock for the table, which are more restrictive than an RX lock, then the RX lock is not needed and is not acquired. If the containing transaction already holds only an RS lock, however, then Oracle Database automatically converts the RS lock to an RX lock.

A transaction that contains a DML statement does not require row locks on any rows selected by a subquery or an implicit query. A subquery or implicit query inside a DML statement is guaranteed to be consistent as of the start of the query and does not see the effects of the DML statement of which it forms a part.

A query in a transaction can see the changes made by previous DML statements in the same transaction, but not the uncommitted changes of other transactions.

Lock Conversion and Escalation

Oracle Database performs *lock conversion* (upgrading) as necessary. In lock conversion, the database automatically converts a table lock of lower restrictiveness to one of higher restrictiveness. For example, if a transaction holds an RS lock for a table, and a DML statement in the transaction intends to modify some rows in the table, the RS lock is automatically converted to an RX lock. Because row locks are acquired at the highest degree of restrictiveness (in exclusive mode), no lock conversion is required or performed.

Lock escalation occurs when numerous locks are held at one level of granularity (for example, rows) and a database raises the locks to a higher level of granularity (for example, table). If a user locks many rows in a table, then some database management systems automatically escalate the row locks to a single table. The number of locks decreases, but the restrictiveness of what is locked increases. Oracle Database never escalates locks. Lock escalation greatly increases the likelihood of deadlocks. Assume that a system is trying to escalate locks on behalf of transaction 1 but cannot because of the locks held by transaction 2. A deadlock is created if transaction 2 also requires lock escalation of the same data before it can proceed.