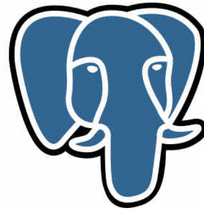
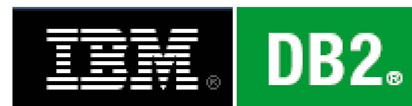
DBMS – II

Study Guide



SYBASE



Prepared By

Sharafat Ibn Mollah Mosharraf

CSE, DU

12th Batch (2005-2006)

Table of Contents

CHAPTER 9: OBJECT-BASED DATABASES **3**

CHAPTER 15: TRANSACTIONS **11**

CHAPTER 16: CONCURRENCY CONTROL **19**

Chapter 9
Object-Based Databases

Terms and Definitions

Object-Relational Data Model

The *object-relational data model* is an extension of the relational data model which provides a richer type system including complex data types and object orientation.

Object-Relational Database Systems

Object-relational database systems are database systems based on the object-relation model and provide a convenient migration path for users of relational databases who wish to use object-oriented features.

Persistent Programming Languages

Persistent Programming Languages refers to extensions of existing programming languages to add persistence and other database features using the native type system of the programming language.

Object-Oriented Database Systems

Object-oriented database systems refers to database systems that support an object-oriented type system and allow direct access to data from an object-oriented programming language using the native type system of the language.

SQL Complex Data Types Syntax with Comparison to OOP Language Syntax

Structured Types

SQL: User-Defined Types

ERD: Composite Attributes

OOP: Class

ERD	SQL	OOP (Java)

firstname
lastname
Name
	create type *Name* **as** (
	firstname **varchar**(20),
	lastname **varchar**(20)
	) **final**
		final class *Name* {
		String *firstname*;
		String *lastname*;
		}

street
Address
city
zipcode
	create type *Address* **as** (
	street **varchar**(20),
	city **varchar**(20),
	zipcode **varchar**(9)
	) **not final**
		class *Address* {
		String *street*;
		String *city*;
		String *zipcode*;
		}

The **final** specification for *Name* indicates that we cannot create subtypes for *name*, whereas the **not final** specification for *Address* indicates that we can create subtypes of *address*.

Creating a Table from These Types

1. By directly using these types in the table declaration:

```
create table customer as (
        name Name,
        address Address,
        dateOfBirth date
)
```

2. **By declaring a type consisting of these types and then declaring a table of this new type:**

```
create type CustomerType as (
        name Name,
        address Address,
        dateOfBirth date
) not final

create table customer of CustomerType
```

Directly Creating a Table Using Unnamed Row Types Instead of Creating Intermediate Types

```
create table customer_r (
        name row (firstname varchar(20),
                  lastname varchar(20)),
        address row (street varchar(20),
                     city varchar(20),
                     zipcode varchar(9)),
        dateOfBirth date
)
```

Methods

SQL	OOP (Java)

create type *CustomerType* **as (**
    name Name,
    address Address,
    dateOfBirth **date**
) not final
method *ageOnDate(onDate* **date**)
    returns interval year
 	
create instance method *ageOnDate(onDate* **date**)
    returns interval year
    for *CustomerType*
begin
    return *onDate* – **self**.*dateOfBirth*;
end
	class *CustomerType* {
	    Name *name*;
	    Address *address*;
	    Date *dateOfBirth*;
	
	    int *ageOnDate*(Date *onDate*) {
	        return *onDate* – **this**.*dateOfBirth*;
	    }
	}

The **for** clause indicates which type this method is for, while the keyword **instance** indicates that this method executes on an instance of the *CustomerType* type. The variable **self** refers to the instance of *CustomerType* on which the method is invoked.

Method Invocation

```
create table customer of CustomerType
```

```
select name.lastname, ageOnDate(current_date)
from customer
```

Constructor Functions

- Used to create values of structured types.
- A function with the same name as a structured type is a constructor function for the structured type.
- Every structured type has a default constructor – i.e. constructor with no arguments.
- Constructors can be overloaded.

```
create function Name (firstname varchar(20), lastname varchar(20))
returns Name
begin
        set self.firstname = firstname;
        set self.lastname = lastname;
end
```

Creating a Tuple Using Constructor

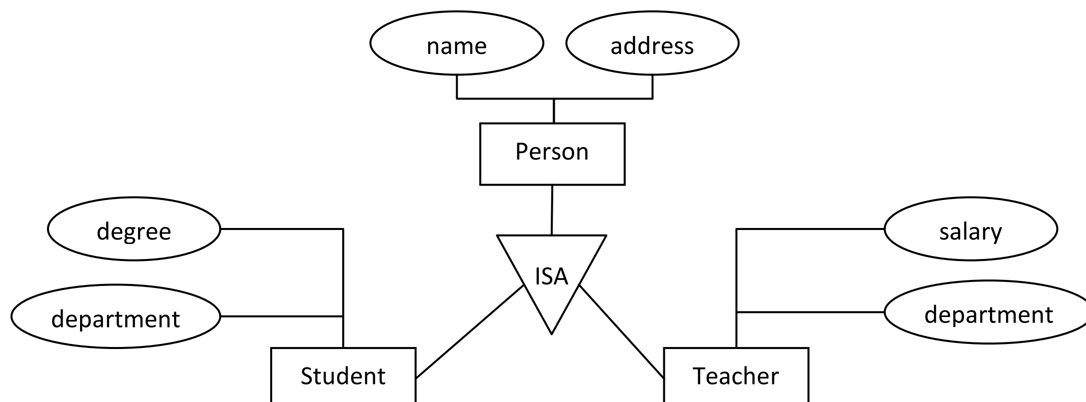
```
insert into customer
values (
        new Name('John', 'Smith'),
        new Address('20 Main Street', 'New York', '11001'),
        date '1960-8-22'
)
```

Creating a Tuple *without* Using Constructor

```
insert into customer
values (
         ('John', 'Smith'),
         ('20 Main Street', 'New York', '11001'),
        '1960-8-22'
)
```

Type Inheritance

ERD



SQL

```
create type Person (
    name varchar(20),
    address varchar(20)
)

create type Student under Person (
    degree varchar(20),
    department varchar(20)
)

create type Teacher under Person (
    salary integer,
    department varchar(20)
)
```

OOP (Java)

```
class Person {
    String name;
    String address;
}

class Student extends Person {
    String degree;
    String department;
}

class Teacher extends Person {
    int salary;
    String department;
}
```

Multiple Inheritance

```
create type TeachingAssistant under Student, Teacher
```

However, the attribute *department* is defined separately in *Student* and *Teacher* and thus conflict in *TeachingAssistant*. To avoid a conflict between the two occurrences of *department*, we can rename them by using an **as** clause:

```
create type TeachingAssistant
    under Student with (department as student_dept),
          Teacher with (department as teacher_dept)
```

Notes

- Multiple inheritance is not supported in current SQL standard (up to SQL:1999 and SQL:2003).
- Subtypes can override methods of the supertype.
- Allowing creation of subtypes from types can be controlled by the keywords **final** and **not final**.

Table Inheritance

```
create table people of Person
create table students of Student under people
create table teachers of Teacher under people
```

Notes

- Types of the subtables must be subtypes of the type of the parent table. Therefore, every attribute present in *people* is also present in the subtables.
- When we declare *students* and *teachers* as subtables of *people*, every tuple present in *students* or *teachers* becomes also implicitly present in *people*. Thus, if a query uses the table *people*, it will find not only tuples directly inserted into that table, but also tuples inserted into its subtables, namely *students* and *teachers*. However, only those attributes that are present in *people* can be accessed.
- SQL permits us to find tuples that are in people but not in its subtables by using "**only** *people*" in place of people in a query. The **only** keyword can also be used in delete and update statements. Without the **only** keyword, a delete statement on a supertable, such as *people*, also deletes tuples that were originally inserted in subtables.
- Multiple inheritance of tables is not supported by SQL. However, conceptually it is possible:

  create table *teaching_assistants* **of** *TeachingAssistant* **under** *students*, *teachers*

  As a result of the declaration, every tuple present in the *teaching_assistants* table is also implicitly present in the *teachers* and in the *students* table, and in turn in the *people* table.

  Therefore, SQL subtables cannot be used to represent overlapping specializations from the ERD.

Consistency Requirements for Subtables

There are some consistency requirements for subtables. Before we state the constraints, we need a definition: we say that tuples in a subtable corresponds to tuples in a parent table if they have the same values for all inherited attributes. Thus, corresponding tuples represent the same entity.

The consistency requirements for subtables are:

1. Each tuple of the supertable can correspond to at most one tuple in each of its immediate subtables.
2. SQL has an additional constraint that all the tuples corresponding to each-other must be derived from one tuple (inserted into one table).

For example, without the first condition, we could have two tuples in students (or teachers) that correspond to the same person. The second condition actually prevents a person from being both a teacher and a student.

Collection Types: Arrays and Multisets

- A multiset is an unordered collection where an element may occur multiple times. Multisets are like sets, except that a set allows each element to occur at most once.
- Unlike elements in a multiset, the elements of an array are ordered.

```
create type Book as (
        title varchar(20),
        author_array varchar(20) array[10],
        pub_date date,
        publisher Publisher,
        keyword_set varchar(20) multiset
)
```

Creating Collection Values

We can insert a tuple into the *books* relation as follows:

```
insert into books
values (
        'Compilers',
        array['Smith','Jones'],
        new Publisher('McGraw-Hill','New York'),
        multiset['parsing','analysis'])
)
```

Querying Collection-Valued Attributes: Nesting and Unnesting

Unnesting

The transformation of a nested relation into a form with fewer (or no) relation-valued attributes is called ***unnesting***.

Consider the following *books* relation:

title	***author_array***	***publisher (name, branch)***	***keyword_set***
Compilers	[Smith, Jones]	(McGraw-Hill, New York)	{parsing, analysis}
Networks	[Jones, Frick]	(Oxford, London)	{Internet, Web}

Suppose that we want to convert the relation into a single flat relation, with no nested relations or structured types as attributes. We can use the following query to carry out the task:

```
select title, A.author, publisher.name as pub_name, publisher.branch as pub_branch, K.keyword
from books as B, unnest (B.author_array) as A(author), unnest (B.keyword_set) as K(keyword)
```

The variable *B* in the **from** clause is declared to range over *books*. The variable *A* is declared to range over the authors in *author_array* for the book *B*, and *K* is declared to range over the keywords in the *keyword_set* of the book *B*.

The result of the preceding query is the following relation which is in 1NF:

title	***author***	***pub_name***	***pub_branch***	***keyword***
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

When unnesting an array, the previous query loses information about the ordering of elements in the array. The **unnest with ordinality** clause can be used to get this information, as illustrated by the following query:

```
select title, A.author, publisher.name as pub_name, publisher.branch as pub_branch, K.keyword
from books as B, unnest (B.author_array) with ordinality as A(author), unnest (B.keyword_set) as K(keyword)
```

Nesting

The reverse process of transforming a 1NF relation into a nested relation is called ***nesting***.

Nesting can be carried out by an extension of grouping in SQL. In the normal use of grouping in SQL, a temporary multiset relation is (logically) created for each group, and an aggregate function is applied on the

temporary relation to get a single (atomic) value. The **collect** function returns the multiset of values instead of creating a single value.

The above 1NF relation can be converted back to the nested relation using the following query:

```
select title, collect(author) as author_set, Publisher(pub_name, pub_branch) as publisher,
        collect(keyword) as keyword_set
from flat_books
group by title, publisher
```

Another approach to creating nested relations is to use subqueries in the **select** clause. An advantage of the subquery approach is that an **order by** clause can be optionally used in the subquery to generate results in a desired order, which can then be used to create an array.

The following query illustrates this approach; the keywords **array** and **multiset** specify that an array and multiset (respectively) are to be created from the results of the subqueries.

```
select  title,
        array (
                select author
                from authors as A
                where A.title = B.title
                order by A.position
        ) as author_array,
        Publisher(pub-name, pub-branch) as publisher,
        multiset (
                select keyword
                from keywords as K
                where K.title = B.title
        ) as keyword_set,
from    flat_books as B
```

Updating Multiset Attributes

The SQL standard does not provide any way to update multiset attributes except by assigning a new value. For example, to delete a value *v* from a multiset attribute *A*, we would have to set it to (*A* **except all multiset**[*v*]).

Object-Identity and Reference Types in SQL

Please read the topic (no. 9.6, pages 376-378 – 5th edition) from the book thoroughly… ☺

Implementing O-R Features

Storing Complex Data Types

The complex data types supported by object-relational systems can be translated to the simpler system of relational databases. The techniques for converting E-R model features to tables can be used, with some extensions, to translate object-relational data to relational data at the storage level.

How Subtables can be Stored in an Efficient Manner

Subtables can be stored in an efficient manner, without replication of all inherited fields, in one of two ways:

- Each table stores the primary key (which may be inherited from a parent table) and the attributes are defined locally. Inherited attributes (other than the primary key) do not need to be stored, and can be derived by means of a join with the supertable, based on the primary key.
- Each table stores all inherited and locally defined attributes. When a tuple is inserted, it is stored only in the table in which it is inserted, and its presence is inferred in each of the supertables. Access to all attributes of a tuple is faster, since a join is not required.

How Arrays and Multisets can be Represented

Implementations may choose to represent array and multiset types directly, or may choose to use a normalized representation internally. Normalized representations tend to take up more space and require an extra join / grouping cost to collect data in an array or multiset. However, normalized representations may be easier to implement.

Summary of Strengths of Various Kinds of Database Systems

- **Relational System:** Simple data types, powerful query languages, high protection.
- **Persistent Programming Language-Based OODBs:** Complex data types, integration with programming language, high performance.
- **Object-Relational Systems:** Complex data types, powerful query languages, high protection.

Chapter 15
Transactions

Transactions

A *transaction* is a unit of program execution that accesses and possibly updates various data items.

For example, a transfer of funds from a checking account to a savings account consists of several operations from the point of view of the database system. All these operations result into a single transaction.

The ACID Properties of Transaction

To ensure integrity of the data, we require that the database system maintain the following properties of the transactions:

- **Atomicity**

  Either all operations of the transaction are reflected properly in the database or none at all.

- **Consistency**

  Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

- **Isolation**

  Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

- **Durability**

  After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

How Transaction Accesses Data

Transactions access data using two operations:

- read(X), which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.
- write(X), which transfers the data item X from the local buffer of the transaction that executed the write back to the database.

ACID Properties Explained

Let T_i be a transaction that transfers \$50 from account A to account B. This transaction can be defined as

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Let us now consider each of the ACID requirements.

- **Consistency**

  The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction.

- **Atomicity**

  If the system crashes after write(A) operation, then the database will not be in a consistent state. Thus, it must be ensured that either all of the operations succeed or none of the operations occur.

- **Durability**

  In a real database system, the write operation does not necessarily result in the immediate update of the data on the disk; the write operation may be temporarily stored in memory and executed on the disk later.

  The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

- **Isolation**

  Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way (concurrency problem), resulting in an inconsistent state.

  For example, after the transaction T_i above completes up to the write(A) operation, another transaction T_j concurrently running reads A and B at this point and computes $A + B$, it will observe an inconsistent value.

  Furthermore, if T_j then performs updates on A and B based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

Transaction State

A transaction must be in one of the following states:

- **Active**, the initial state; the transaction stays in this state while it is executing.
- **Partially committed**, after the final statement has been executed.
- **Failed**, after the discovery that normal execution can no longer proceed.
- **Aborted**, after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.

  Two options after a transaction has been aborted:

  1. Restart the transaction; can be done only if hardware or software error occurs.
  2. Kill the transaction - internal logical error.

- **Committed**, after successful completion.

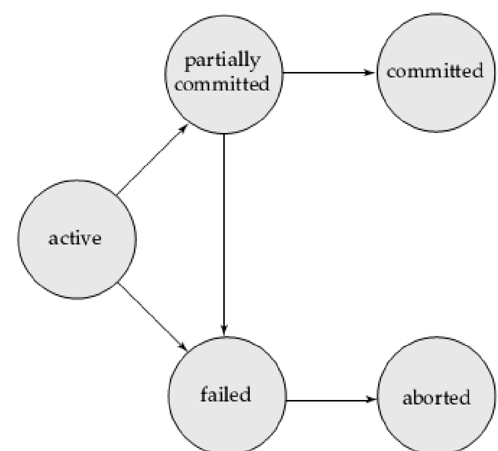


Figure: State diagram of a transaction.

Implementation of Atomicity and Durability

The *recovery-management component* of a database system supports atomicity and durability by a variety of schemes.

The *Shadow-Copy* Scheme

- Assume that only one transaction is active at a time. It also assumes that the database is simply a file on disk.
- A pointer called db_pointer always points to the current consistent copy of the database.
- All updates are made on a *shadow copy* of the database, and **db_pointer** is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.

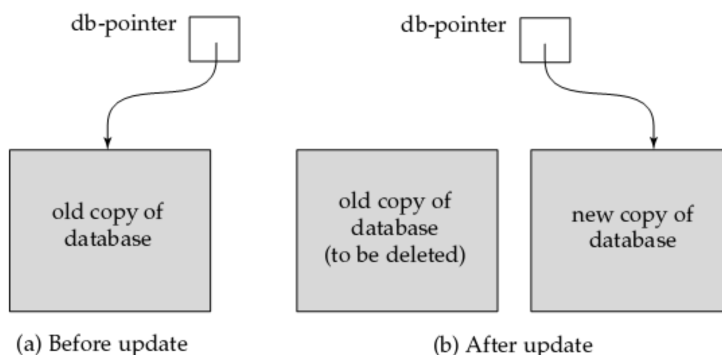


Figure: Shadow-copy technique for atomicity and durability.

- In case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.

Drawbacks of shadow-copy scheme

1. Assumes disks do not fail
2. Useful for text editors, but extremely inefficient for large databases since executing a single transaction requires copying the *entire* database.
3. Does not handle concurrent transactions.

Concurrent Executions

Advantages of Concurrent Executions

Multiple transactions are allowed to run concurrently in the system. Advantages are:

1. **Improved throughput and resource utilization:** *I/O activity* and *CPU activity* can operate in parallel leading to better transaction *throughput.* One transaction can be using the CPU while another is reading from or writing to the disk. The processor and disk utilization also increase; the processor and disk spend less time idle.
2. **Reduced waiting time and average response time:** Short transactions need not wait behind long ones. If the transactions are operating on different parts of the database, it is better to run them concurrently, sharing the CPU cycle and disk accesses among them. It also reduces the *average response time* - the average time for a transaction to be completed after it has been submitted.

Concept of Concurrent Executions

Overview:

First, we need to know how to *schedule* the instructions in transactions for execution.

Next, we'll see how to *serial schedule* transactions, i.e. how to schedule transactions so that one transaction starts executing after another one finishes execution.

Then we'll consider concurrent execution rather than serial execution of transactions. We'll see that concurrent executions might cause the *isolation* property to fail, i.e. the database may become inconsistent because of concurrent executions. We'll try to find out exactly in which cases concurrent execution fails to maintain isolation. Then we'll try to find out how to serial schedule transactions to maintain isolation so that they might appear like they are concurrently executing. More elaborately, we'll split each transaction into pieces and then serial schedule those pieces. This is called *serializability*.

Example transactions we're going to use for explaining the concept of concurrent executions:

Let T_1 transfer $50 from A to B, and T_2 transfer 10% of the balance from A to B. These transactions can be scheduled serially in two ways – T_2 after T_1, and T_1 after T_2. The definition of the transactions and these two possible combinations of serial schedule are depicted as follows:

Schedule 1

T_1	T_2
read(A);	
$A := A - 50$;	
write(A);	
read(B);	
$B := B + 50$;	
write(B)	
	read(A);
	$temp := A * 0.1$;
	$A := A - temp$;
	write(A);
	read(B);
	$B := B + temp$;
	write(B)

Schedule 2

T_1	T_2
	read(A);
	$temp := A * 0.1$;
	$A := A - temp$;
	write(A);
	read(B);
	$B := B + temp$;
	write(B)
read(A);	
$A := A - 50$;	
write(A);	
read(B);	
$B := B + 50$;	
write(B)	

Schedule

A schedule is a sequence of instructions that specify the chronological order in which instructions of transactions are executed.

- A schedule for a set of transactions must consist of all instructions of those transactions.
- A schedule must preserve the order in which the instructions appear in each individual transaction.

  For example, in transaction T_1, the instruction write(A) must appear before the instruction read(B), in any valid schedule.

Serial Schedule

A serial schedule is a schedule which consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.

Thus, for a set of n transactions, there exist $n!$ different valid serial schedules.

The Case of Concurrent Schedules

When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial. If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for a little while, then switch back to the first transaction for some time, and so on.

Several execution sequences are possible, since the various instructions from both transactions may now be interleaved.

In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction. Thus, the number of possible schedules for a set of n transactions is much larger than $n!$

However, not all concurrent executions result in a correct or consistent state.

For example, schedule 3 in the next figure preserves isolation, but schedule 4 doesn't.

We can ensure consistency of the database under concurrent execution by making that any schedule that is executed has the same effect as a schedule that could have occurred without any concurrent execution. That is, the schedule should, in some sense, be equivalent to a *serial schedule*.

Schedule 3

T_1	T_2

read(A);
$A := A - 50$;
write(A);	

	read(A);
	$temp := A * 0.1$;
	$A := A - temp$;
	write(A);

read(B); $B := B + 50$; write(B)	
	read(B); $B := B + temp$; write(B)

Schedule 3 – A concurrent schedule equivalent to schedule 1.

Schedule 4

T_1	T_2
read(A); $A := A - 50$;	

	read(A);
	$temp := A * 0.1$;
	$A := A - temp$;
	write(A);
	read(B);

write(A);
read(B);
$B := B + 50$;
write(B)	

	$B := B + temp$; write(B)

Schedule 4 – A concurrent schedule.

Serializability

Since transactions are programs, it is computationally difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. So, we ignore operations other than read and write instructions; and we assume that, between a read(Q) and a write(Q) instruction on a data item Q, a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction. Our simplified schedules consist of only read and write instructions as depicted in the figure beside.

Schedule 3

T_1	T_2
read(A) write(A)	
	read(A) write(A)
read(B) write(B)	
	read(B) write(B)

Schedule 3 – Showing only the read and write instructions.

In this section we discuss different forms of schedule equivalence; they lead to the notions of *conflict serializability* and *view serializability*.

Conflict Serializability

Conflicting Instructions

We need to find out the cases when concurrent executions fail.

Let us consider a schedule S in which there are two consecutive instructions, l_i and l_j ($i \neq j$) of transactions T_i and T_j respectively.

If l_i and l_j, refer to *different* data item, then we can swap them without affecting the results of any instruction in the schedule.

However, if l_i and l_j refer to *the same* data item Q, then the order of the two steps may matter. The following four cases need to be considered:

Schedule S

T_i	T_j
l_i	
	l_j

Schedule S

T_i	T_j
read(Q)	
	read(Q)

Schedule S

T_i	T_j
read(Q)	
	write(Q)

Schedule S

T_i	T_j
write(Q)	
	read(Q)

Schedule S

T_i	T_j
write(Q)	
	write(Q)

Schedule S

T_i	T_j
	read(Q)
read(Q)	

Schedule S

T_i	T_j
	write(Q)
read(Q)	

Schedule S

T_i	T_j
	read(Q)
write(Q)	

Schedule S

T_i	T_j
	write(Q)
write(Q)	

Order matters?	No	Yes	Yes	Yes
Why?	The same value of Q is read by T_i and T_j regardless of the order.	If I_i comes before I_j, T_i doesn't read the value of Q that is written by T_j. Else, T_i reads the value of Q that is written by T_j.	Similar to the previous case.	The value obtained by the next read(Q) instruction of S is affected, since the result of only the latter write instruction is preserved in the database.

So, instructions l_i and l_j of transactions T_i and T_j *conflict* if and only if there exists *same* item Q accessed by both l_i and l_j and at least one of these instructions is a write operation on Q.

Conflict Equivalence and Conflict Serializability

Let l_i and l_j be consecutive instructions of a schedule S. If l_i and l_j are instructions of different transactions and l_i and l_j do not conflict, then we can swap the order of l_i and l_j to produce a new schedule S'. We expect S to be equivalent of S', since all instructions appear in the same order in both schedules except for l_i and l_j, whose order does not matter.

If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are ***conflict equivalent***.

We say that a schedule S is ***conflict serializable*** if it is conflict equivalent to a serial schedule.

Schedule 3

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
read(B)	
	read(A)
	write(A)
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
read(B)	
	read(A)
write(B)	
	write(A)
	read(B)
	write(B)

Schedule 6

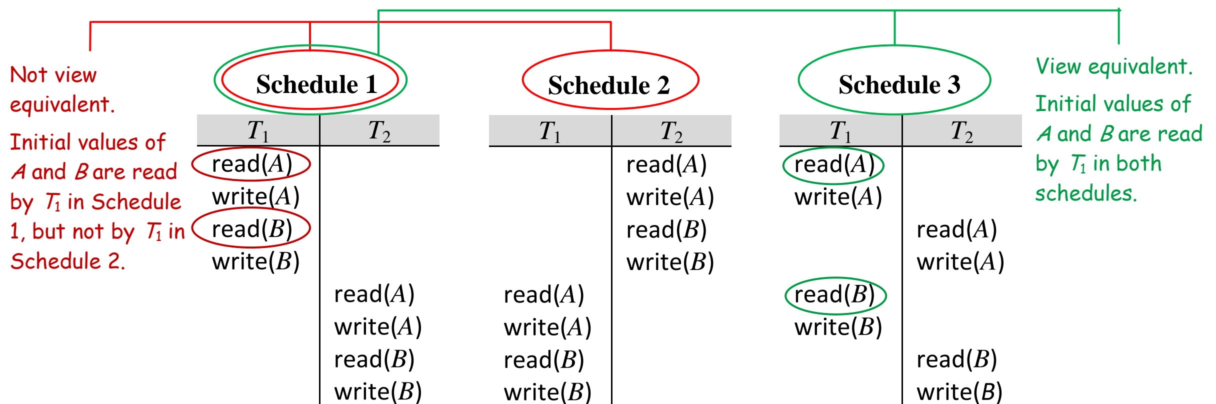
T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figure: Transforming Schedule 3 – which is conflict equivalent to serial schedule 6 – into Serial Schedule 6.

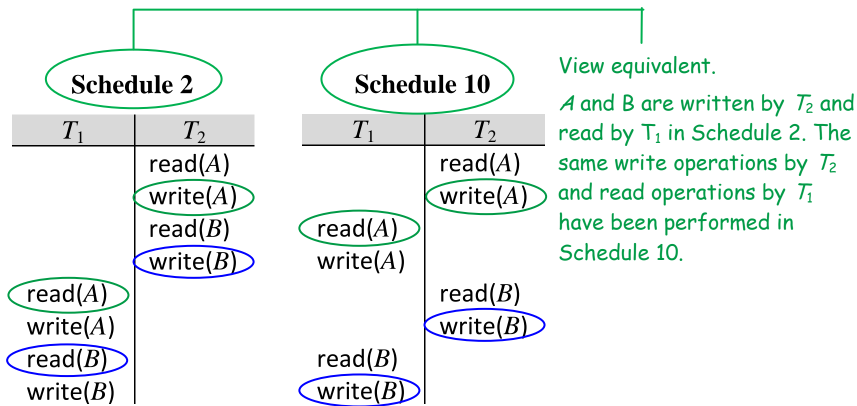
View Serializability

Let S and S' be two schedules with the same set of transactions. The schedules S and S' are said to be ***view equivalent*** if the following three conditions are met:

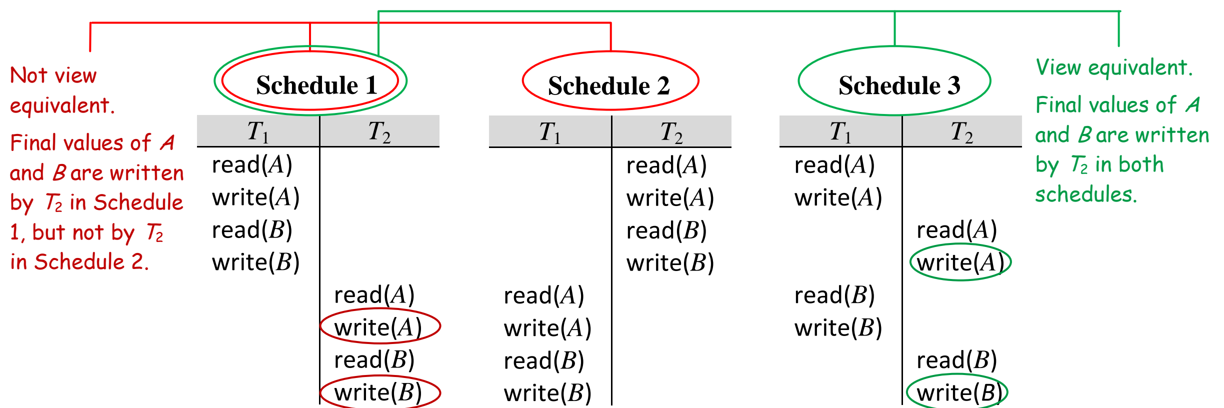
1. For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i must, in schedule S', also read the initial value of Q.



2. For each data item Q, if transaction T_i executes **read**(Q) in schedule S, and that value was produced by a **write**(Q) operation executed by transaction T_j, then the **read**(Q) operation of transaction T_i must, in schedule S', also read the value of Q that was produced by the same **write**(Q) operation in transaction T_j.

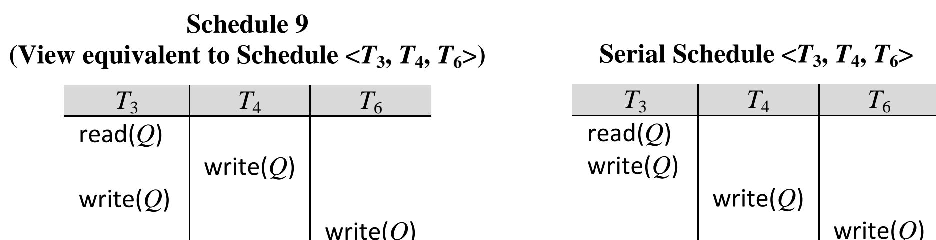


3. For each data item Q, the transaction (if any) that performs the final **write**(Q) operation in schedule S must perform the final **write**(Q) operation in schedule S'.



Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.

Every conflict serializable schedule is also view serializable, but there are view serializable schedules that are *not* conflict serializable. Schedule 9 is view serializable, but not conflict serializable, since every pair of consecutive instructions conflict, and, thus, no swapping of instructions is possible.



Blind Writes

Observe that, in schedule 9, transactions T_4 and T_6 perform write(Q) operations without having performed a read(Q) operation. Writes of this sort is called ***blind writes*** that appear in every view serializable schedule that is not conflict serializable.

Recoverability

So far we have assumed that there are no transaction failures. We now need to address the effect of transaction failures on concurrently running transactions.

If a transaction T_i fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transaction T_j that is dependent on T_i (i.e. T_j has read data written by T_i) is also aborted. To achieve this surety, we need to place restrictions on the type of schedules permitted in the system.

Recoverable Schedules

A ***recoverable schedule*** is one where, for each pair of transactions T_i and T_j such that transaction T_j reads a data item previously written by a transaction T_i, the commit operation of T_i appears before the commit operation of T_j.

Cascading Rollback and Cascadeless Schedules

The phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called ***cascading rollback***.

A ***cascadeless schedule*** is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.

Testing for Serializability

The problem: How to determine, given a particular schedule S, whether the schedule is serializable?

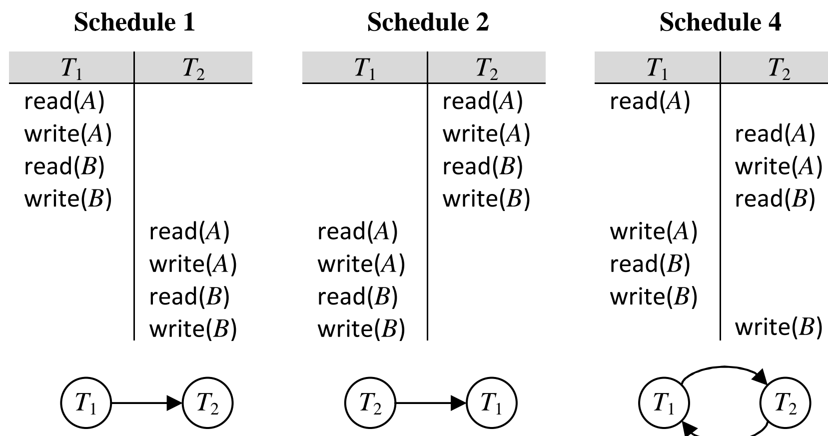
Testing for Conflict Serializability

1. Consider a schedule S of a set of transactions T_1, T_2, ..., T_n.
2. We construct a directed graph called ***precedence graph*,** from S. This graph consists of a pair $G = (V, E)$, where V is the set of vertices and E is a set of edges.

   The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all the edges $T_i \rightarrow T_j$ for which one of the three conditions holds:

   a. T_i executes write(Q) before T_j executes read(Q).
   b. T_i executes read(Q) before T_j executes write(Q).
   c. T_i executes write(Q) before T_j executes write(Q).

   If an edge $T_i \rightarrow T_j$ exists in the precedence graph, then, in any serial schedule S' equivalent to S, T_i must appear before T_j.
3. If the precedence graph has a cycle, then schedule S is *not* conflict serializable. If no cycle exists, then it is conflict serializable.
4. If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.



Schedule 1

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 2

T_1	T_2
	read(A)
	write(A)
	read(B)
	write(B)
read(A)	
write(A)	
read(B)	
write(B)	

Schedule 4

T_1	T_2
read(A)	
	read(A)
	write(A)
	read(B)
write(A)	
read(B)	
write(B)	
	write(B)

Testing for View Serializability

The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems. Extension of the testing for conflict serializability to test for view serializability has cost exponential in the size of the precedence graph.

However practical algorithms that just check some *sufficient conditions* for view serializability can still be used. That is, if the sufficient conditions are satisfied, the schedule is view serializable, but there may be view serializable schedules that do not satisfy sufficient conditions.

Chapter 16
Concurrency Control

Objective

How to ensure/implement the serializability property of concurrent schedules?

Concurrency Schemes/Protocols

1. **Lock-Based Protocols**
   Theme: Data items are accessed in a mutually exclusive manner.
   ***Locking Modes*:**
   1. Shared – Read-only
   2. Exclusive – Read-Write
   Locking Protocol:
   When a transaction T_i requests a lock on a data item Q in a particular mode M, the lock can be granted provided that:
   1. There is no other transaction holding a lock on Q in a mode that conflicts with M. [Ensuring serializability]
   2. There is no other transaction that is waiting for a lock on Q and that made its lock request before T_i. [Ensuring that starvation doesn't occur]
      a. **Two-Phase Locking Protocol**
      Each transaction issues lock and unlock requests in two phases:
      1. Growing Phase – May obtain locks, but may not release locks
      2. Shrinking Phase – May release locks, but may not obtain any new locks
      Problems:
      1. Doesn't ensure freedom from deadlock.
      2. Cascading rollback might occur.
         Solution:
         1. **Strict Two-Phase Locking Protocol**
            Requirements:
            1. Two-phase locking
            2. All *exclusive-mode* locks taken by a transaction be held until that transaction commits.
         2. **Rigorous Two-Phase Locking Protocol**
            Requirements:
            1. Two-phase locking
            2. *All* locks taken by a transaction be held until that transaction commits.
      3. Concurrency might become less.
         Solution:
         Two-Phase Locking with Lock Conversion
         Requirements:
         1. Two-phase locking
         2. A shared lock can be upgraded to an exclusive lock in the growing phase, and an exclusive lock can be downgraded to a shared lock in the shrinking phase.
2. **Graph-Based Protocols**
   Theme: Construct locking protocols requiring having prior knowledge about the order in which the database items will be accessed.
      a. **Tree Protocol**
      The only lock instruction allowed is lock-X. Each transaction T_i can lock a data item at most once, and must observe the following rules:
      1. The first lock by T_i may be on any data item.

2. Subsequently, a data item Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i.

Advantages:

1. Ensures conflict serializability.
2. Ensures freedom from deadlock.

Problems:

1. Doesn't ensure recoverability.
2. Doesn't ensure cascadelessness.

How to ensure recoverability and cascadelessness:

The protocol can be modified to not permit release of exclusive locks until the end of the transaction.

Problem with this solution:

Holding exclusive locks until the end of the transaction reduces concurrency.

Alternative solution improving concurrency, but ensuring only recoverability:

For each data item with an uncommitted write, we record which transaction performed the last write to the data item. Whenever a transaction T_i performs a read of an uncommitted data item, we record a *commit dependency* of T_i on the transaction that performed the last write to the data item. Transaction T_i is then not permitted to commit until the commit of all transactions on which it has a commit dependency. If any of these transactions aborts, T_i must also be aborted.

Advantages over two-phase locking protocol:

1. Unlike two-phase locking, it's deadlock-free, so no rollbacks are required.
2. Unlocking may occur earlier which may lead to shorter waiting times and to an increase in concurrency.

Disadvantages:

1. In some cases, a transaction may have to lock data items that it does not access. This additional locking results in increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency.
2. Without prior knowledge of what data items will need to be locked, transactions will have to lock the root of the tree, and that can reduce concurrency greatly.

3. Timestamp-Based Protocols

Theme: Determines the serializability order by selecting an ordering among transactions in advance.

Timestamp Values:

1. **W-timestamp(Q)** – largest timestamp of any transaction that executed write(Q) successfully.
2. **R-timestamp(Q)** – largest timestamp of any transaction that executed read(Q) successfully.

a. Timestamp-Ordering Protocol

Objective: Ensures that any conflicting read and write operations are executed in timestamp order.

Protocol Operation:

1. Suppose a transaction T_i issues a **read**(Q).
   a. If TS(T_i) < W-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
   Hence, the **read** operation is rejected, and T_i is rolled back.
   b. If TS(T_i) ≥ W-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).
2. Suppose that transaction T_i issues **write**(Q).
   a. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
   Hence, the **write** operation is rejected, and T_i is rolled back.
   b. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
   Hence, this **write** operation is rejected, and T_i is rolled back.
   c. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

If a transaction T_i is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

Advantages:

1. Ensures conflict serializability.
2. Ensures freedom from deadlock.

Disadvantages:

1. Possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction.
   Possible Solution:
   If a transaction is found to be getting restarted repeatedly, conflicting transactions need to be temporarily blocked to enable the transaction to finish.
2. Generates schedules that are not recoverable.
   Possible Solutions:
   1. (Recoverability and cascadelessness)
      A transaction is structured such that its writes are all performed at the end of its processing.
      All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written.
      A transaction that aborts is restarted with a new timestamp.
   2. (Recoverability and cascadelessness)
      Limited form of locking; whereby reads of uncommitted items are postponed until the transaction that updated the item commits.
   3. (Only recoverability)
      Use commit dependencies to ensure recoverability.

b. Thomas' Write Rule

Objective: Allowing greater potential concurrency than the general timestamp ordering protocol.
Protocol:
Exactly the same as the general timestamp ordering protocol except that in rule 2(b), the write operation is ignored in cases where $\text{TS}(T_i) \geq \text{R-timestamp}(Q)$.

4. Validation-Based Protocols

Theme: In cases where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low. It may be better to use a scheme that imposes less overhead.

a. Optimistic Concurrency Control Scheme

The validation test for transaction T_j requires that, for all T_i with $\text{TS}(T_i) < \text{TS}(T_j)$ either one of the following conditions holds:

1. **finish**(T_i) < **start**(T_j)
2. **start**(T_j) < **finish**(T_i) < **validation**(T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j.

Advantage: Automatically guards against cascading rollbacks.
Disadvantage: Possibility of starvation of long transactions.
Avoiding Starvation: Conflicting transactions must be temporarily blocked to enable the long transaction to finish.

5. Multiple Granularity[1]

Theme: So far the concurrency-control schemes described used each individual data item as the unit on which synchronization has performed. There are circumstances, where it would be advantageous to group several data items and to treat them as one individual synchronization unit. So, a mechanism is needed to allow the system to define multiple levels of *granularities*. This allows data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones.

a. Multiple Granularity Locking Protocol

Each transaction T_i can lock a node Q, using the following rules:

[1] *Granularity*: the quality of being composed of relatively large particles.

1. The lock compatibility matrix must be observed.
2. The root of the tree must be locked first, and may be locked in any mode.
3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.

Advantages:

1. Enhances concurrency
2. Reduces lock overhead

Disadvantage: Deadlock is possible. However, there are techniques to reduce deadlock frequency in this protocol, and also to eliminate deadlock entirely.

6. Multiversion Schemes

Theme: The concurrency-control schemes discussed thus far ensure serializability by either delaying an operation or aborting the transaction that issued the operation. These difficulties could be avoided if old copies of each data item were kept in a system.

a. Multiversion Timestamp-Ordering Scheme

Suppose that transaction T_i issues a **read**(Q) or **write**(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).

1. If transaction T_i issues a **read**(Q), then the value returned is the content of version Q_k.
2. If transaction T_i issues a **write**(Q)
    1. If TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back.
    2. If TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten.
    3. Else a new version of Q is created.

Advantage: A read request never fails and is never made to wait. In typical database systems, where reading is a more frequent operation than is writing, this advantage may be of major practical significance.

Disadvantages:

1. The reading of a data item also requires the updating of the R-timestamp field, resulting in two potential disk accesses, rather than one.
2. The conflicts between transactions are resolved through rollbacks, rather than through waits. This alternative may be expensive.
   Possible Solution: Use multiversion two-phase locking.
3. Does not ensure recoverability and cascadelessness.
   Possible Solution: It can be extended in the same manner as the basic timestamp-ordering scheme to make it recoverable and cascadeless.

b. Multiversion Two-Phase Locking

Objective: Attempts to combine the advantages of multiversion concurrency control with the advantages of two-phase locking.

Operation:

1. Read-only transactions are assigned a timestamp by reading the current value of ts-counter before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.
2. When an update transaction wants to read a data item, it obtains a shared lock on it, and reads the latest version.
3. When it wants to write an item, it obtains X-lock on it; it then creates a new version of the item and sets this version's timestamp to ∞.
4. When update transaction T_i completes, commit processing occurs:
    a. T_i sets timestamp on the versions it has created to **ts-counter** + 1
    b. T_i increments **ts-counter** by 1

Concurrency Control

What is a concurrency control scheme and why is it needed?

One of the fundamental properties of a transaction is isolation. When several transactions execute concurrently in the database, the isolation property may no longer be preserved. To ensure this, the system must control the interaction among the concurrent transactions and this control is achieved through one of the variety of mechanisms called ***concurrency-control*** schemes.

The concurrency-control schemes are based on the serializability property. That is, all the schemes ensure that the schedules are serializable schedules.

Lock-Based Protocols

One way to ensure serializability is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item.

The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item.

Locks

A lock is a mechanism to control concurrent access to a data item.

Locking Modes

Data items can be locked in two modes:

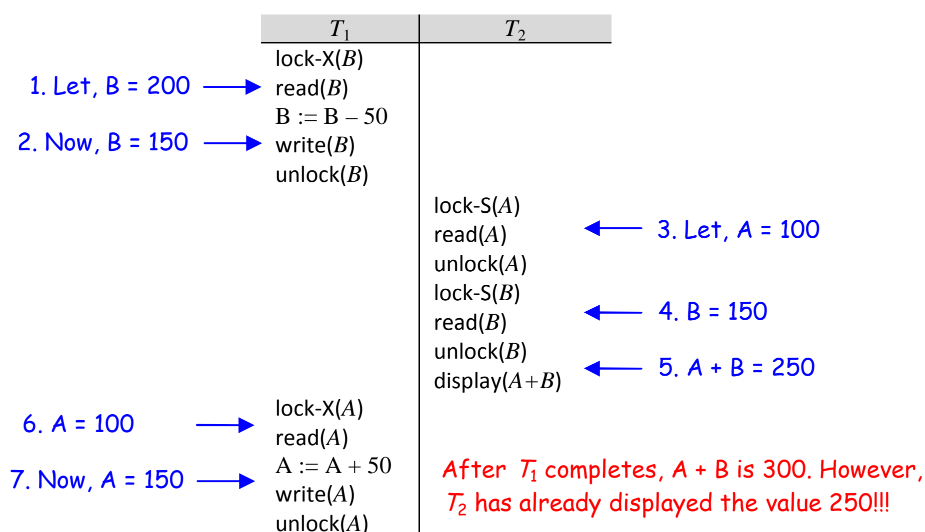
1. **Shared (S) Mode –** data item can only be read. If a transaction T_i has obtained a shared-mode lock on item Q, then T_i can read, but cannot write Q. S-lock is requested using **lock-S** instruction.
2. **Exclusive (X) Mode –** data item can be both read and written. If a transaction T_i has obtained an exclusive-mode lock on item Q, then T_i can both read and write Q. X-lock is requested using **lock-X** instruction.

How Locks Work

- A lock on a data item can be granted to a transaction if:
  1. No other transaction is holding a lock on it.
  2. A transaction is holding an S-lock and the requesting transaction is requesting for an S-lock.
- A transaction must hold a lock on a data item as long as it accesses that item.

  Problem with this: For a transaction to unlock a data item immediately after its final access of that data item is not always desirable, since serializability may not be ensured.

  Illustration of this problem:



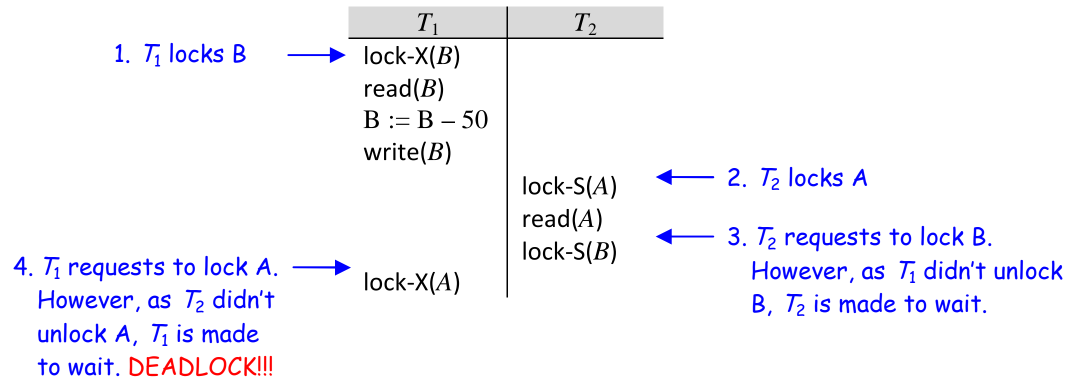
Solution of this problem:

Delay unlocking to the end of the transaction.

- **Problems with locking**

1. Deadlock

Illustration of deadlock:



Solution: Coming later in this chapter.

2. Starvation

A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item. Thus the first transaction never makes progress and is said to be starved.

Solution:

When a transaction T_i requests a lock on a data item Q in a particular mode M, the lock can be granted provided that:

1. There is no other transaction holding a lock on Q in a mode that conflicts[2] with M.
2. There is no other transaction that is waiting for a lock on Q and that made its lock request before T_i.

The Two-Phase Locking Protocol

This protocol requires that each transaction issue lock and unlock requests in two phases:

1. **Growing phase**
   - Transaction may obtain locks
   - Transaction may not release any lock
2. **Shrinking phase**
   - Transaction may release locks
   - Transaction may not obtain any new locks

T_3
lock-X(B)
read(B)
B := B – 50
write(B)
lock-X(A)
read(A)
A := A + 50
write(A)
unlock(B)
unlock(A)

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

Example:

Transaction T_3 on the right side is two-phase.

Note that the unlock instructions do not need to appear at the end of the transaction. For example, in T_3, we could move the unlock(B) instruction to just after the lock-X(A) instruction.

[2] In the lock-compatibility matrix below, all the modes other than the S-S are conflicting:

	S	**X**
S	true	false
X	false	false

Problems with two-phase locking protocol:

1. **Deadlock is not ensured.**

   The transactions T_1 and T_2 are in two-phase, but not deadlock-free as illustrated in the previous illustration of deadlock.

   Solution: see deadlock handling later in this chapter.

2. **Cascading rollback may occur.**

   Illustration of this problem:

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-X(A)
		read(A)

Failure of T_5 after this step leads to cascading rollback of T_6 and T_7.

   Solution:

   The two-phase locking protocol may be modified in any of the following ways:

   1. **Strict two-phase locking protocol:** requires not only that locking be two-phase, but also that *all exclusive-mode locks* taken by a transaction be held until that transaction commits.

      This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.

   2. **Rigorous two-phase locking protocol:** requires not only that locking be two-phase, but also that *all locks* taken by a transaction be held until that transaction commits.

3. **Concurrency might become less.**

   Illustration of this problem:

T_8	T_9
read(a_1)	
read(a_2)	
...	
read(a_n)	
write(a_1)	
	read(a_1)
	read(a_2)
	display(a_1+a_2)

As T_8 is writing a_1, so it must X-lock a_1 before read(a_1) in two-phase locking.

However, if T_8 could initially lock a_1 in shared mode and then could later change the lock to exclusive mode, we could get more concurrency, since T_8 and T_9 could access a_1 and a_2 simultaneously.

   Solution:

   Allow lock conversion.

   - We shall provide a mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock.
   - However, upgrading can take place only in the growing phase, whereas downgrading can take place only in the shrinking phase.

- Note that a transaction attempting to upgrade a lock on an item Q may be forced to wait. This enforced wait occurs if Q is currently locked by another transaction in *shared* mode.
- Further, if exclusive locks are held until the end of the transaction, the schedules are cascadeless.

Example:

T_8	T_9
lock-S(a_1)	
	lock-S(a_1)
lock-S(a_2)	
	lock-S(a_2)
lock-S(a_3)	
lock-S(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S(a_n)	
upgrade(a_1)	

Use of two-phase locking protocol:

Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.

Graph-Based Protocols

If we have prior knowledge about the order in which the database items will be accessed, it is possible to construct locking protocols that are not two phase but ensures conflict serializability.

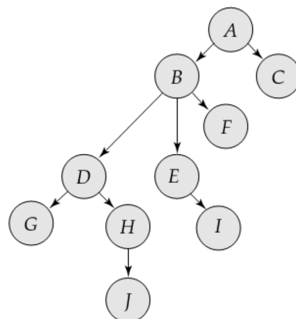
To acquire such prior knowledge, we impose a partial ordering $\rightarrow$ on the set D = $\{d_1, d_2, .., d_h\}$ of all data items. If $d_i \rightarrow d_j$, then any transaction accessing both d_i and d_j must access d_i before accessing d_j. This partial ordering may be the result of either the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control.

The Tree Protocol

In the tree protocol, the only lock instruction allowed is lock-X. Each transaction T_i can lock a data item at most once, and must observe the following rules:

1. The first lock by T_i may be on any data item.
2. Subsequently, a data item Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i.

Example:



T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock(G)			

Problem with Tree Protocol:

The tree protocol does not ensure recoverability and cascadelessness.

Solution:

To ensure recoverability and cascadelessness, the protocol can be modified to not permit release of exclusive locks until the end of the transaction.

Problem with this solution:

Holding exclusive locks until the end of the transaction reduces concurrency.

Alternate solution improving concurrency, but ensuring only recoverability:

For each data item with an uncommitted write, we record which transaction performed the last write to the data item. Whenever a transaction T_i performs a read of an uncommitted data item, we record a *commit dependency* of T_i on the transaction that performed the last write to the data item. Transaction T_i is then not permitted to commit until the commit of all transactions on which it has a commit dependency. If any of these transactions aborts, T_i must also be aborted.

Advantages (Over Two-Phase Locking Protocol):

1. Unlike two-phase locking, it's deadlock-free, so no rollbacks are required.
2. Unlocking may occur earlier which may lead to shorter waiting times and to an increase in concurrency.

Disadvantages:

1. In some cases, a transaction may have to lock data items that it does not access. This additional locking results in increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency.

   For example, a transaction that needs to access data items A and J in the database graph depicted previously must lock not only A and J, but also data items B, D and H.
2. Without prior knowledge of what data items will need to be locked, transactions will have to lock the root of the tree, and that can reduce concurrency greatly.

Comments:

Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.

Timestamp-Based Protocols

Another method for determining the serializability order is to select an ordering among transactions in advance using *timestamp-ordering*.

Timestamps

Each transaction is assigned a timestamp when it enters the system.

If an old transaction T_i has timestamp TS(T_i), a new transaction T_j is assigned timestamp TS(T_j) such that TS(T_i) < TS(T_j).

How a timestamp can be generated:

1. **Value of system clock –** A transaction's timestamp is equal to the value of the clock when it enters the system.
2. **Logical counter –** It is incremented after a new timestamp has been assigned. A transaction's timestamp is equal to the value of the counter when it enters the system.

Basic Concept

The timestamps determine the serializability order. Thus, if TS(T_i) < TS(T_j), then the system must ensure that the produced schedule is equivalent to a serial schedule in which T_i appears before T_j.

To implement this, the protocol maintains for each data Q two timestamp values:

1. **W-timestamp(Q)** – largest timestamp of any transaction that executed write(Q) successfully.
2. **R-timestamp(Q)** – largest timestamp of any transaction that executed read(Q) successfully.

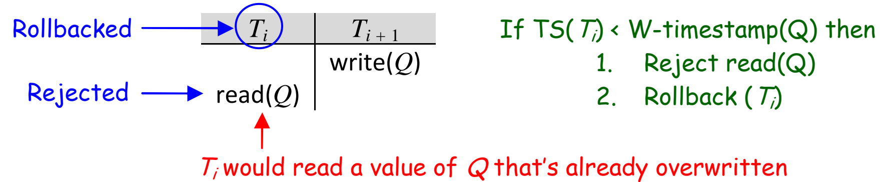
These timestamps are updated whenever a new read(Q) or write(Q) instructions are executed.

The Timestamp Ordering Protocol

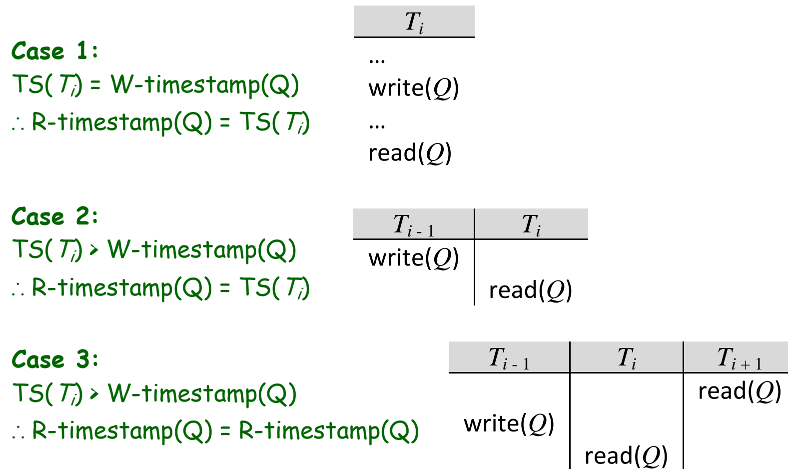
The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.

1. Suppose a transaction T_i issues a **read**(Q).

   a. If TS(T_i) < W-timestamp(Q), then T_i needs to read a value of Q that was already overwritten. Hence, the **read** operation is rejected, and T_i is rolled back.



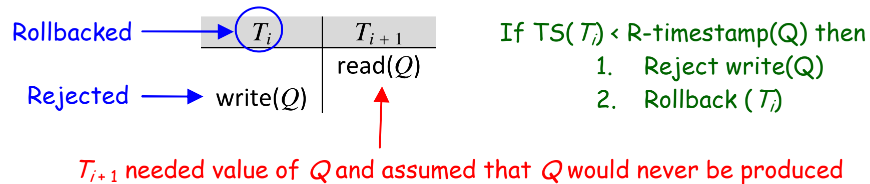
   b. If TS(T_i) ≥ W-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).



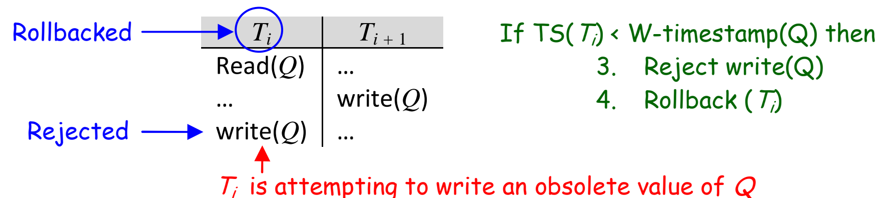
2. Suppose that transaction T_i issues **write**(Q).

   a. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.

   Hence, the **write** operation is rejected, and T_i is rolled back.



   b. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this **write** operation is rejected, and T_i is rolled back.



   c. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

If a transaction T_i is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

Advantages:

1. Ensures conflict serializability.
2. Ensures freedom from deadlock as no transaction ever waits.

Disadvantages:

1. Possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction.

   Solution:

   If a transaction is found to be getting restarted repeatedly, conflicting transactions need to be temporarily blocked to enable the transaction to finish.

2. Generates schedules that are not recoverable and may require cascading rollbacks.

   Possible Solutions:

   1. Ensuring both recoverability and cascadelessness

      A transaction is structured such that its writes are all performed at the end of its processing.

      All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written.

      A transaction that aborts is restarted with a new timestamp.

   2. Ensuring both recoverability and cascadelessness

      Limited form of locking; whereby reads of uncommitted items are postponed until the transaction that updated the item commits.

   3. Ensuring only recoverability

      Use commit dependencies to ensure recoverability.

Thomas' Write Rule

Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances, thus allowing greater potential concurrency.

T_i	T_{i+1}
read(Q)	
	write(Q)
write(Q)	

The timestamp ordering protocol requires that T_i be rolled back if T_i issues write(Q) and TS(T_i) < W-timestamp(Q). However, in Thomas' Write Rule, in those cases where TS(T_i) ≥ R-timestamp(Q), we ignore the obsolete write.

Thomas' Write Rule allows greater potential concurrency. This change makes it possible to generate some serializable schedules that are not possible under the other protocols.

Validation-Based Protocols

In cases where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low. It may be better to use a scheme that imposes less overhead.

Phases of a Transaction

A difficulty in reducing the overhead is to know in advance which transactions will be involved in a conflict. To gain this knowledge, a scheme for monitoring the system is needed.

We assume that execution of transaction T_i is done in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order:

1. **Read and execution phase:** The Transaction T_i reads the value of various data items and store them in variables local to T_i. It performs all write operations on temporary local variables, without updates of the actual database.

2. **Validation phase**: Transaction T_i performs a *validation test* to determine if local variables can be written to database without violating serializability.
3. **Write phase**: If T_i is validated, the updates are applied to the actual database; otherwise, T_i is rolled back.

The Validation Test

To perform the validation test, it is needed to know when the various phases of transaction T_i took place. It is associated with 3 timestamps:

1. **Start(T_i):** the time when T_i started its execution.
2. **Validation(T_i):** the time when T_i entered its validation phase.
3. **Finish(T_i):** the time when T_i finished its write phase.

Serializability order is determined by timestamp given at validation time to increase concurrency. Thus, TS(T_i) = Validation(T_i).

The validation test for transaction T_j requires that, for all T_i with TS(T_i) < TS(T_j) either one of the following conditions holds:

1. **finish**(T_i) < **start**(T_j).

   Since T_i completes its execution before T_j started, the serializability order is indeed maintained.

2. **start**(T_j) < **finish**(T_i) < **validation**(T_j). That is, the set of data items written by T_i does not intersect with the set of data items read by T_j, and T_i completes its write phase before T_j starts its validation phase.

   This condition ensures that the writes of T_i and T_j do not overlap. Since the writes of T_i do not affect the read of T_j, and since T_j cannot affect the read of T_i, the serializability order is indeed maintained.

Example:

T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
<*validate*>	
display($A + B$)	
	<*validate*>
	write(B)
	write(A)

Advantage:

Automatically guards against cascading rollbacks, since the actual writes take place only after the transaction issuing the write has committed.

Problem:

Possibility of starvation of long transactions, due to a sequence of conflicting short transactions that cause repeated restarts of the long transaction.

Solution: Conflicting transactions must be temporarily blocked to enable the long transaction to finish.

Comments:

The validation scheme is also called as ***optimistic concurrency control*** since transaction executes fully in the hope that all will go well during validation. In contrast, locking and timestamp ordering are ***pessimistic*** in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serializable.

Multiple Granularity

The Problem

In the concurrency-control schemes described thus far, we have used each individual data item as the unit on which synchronization is performed. There are circumstances, however, where it would be advantageous to group several data items, and to treat them as one individual synchronization unit.

For example, if a transaction T_i needs to access the entire database, and a locking protocol is used, then T_i must lock each item in the database. Clearly, executing these locks is time consuming. It would be better if T_i could issue a single lock request to lock the entire database. On the other hand, if transaction T_j needs to access only a few data items, it should not be required to lock the entire database, since otherwise concurrency is lost.

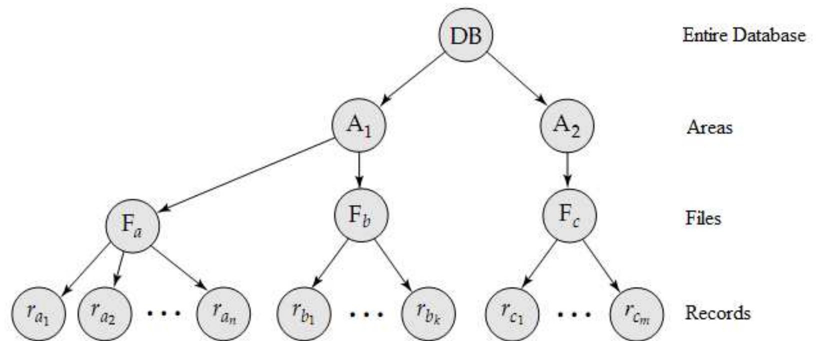
Solution

What is needed is a mechanism to allow the system to define multiple levels of granularity. We can make one by allowing data items to be of various sizes and defining a hierarchy of data granularities, where the small granularities are nested within larger ones. Such a hierarchy can be represented graphically as a tree.

Difference between the Multiple Granularity tree and the tree in Tree Protocol

A nonleaf node of the multiple-granularity tree represents the data associated with its descendants. In the tree protocol, each node is an independent data item.

Illustration of the Protocol



Locking Nodes

- Each node in the tree can be locked individually.
- There are two lock modes – *shared* and *exclusive*.
- When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode.

  For example, if transaction T_i gets an *explicit* lock on file F_b of the above figure in exclusive mode, then it has an *implicit* lock in exclusive mode all the records belonging to that file. It does not need to lock the individual records of F_b explicitly.

Problem 1: How the system would determine whether a transaction can lock a node?

Suppose that transaction T_j wishes to lock record r_{b6} of file F_b. Since T_i has locked F_b explicitly, it follows that r_{b6} is also locked (implicitly). But, when T_j issues a lock request for r_{b6}, r_{b6} is not explicitly locked! How does the system determine whether T_j can lock r_{b6}?

Solution to Problem 1

T_j must traverse the tree from the root to record r_{b6}. If any node in that path is locked in an incompatible mode, then T_j must be delayed.

Problem 2: How does the system determine if the root node can be locked?

Suppose now that transaction T_k wishes to lock the entire database. To do so, it simply must lock the root of the hierarchy. Note, however, that T_k should not succeed in locking the root node, since T_i is currently holding a lock on part of the tree (specifically, on file F_b). But how does the system determine if the root node can be locked?

A Possible Solution to Problem 2

T_k should search the entire tree.

Problem with this solution: This solution defeats the whole purpose of the multiple-granularity locking scheme.

More Efficient Solution to Problem 2

Introduce a new class of lock modes, called *intention lock modes*.

- If a node is locked in an intention mode, explicit locking is being done at a lower level of the tree (that is, at a finer granularity).
- Intention locks are put on all the ancestors of a node before that node is locked explicitly.
- A transaction wishing to lock a node — say, Q — must traverse a path in the tree from the root to Q. While traversing the tree, the transaction locks the various nodes in an intention mode.

Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully.

Different Types of Intention Mode Locks

1. **Intention-Shared (IS) Mode:** If a node is locked in intention-shared (IS) mode, explicit locking is being done at a lower level of the tree, but with only shared-mode locks.
2. **Intention-Exclusive (IX) Mode:** If a node is locked in intention-exclusive (IX) mode, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks.
3. **Shared and Intention-Exclusive (SIX) mode:** If a node is locked in shared and intention-exclusive (SIX) mode, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks.

Compatibility Function for the Various Lock Modes

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

The Multiple-Granularity Locking Protocol

Each transaction T_i can lock a node Q by following these rules:

1. It must observe the lock-compatibility function of various lock modes.
2. It must lock the root of the tree first, and can lock it in any mode.
3. It can lock a node Q in S or IS mode only if it currently has the parent of Q locked in either IX or IS mode.
4. It can lock a node Q in X, SIX, or IX mode only if it currently has the parent of Q locked in either IX or SIX mode.
5. It can lock a node only if it has not previously unlocked any node (that is, T_i is two phase).
6. It can unlock a node Q only if it currently has none of the children of Q locked.

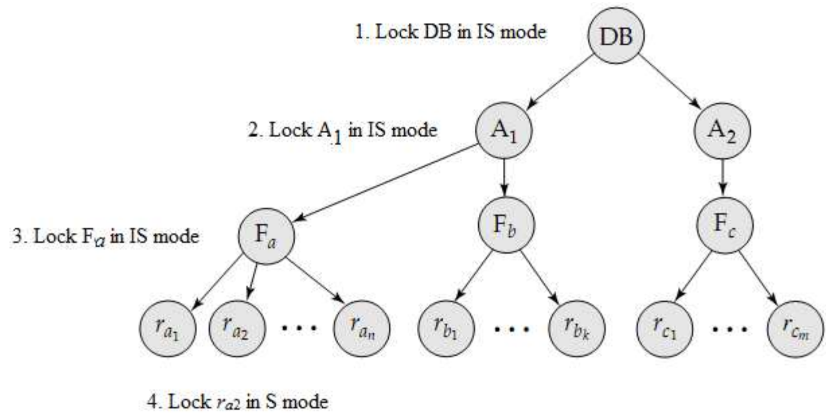
Observe that the multiple-granularity protocol requires that locks be acquired in top-down (root-to-leaf) order, whereas locks must be released in bottom-up (leaf-to-root) order.

Examples

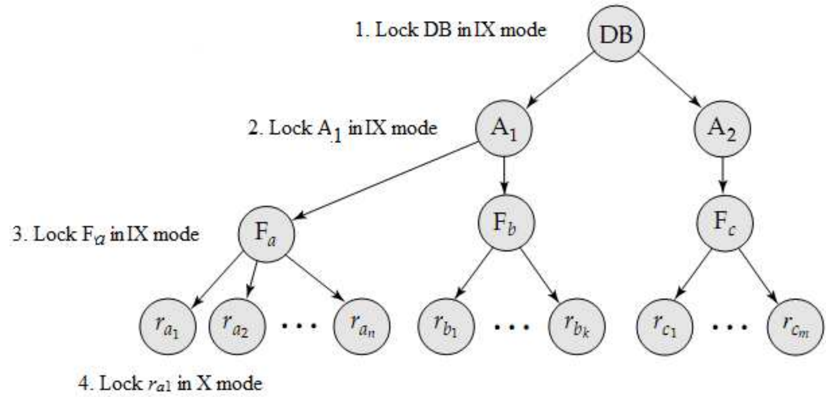
Consider the following four transactions.

Note that transactions T_{16}, T_{18}, and T_{19} can access the database concurrently. Transaction T_{17} can execute concurrently with T_{16}, but not with either T_{20} or T_{21}.

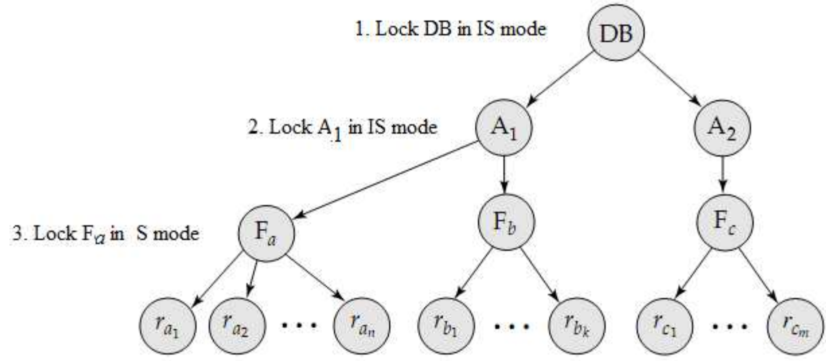
T_{16}
read(r_{a2})



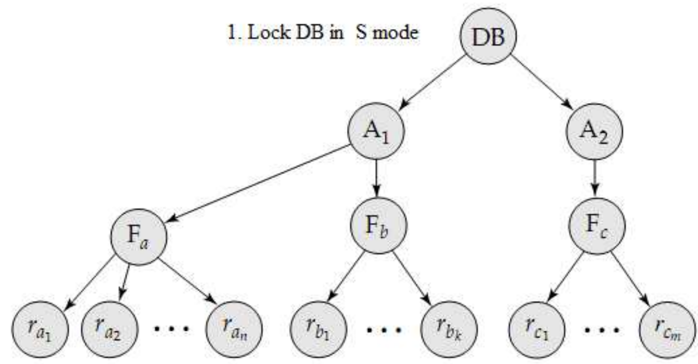
T_{17}
write(r_{a1})



T_{18}
read(F_a)



T_{19}
read(DB)



Advantages

1. Ensures serializability
2. Enhances concurrency and reduces lock overhead.
3. Particularly useful in applications that include a mix of
   a. Short transactions that access only a few data items
   b. Long transactions that produce reports from an entire file or set of files

Disadvantage

Deadlock is possible in the protocol that we have, as it is in the two-phase locking protocol.

However, there are techniques to reduce deadlock frequency in the multiple-granularity protocol, and also to eliminate deadlock entirely.

Deadlock Handling

A system is said to be in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions $\{T_0, T_1, \ldots, T_n\}$ such that T_0 is waiting for a data item that T_1 holds, and T_1 is waiting for a data item that T_2 holds, and …, and T_{n-1} is waiting for a data item that T_n holds, and T_n is waiting for a data item that T_0 holds. None of the transactions can make progress in such a situation.

Principal Methods for Dealing with Deadlock Problem

1. We can use a ***deadlock prevention*** protocol to ensure that the system will never enter a deadlock state.
2. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a ***deadlock detection and deadlock recovery*** scheme.

As we shall see, both methods may result in transaction rollback.

Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise, detection and recovery are more efficient.

Deadlock Prevention

There are two approaches to deadlock prevention:

1. Ensuring that no cycle waits can occur by ordering the requests for locks, or requiring all locks to be acquired together.

Different Schemes for this approach:

1. Each transaction locks all its data items before it begins execution. Moreover, either all are locked in one step or none are locked.

   Disadvantages:

   1. It is often hard to predict, before the transaction begins, what data items need to be locked.
   2. Data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

2. Impose an ordering of all data items, and a transaction is required to lock data items only in a sequence consistent with the ordering.

   We have seen one such scheme in the tree protocol, which uses a partial ordering of data items.

3. A variation of the above approach is to use a total order of data items, in conjunction with two-phase locking. Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the ordering.

   This scheme is easy to implement, as long as the set of data items accessed by a transaction is known when the transaction starts execution.

2. Using preemption and transaction rollbacks.

- In preemption, when a transaction T_2 requests a lock that transaction T_1 holds, the lock granted to T_1 may be preempted by rolling back of T_1, and granting of the lock to T_2.
- To control the preemption, we assign a unique timestamp to each transaction. The system uses these timestamps only to decide whether a transaction should wait or roll back.
- If a transaction is rolled back, it retains its old timestamp when restarted.
- Locking is still used for concurrency control.

Deadlock Prevention Schemes using Timestamps

1. **Wait-Die Scheme** [*Non-Preemptive Technique*]

   When transaction T_i requests a data item currently held by T_j, T_i is allowed to wait only if it has a timestamp smaller than that of T_j (i.e. T_i is older than T_j). Otherwise, T_i is rolled back (dies).

   For example, suppose that transactions T_{22}, T_{23} and T_{24} have timestamps 5, 10 and 15 respectively. If T_{22} requests a data item held by T_{23}, then T_{22} will wait. If T_{24} requests a data item held by T_{23}, then T_{24} will be rolled back.

2. **Wound-Wait Scheme** [*Preemptive Technique*]

   This scheme is a counterpart to the wait-die scheme.

   When transaction T_i requests a data item currently held by T_j, T_i is allowed to wait only if it has a timestamp larger than that of T_j (i.e. T_i is younger than T_j). Otherwise, T_j is rolled back (T_j is wounded by T_i).

   Returning to our example, with transactions T_{22}, T_{23} and T_{24}, if T_{22} requests a data item held by T_{23}, then the data item will be preempted from T_{23}, and T_{23} will be rolled back. If T_{24} requests a data item held by T_{23}, then T_{24} will wait.

Wait-Die Scheme **Wound-Wait Scheme**

Proof That Both Wait-Die and Wound-Wait Schemes Avoid Starvation

At any time, there is a transaction with the smallest timestamp. This transaction cannot be required to roll back in either scheme. Since timestamps always increase, and since transactions are not assigned new timestamps when they are rolled back, a transaction that is rolled back repeatedly will eventually have the smallest timestamp, at which point it will not be rolled back again.

Differences Between Wait-Die and Wound-Wait Schemes

1. In the wait–die scheme, an older transaction must wait for a younger one to release its data item. Thus, the older the transaction gets, the more it tends to wait.

   By contrast, in the wound–wait scheme, an older transaction never waits for a younger transaction.

2. In the wait–die scheme, if a transaction T_{i} dies and is rolled back because it requested a data item held by transaction T_j, then T_i may reissue the same sequence of requests when it is restarted. If the data item is still held by T_j, then T_i will die again. Thus, T_i may die several times before acquiring the needed data item.

   Contrast this series of events with what happens in the wound–wait scheme. Transaction T_i is wounded and rolled back because T_j requested a data item that it holds. When T_i is restarted and requests the data item now being held by T_j , T_i waits. Thus, there may be fewer rollbacks in the wound–wait scheme.

***Major Problem with Both of the Schemes*:** Unnecessary rollbacks may occur.

Timeout-Based Scheme

- In this approach, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts.
- If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed.

- This scheme falls somewhere between deadlock prevention, where a deadlock will never occur, and deadlock detection and recovery.

Advantages

1. Particularly easy to implement
2. Works well if transactions are short and if long waits are likely to be due to deadlocks.

Disadvantages

1. In general, it is hard to decide how long a transaction must wait before timing out. Too long a wait results in unnecessary delays once a deadlock has occurred. Too short a wait results in transaction rollback even when there is no deadlock, leading to wasted resources.
2. Starvation is also a possibility with this scheme.

Deadlock Detection

The *Wait-For* Graph

Deadlocks can be described precisely in terms of a directed graph called a *wait-for* graph.

This graph consists of a pair $G = (V, E)$, where

V is a set of vertices which consists of all the transactions in the system

E is a set of edges where each element is an ordered pair $T_i \rightarrow T_j$.

If $T_i \rightarrow T_j$ is in E, then there is a directed edge from transaction T_i to T_j, implying that transaction T_i is waiting for transaction T_j to release a data item that it needs.

When transaction T_i requests a data item currently being held by transaction T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when transaction T_j is no longer holding a data item needed by transaction T_i.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

Example

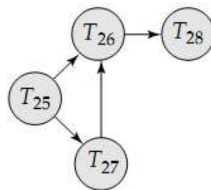


Figure 16.18 Wait-for graph with no cycle.

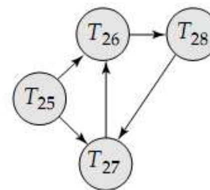


Figure 16.19 Wait-for graph with a cycle.

When Should the Detection Algorithm be Invoked?

The answer depends on two factors:

1. How often does a deadlock occur?
2. How many transactions will be affected by the deadlock?

If deadlocks occur frequently, then the detection algorithm should be invoked more frequently than usual. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken. In addition, the number of cycles in the graph may also grow. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

Deadlock Recovery

When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

1. **Selection of a Victim**

Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one. Many factors may determine the cost of a rollback, including

a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
b. How many data items the transaction has used.
c. How many more data items the transaction needs for it to complete.
d. How many transactions will be involved in the rollback.

2. **Rollback**

Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.

1. **Total Rollback:** Abort transaction and then restart it.
2. **Partial Rollback:** Roll back the transaction only as far as necessary to break the deadlock.

3. **Starvation**

In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is starvation. We must ensure that transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.