
Basics of Functional Dependencies and Normalization for Relational Databases: Boyce-Codd Normal Form, Multi-valued Dependencies and Fourth Normal Form, Join Dependencies and Fifth Normal Form.

Relational Database Design Algorithms and Further Dependencies: Inference Rules, Equivalence, and Minimal cover.

Introduction to Transaction Processing Concepts and Theory: Introduction to transaction processing, desirable properties of transactions, characterizing schedules based on Serializability.

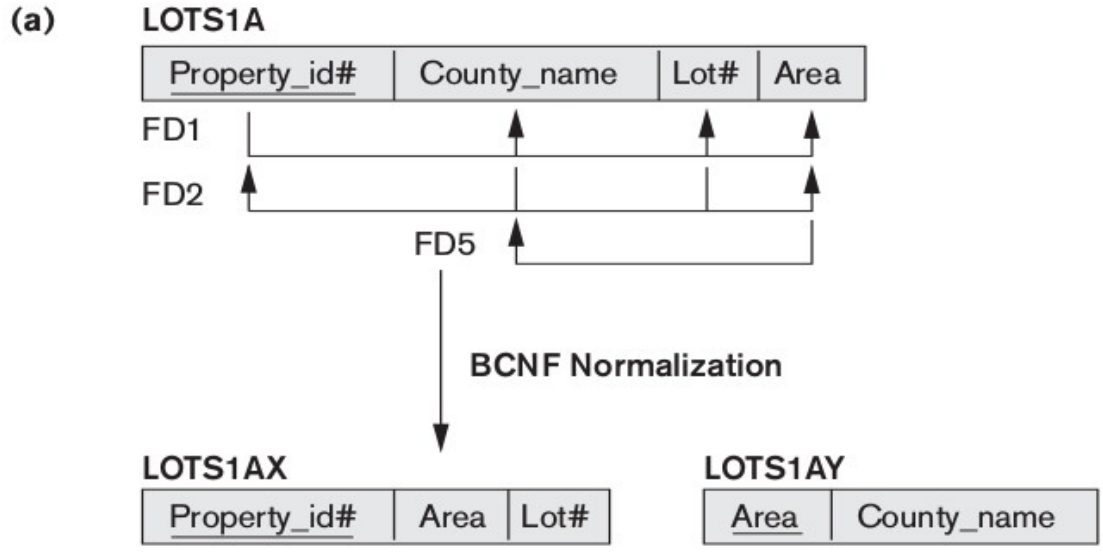
Concurrency Control Techniques: Two-phase locking techniques for concurrency control.

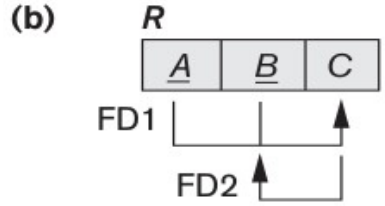
Boyce-Codd Normal Form

We can see the need for a stronger normal form than 3NF by going back to the LOTS relation schema, with its four functional dependencies FD1 through FD4 . Suppose that we have thousands of lots in the relation but the lots are from only two counties: DeKalb and Fulton. Suppose also that lot sizes in DeKalb County are only 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0 acres, whereas lot sizes in Fulton County are restricted to 1.1, 1.2, ..., 1.9, and 2.0 acres. In such a situation we would have the additional functional dependency FD5: Area→County_name. If we add this to the other dependencies, the relation schema LOTS1A still is in 3NF because County_name is a prime attribute.

The area of a lot that determines the county, as specified by FD5, can be represented by 16 tuples in a separate relation R(Area, County_name), since there are only 16 possible Area values. This representation reduces the redundancy of repeating the same information in the thousands of LOTS1A tuples. BCNF is a stronger normal form that would disallow LOTS1A and suggest the need for decomposing it.

Definition. A relation schema R is in BCNF if whenever a nontrivial functional dependency X → A holds in R, then X is a superkey of R.





The

formal definition of BCNF differs from the definition of 3NF in that condition (b) of 3NF, which allows A to be prime, is absent from BCNF. That makes BCNF a stronger normal form compared to 3NF. In our example, FD5 violates BCNF in LOTS1A because AREA is not a superkey of LOTS1A.

We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure above. This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation after decomposition.

As another example, consider Figure below, which shows a relation TEACH with the following dependencies:

FD1: { Student , Course } → Instructor

FD2: Instructor → Course

Note that {Student, Course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure (b) above, with Student as A, Course as B, and Instructor as C. Hence this relation is in 3NF but not BCNF. Decomposition of this relation schema into two schemas is not straightforward because it may be decomposed into one of the three following possible pairs:

1. { Student , Instructor } and { Student , Course }.
2. { Course , Instructor } and { Course , Student }.
3. { Instructor , Course } and { Instructor , Student }.

All three decompositions lose the functional dependency FD1. The desirable decomposition of those just shown is 3 because it will not generate spurious tuples after a join. We make sure that we meet this property, because nonadditive decomposition is a must during normalization.

TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

Multivalued Dependency and Fourth Normal Form

Here we discuss the concept of multivalued dependency (MVD) and define fourth normal form, which is based on this dependency. Multivalued dependencies are a consequence of first normal form (1NF), which disallows an attribute in a tuple to have a set of values, and the accompanying process of converting an unnormalized relation into 1NF.

If we have two or more multivalued independent attributes in the same relation schema, we get into a problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain the independence among the attributes involved. This constraint is specified by a multivalued dependency.

For example, consider the relation EMP shown in Figure below. A tuple in this EMP relation represents the fact that an employee whose name is Ename works on the project whose name is Pname and has a dependent whose name is Dname. An employee may work on several projects and may have several dependents, and the employee's projects and dependents are independent of one another. To keep the relation state consistent, and to avoid any spurious relationship between the two independent attributes, we must have a separate tuple to represent every combination of an employee's dependent and an employee's project. This constraint is specified as a multivalued dependency on the EMP relation.

(a) EMP

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

Formal Definition of Multivalued Dependency

Definition. A multivalued dependency $X \twoheadrightarrow Y$ specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation state r of R: If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in r with the following properties, where we use Z to denote $(R - (X \cup Y))$:

- $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
- $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$.
- $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$.

Whenever $X \twoheadrightarrow Y$ holds, we say that X multidetermines Y. Because of the symmetry in the definition, whenever $X \twoheadrightarrow Y$ holds in R, so does $X \twoheadrightarrow Z$. Hence, $X \twoheadrightarrow Y$ implies $X \twoheadrightarrow Z$, and therefore it is sometimes written as $X \twoheadrightarrow Y \mid Z$.

An MVD $X \twoheadrightarrow Y$ in R is called a trivial MVD if (a) Y is a subset of X, or (b) $X \cup Y = R$. For example, the relation EMP_PROJECTS in Figure (b) below has the trivial Ename$\twoheadrightarrow$ Pname. An MVD that satisfies neither (a) nor (b) is called a nontrivial MVD.

(b) EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y

EMP_DEPENDENTS

Ename	Dname
Smith	John
Smith	Anna

If we have a nontrivial MVD in a relation, we may have to repeat values redundantly in the tuples. In the EMP relation of Figure (a), the values 'X' and 'Y' of Pname are repeated with each value of Dname (or, by symmetry, the values 'John' and 'Anna' of Dname are repeated with each value of Pname). This redundancy is clearly undesirable. Therefore, we need to define a fourth normal form that is stronger than BCNF and disallows relation schemas such as EMP. Notice that relations containing nontrivial MVDs tend to be all-key relations—that is, their key is all their attributes taken together.

We now present the definition of fourth normal form (4NF), which is violated when a relation has undesirable multivalued dependencies, and hence can be used to identify and

decompose such relations.

Definition. A relation schema R is in 4NF with respect to a set of dependencies if, for every nontrivial multivalued dependency $X \twoheadrightarrow Y$, X is a superkey for R.

We can state the following points:

- An all-key relation is always in BCNF since it has no Fds.
- An all-key relation such as the EMP relation in Figure (a), which has no FDs but has the MVD Ename $\twoheadrightarrow$ Pname | Dname, is not in 4NF.
- A relation that is not in 4NF due to a nontrivial MVD must be decomposed to convert it into a set of relations in 4NF.
- The decomposition removes the redundancy caused by the MVD.

The process of normalizing a relation involving the nontrivial MVDs that is not in 4NF consists of decomposing it so that each MVD is represented by a separate relation where it becomes a trivial MVD. Consider the EMP relation in Figure (a). EMP is not in 4NF because in the nontrivial MVDs Ename$\twoheadrightarrow$Pname and Ename$\twoheadrightarrow$Dname, and Ename is not a superkey of EMP. We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, shown in Figure (b). Both EMP_PROJECTS and EMP_DEPENDENTS are in 4NF, because the MVDs Ename$\twoheadrightarrow$Pname in EMP_PROJECTS and Ename$\twoheadrightarrow$Dname in EMP_DEPENDENTS are trivial MVDs.

Join Dependencies and Fifth Normal Form

In some cases there may be no nonadditive join decomposition of R into two relation schemas, but there may be a nonadditive join decomposition into more than two relation schemas. Moreover, there may be no functional dependency in R that violates any normal form up to BCNF, and there may be no nontrivial MVD present in R either that violates 4NF. We then resort to another dependency called the join dependency and, if it is present, carry out a multiway decomposition into fifth normal form (5NF).

Definition. A join dependency (JD), denoted by JD(R_1, R_2, ...,R_n), specified on relation schema R, specifies a constraint on the states r of R. The constraint states that every legal state r of R should have a nonadditive join decomposition into R_1, R_2, ...,R_n. Hence, for every such r we have

- $(\pi_{R1}(r), \pi_{R2}(r), ...,\pi_{Rn}(r)) = r$

A join dependency JD(R_1, R_2, ...,R_n), specified on relation schema R, is a trivial JD if one of the relation schemas R_i in JD(R_1, R_2, ...,R_n) is equal to R. Such a dependency is called trivial because it has the nonadditive join property for any relation state r of R and thus does not specify any constraint on R. We can now define fifth normal form, which is also called project-join normal form.

Definition. A relation schema R is in fifth normal form (5NF) (or project-join normal form (PJNF)) with respect to a set F of functional, multivalued, and join dependencies if, for every nontrivial join dependency JD(R_1, R_2, ...,R_n) in F^+ (that is, implied by F), every R_i is a superkey of R.

For an example of a JD, consider once again the SUPPLY all-key relation in Figure below. Suppose that the following additional constraint always holds: Whenever a supplier s supplies part p, and a project j uses part p, and the supplier s supplies at least one part to project j, then supplier s will also be supplying part p to project j. This constraint can be restated in other ways and specifies a join dependency JD(R_1, R_2, R_3) among the three projections R_1(Sname, Part_name), R_2(Sname, Proj_name), and R_3(Part_name, Proj_name) of SUPPLY. If this constraint holds, the tuples below the dashed line in Figure must exist in any legal state of the SUPPLY relation that also contains the tuples above the dashed line.

SUPPLY

Sname	Part_name	Proj_name
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

Figure below shows how the SUPPLY relation with the join dependency is decomposed into three relations R1, R2, and R3 that are each in 5NF. Notice that applying a natural join to any two of these relations produces spurious tuples, but applying a natural join to all three together does not. Discovering JDs in practical databases with hundreds of attributes is next to impossible. It can be done only with a great degree of intuition about the data on the part of the designer.

R_1

Sname	Part_name
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R_2

Sname	Proj_name
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

R_3

Part_name	Proj_name
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

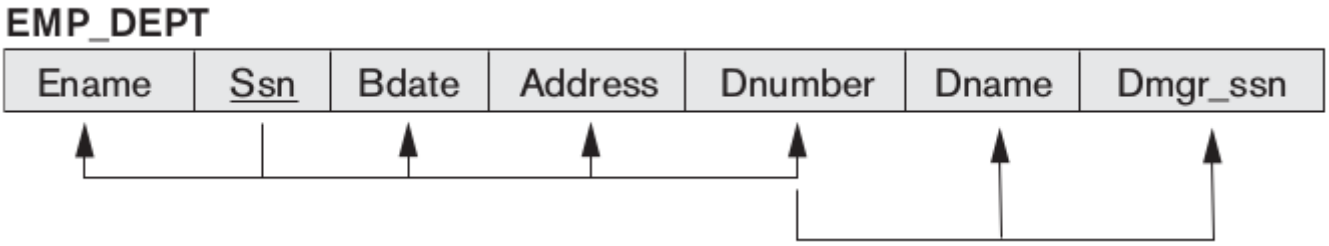
Further Topics in Functional Dependencies: Inference Rules, Equivalence, and Minimal Cover

Now we return to the study of functional dependencies and show how new dependencies can be inferred from a given set and discuss the concepts of closure, equivalence, and minimal cover that we will need when we later consider a synthesis approach to design of relations given a set of Fds.

Inference Rules for Functional Dependencies

We denote by F the set of functional dependencies that are specified on relation schema R. In real life, it is impossible to specify all possible functional dependencies for a given situation. For example, if each department has one manager, so that Dept_no uniquely determines Mgr_ssn (Dept_no → Mgr_ssn), and a manager has a unique phone number called Mgr_phone (Mgr_ssn → Mgr_phone), then these two dependencies together imply that Dept_no → Mgr_phone. This is an inferred FD and need not be explicitly stated in addition to the two given FDs. Therefore, it is useful to define a concept called closure formally that includes all possible dependencies that can be inferred from the given set F.

Definition. Formally, the set of all dependencies that include F as well as all dependencies that can be inferred from F is called the closure of F; it is denoted by F^+. For example, suppose that we specify the following set F of obvious functional dependencies on the relation schema in Figure below:



F = {Ssn → {Ename, Bdate, Address, Dnumber}, Dnumber → {Dname, Dmgr_ssn}}

Some of the additional functional dependencies that we can infer from F are the following:

Ssn → {Dname, Dmgr_ssn}

Ssn → Ssn

Dnumber → Dname

An FD X → Y is inferred from a set of dependencies F specified on R if X → Y holds in every legal relation state r of R; that is, whenever r satisfies all the dependencies in F, X → Y also holds in r. The closure F^+ of F is the set of all functional dependencies that can be inferred from F.

To determine a systematic way to infer dependencies, we must discover a set of inference rules that can be used to infer new dependencies from a given set of dependencies. We use the notation F| = X → Y to denote that the functional dependency X → Y is inferred from the set of functional dependencies F. The FD {X, Y} → Z is abbreviated to XY → Z, and the FD {X, Y, Z} → {U, V} is abbreviated to XYZ → UV. The following six rules IR1 through IR6 are well-known inference rules for functional dependencies:

IR1 (reflexive rule) 1 : If X ⊇ Y, then X →Y.

IR2 (augmentation rule) 2 : {X → Y} |= XZ → YZ.

IR3 (transitive rule): {X → Y, Y → Z} |= X → Z.

IR4 (decomposition or projective rule): {X → YZ} |=X → Y.

IR5 (union or additive rule): {X → Y, X → Z} |= X → YZ.

IR6 (pseudotransitive rule): {X → Y, WY → Z} |=WX → Z.

1. The reflexive rule (IR1) states that a set of attributes always determines itself or any of its subsets, which is obvious. Because IR1 generates dependencies that are always true, such dependencies are called trivial. Formally, a functional dependency X → Y is trivial if X ⊇ Y; otherwise, it is nontrivial.
2. The augmentation rule (IR2) says that adding the same set of attributes to both the left and right-hand sides of a dependency results in another valid dependency.
3. According to IR3, functional dependencies are transitive.
4. The decomposition rule (IR4) says that we can remove attributes from the right-hand side of a dependency; applying this rule repeatedly can decompose the FD X → {A_1, A_2, ..., A_n} into the set of dependencies {X → A_1, X → A_2, ..., X → A_n}.
5. The union rule (IR5) allows us to do the opposite; we can combine a set of dependencies {X → A_1, X → A_2, ..., X → A_n} into the single FD X → {A_1, A_2, ..., A_n}.
6. The pseudotransitive rule (IR6) allows us to replace a set of attributes Y on the left hand side of a dependency with another set X that functionally determines Y, and can be derived from IR2 and IR3 if we augment the first functional dependency X → Y with W (the augmentation rule) and then apply the transitive rule.

Each of the preceding inference rules can be proved from the definition of functional dependency, either by direct proof or by contradiction.

Proof of IR1. Suppose that $X \supseteq Y$ and that two tuples t_1 and t_2 exist in some relation instance r of R such that $t_1[X] = t_2[X]$. Then $t_1[Y] = t_2[Y]$ because $X \supseteq Y$; hence, $X \rightarrow Y$ must hold in r.

Proof of IR2 (by contradiction). Assume that $X \rightarrow Y$ holds in a relation instance r of R but that $XZ \rightarrow YZ$ does not hold. Then there must exist two tuples t_1 and t_2 in r such that

1. $t_1[X] = t_2[X]$
2. $t_1[Y] = t_2[Y]$
3. $t_1[XZ] = t_2[XZ]$, and
4. $t_1[YZ] \neq t_2[YZ]$. This is not possible because from (1) and (3) we deduce
5. $t_1[Z] = t_2[Z]$, and from (2) and (5) we deduce
6. $t_1[YZ] = t_2[YZ]$, contradicting (4).

Proof of IR3. Assume that (1) $X \rightarrow Y$ and (2) $Y \rightarrow Z$ both hold in a relation r. Then for any two tuples t_1 and t_2 in r such that $t_1[X] = t_2[X]$, we must have (3) $t_1[Y] = t_2[Y]$, from assumption (1); hence we must also have (4) $t_1[Z] = t_2[Z]$ from (3) and assumption (2); thus $X \rightarrow Z$ must hold in r.

A simpler way to prove that an inference rule for functional dependencies is valid is to prove it by using inference rules that have already been shown to be valid. For example, we can prove IR4 through IR6 by using IR1 through IR3 as follows.

Proof of IR4 (Using IR1 through IR3).

1. $X \rightarrow YZ$ (given).
2. $YZ \rightarrow Y$ (using IR1 and knowing that $YZ \supseteq Y$).
3. $X \rightarrow Y$ (using IR3 on 1 and 2).

Proof of IR5 (using IR1 through IR3).

1. $X \rightarrow Y$ (given).
2. $X \rightarrow Z$ (given).
3. $X \rightarrow XY$ (using IR2 on 1 by augmenting with X; notice that XX = X).
4. $XY \rightarrow YZ$ (using IR2 on 2 by augmenting with Y).
5. $X \rightarrow YZ$ (using IR3 on 3 and 4).

Proof of IR6 (using IR1 through IR3).

1. $X \rightarrow Y$ (given).
2. $WY \rightarrow Z$ (given).
3. $WX \rightarrow WY$ (using IR2 on 1 by augmenting with W).
4. $WX \rightarrow Z$ (using IR3 on 3 and 2).

It has been shown by Armstrong (1974) that inference rules IR1 through IR3 are **sound and complete**. By sound, we mean that given a set of functional dependencies F specified on a relation schema R, any dependency that we can infer from F by using IR1 through IR3 holds in every relation state r of R. By complete, we mean that using IR1 through IR3 repeatedly to infer dependencies until no more dependencies can be inferred results in the complete set of all possible dependencies that can be inferred from F. In other words, the set of dependencies F^+, which we called the closure of F, can be determined from F by using only inference rules IR1 through IR3. Inference rules IR1 through IR3 are known as Armstrong's inference rules.

Typically, database designers first specify the set of functional dependencies F that can easily be determined from the semantics of the attributes of R; then IR1, IR2, and IR3 are used to infer additional functional dependencies that will also hold on R. A systematic way to determine these additional functional dependencies is first to determine each set of attributes X that appears as a left-hand side of some functional dependency in F and then to determine the set of all attributes that are dependent on X.

Definition. For each such set of attributes X, we determine the set X^+ of attributes that are functionally determined by X based on F; X^+ is called the closure of X under F.

Algorithm 1. Determining X^+, the Closure of X under F

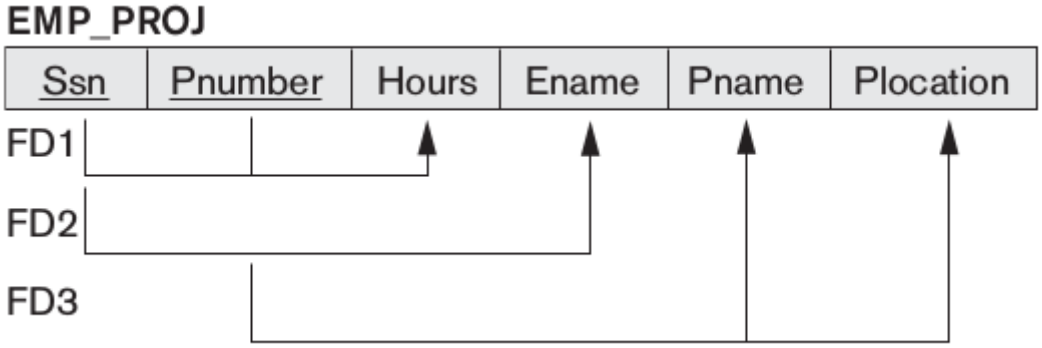
Input: A set F of FDs on a relation schema R, and a set of attributes X, which is a subset of R.

$X^+ := X$;
repeat
old$X^+ := X^+$;
for each functional dependency $Y \rightarrow Z$ in F do
    if $X^+ \supseteq Y$ then $X^+ := X^+ \cup Z$;
until (X^+ = oldX^+);

Algorithm starts by setting X^+ to all the attributes in X. By IR1, we know that all these attributes are functionally dependent on X. Using inference rules IR2 and IR3, we add attributes to X^+, using each functional dependency in F. We keep going through all the dependencies in F (the repeat loop) until no more attributes are added to X^+ during a complete cycle (of the for loop) through the dependencies in F.

For example, consider the relation schema EMP_PROJ in Figure below; from the semantics of the attributes, we specify the following set F of functional dependencies that should hold on EMP_PROJ :

F = { Ssn → Ename, Pnumber →{ Pname, Plocation }, {Ssn, Pnumber } → Hours }



Using Algorithm, we calculate the following closure sets with respect to F:

$\{Ssn\}^+$ = { Ssn, Ename }

$\{Pnumber\}^+$ = { Pnumber, Pname, Plocation }

$\{Ssn, Pnumber\}^+$ = { Ssn, Pnumber, Ename, Pname, Plocation, Hours}

Equivalence of Sets of Functional Dependencies

Definition. A set of functional dependencies F is said to cover another set of functional dependencies E if every FD in E is also in F^+; that is, if every dependency in E can be inferred from F; alternatively, we can say that E is covered by F.

Definition. Two sets of functional dependencies E and F are equivalent if $E^+ = F^+$. Therefore, equivalence means that every FD in E can be inferred from F, and every FD in F can be inferred from E; that is, E is equivalent to F if both the conditions—E covers F and F covers E—hold.

It is left to the reader as an exercise to show that the following two sets of FDs are equivalent:

F = {A → C, AC → D, E → AD, E → H}
and G = {A → CD, E → AH}.

Minimal Sets of Functional Dependencies

Informally, a minimal cover of a set of functional dependencies E is a set of functional dependencies F that satisfies the property that every dependency in E is in the closure F^+ of F. To satisfy these properties, we can formally define a set of functional dependencies F to be minimal if it satisfies the following conditions:

1. Every dependency in F has a single attribute for its right-hand side.
2. We cannot replace any dependency X → A in F with a dependency Y → A, where Y is a proper subset of X, and still have a set of dependencies that is equivalent to F.
3. We cannot remove any dependency from F and still have a set of dependencies that is equivalent to F.

Definition. A minimal cover of a set of functional dependencies E is a minimal set of dependencies that is equivalent to E. We can always find at least one minimal cover F for any set of dependencies E using Algorithm below.

Algorithm 2 Finding a Minimal Cover F for a Set of Functional Dependencies E

Input: A set of functional dependencies E.

1. Set F := E.
2. Replace each functional dependency X → {A_1, A_2, ..., A_n} in F by the functional dependencies X →A_1, X →A_2, ..., X → A_n.
3. For each functional dependency X → A in F for each attribute B that is an element of X if { {F – {X → A} } ∪ { (X – {B}) → A} } is equivalent to F then replace X → A with (X – {B}) → A in F.
4. For each remaining functional dependency X → A in F if {F – {X → A} } is equivalent to F, then remove X → A from F.

We illustrate the above algorithm with the following:

Let the given set of FDs be E: {B → A, D → A, AB → D}. We have to find the minimal cover of E.

- All above dependencies are in canonical form (that is, they have only one attribute on the right-hand side), so we have completed step 1 of Algorithm and can proceed to step 2. In step 2 we need to determine if AB → D has any redundant attribute on the left-hand side; that is, can it be replaced by B → D or A → D?
- Since B → A, by augmenting with B on both sides (IR2), we have BB → AB, or B→ AB (i). However, AB → D as given (ii).
- Hence by the transitive rule (IR3), we get from (i) and (ii), B → D. Thus AB → D may be replaced by B → D.
- We now have a set equivalent to original E, say E′: {B → A, D → A, B → D}. No further

reduction is possible in step 2 since all FDs have a single attribute on the left-hand side.

- In step 3 we look for a redundant FD in E′. By using the transitive rule on B → D and D → A, we derive B → A. Hence B → A is redundant in E′ and can be eliminated.
- Therefore, the minimal cover of E is {B → D, D → A}.

Next, we provide a simple algorithm to determine the key of a relation:

Algorithm 2a. Finding a Key K for R Given a set F of Functional Dependencies

Input: A relation R and a set of functional dependencies F on the attributes of R.

1. Set K := R.
2. For each attribute A in K
   {compute $(K – A)^+$ with respect to F;
   if $(K – A)^+$ contains all the attributes in R, then set K := K – {A} };

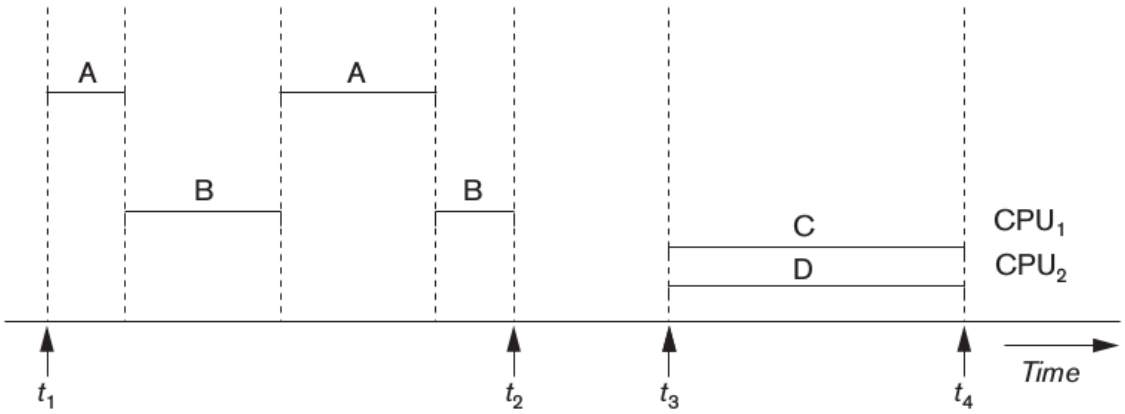
In Algorithm above, we start by setting K to all the attributes of R; we then remove one attribute at a time and check whether the remaining attributes still form a superkey. Notice, too, that Algorithm determines only one key out of the possible candidate keys for R; the key returned depends on the order in which attributes are removed from R in step 2.

Introduction to Transaction Processing

Single-User versus Multiuser Systems

A DBMS is single-user if at most one user at a time can use the system, and it is multiuser if many users can use the system—and hence access the database—concurrently. Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of travel agents and reservation clerks concurrently.

A single central processing unit (CPU) can only execute at most one process at a time. However, multiprogramming operating systems execute some commands from one process, then suspend that process and execute some commands from the next process, and so on. A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually interleaved, as illustrated in Figure below, which shows two processes, A and B, executing concurrently in an interleaved fashion. Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk.



If the computer system has multiple hardware processors (CPUs), parallel processing of multiple processes is possible, as illustrated by processes C and D in Figure above.

Transactions, Database Items, Read and Write Operations, and DBMS Buffers

A transaction includes one or more database access operations—these can include insertion, deletion, modification, or retrieval operations. One way of specifying the transaction boundaries is by specifying explicit begin transaction and end transaction statements in an application program; in this case, all database access operations between the two are considered as forming one transaction.

the basic database access operations that a transaction can include are as follows:

- read_item(X). Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
- write_item(X). Writes the value of program variable X into the database item named X.

Executing a read_item (X) command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the buffer to the program variable named X.

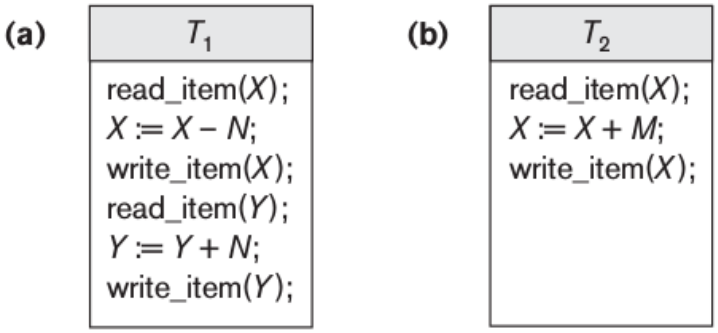
Executing a write_item (X) command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. 3. Copy item X from the program variable named X into its correct location in the buffer.
4. 4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Transactions submitted by the various users may execute concurrently and may access and update the same database items. If this concurrent execution is uncontrolled, it may lead to problems, such as an inconsistent database.

Why Concurrency Control Is Needed

Several problems can occur when concurrent transactions execute in an uncontrolled manner. We illustrate some of these problems by referring to a much simplified airline reservations database in which a record is stored for each airline flight. Each record includes the number of reserved seats on that flight as a named data item, among other information. Figure (a) below shows a transaction T_1 that transfers N reservations from one flight whose number of reserved seats is stored in the database item named X to another flight whose number of reserved seats is stored in the database item named Y. Figure (b) below shows a simpler transaction T_2 that just reserves M seats on the first flight (X) referenced in transaction T_1.



The Lost Update Problem. This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect. Suppose that transactions T_1 and T_2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure (a) below; then the final value of item X is incorrect because T_2 reads the value of X before T_1 changes it in the database, and hence the updated value resulting from T_1 is lost.

For example, if X = 80 at the start (originally there were 80 reservations on the flight), N = 5 (T_1 transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y), and M = 4 (T_2 reserves 4 seats on X), the final result should be X = 79. However, in the interleaving of operations shown in Figure (a) below, it is X = 84 because the update in T_1 that removed the five seats from X was lost.

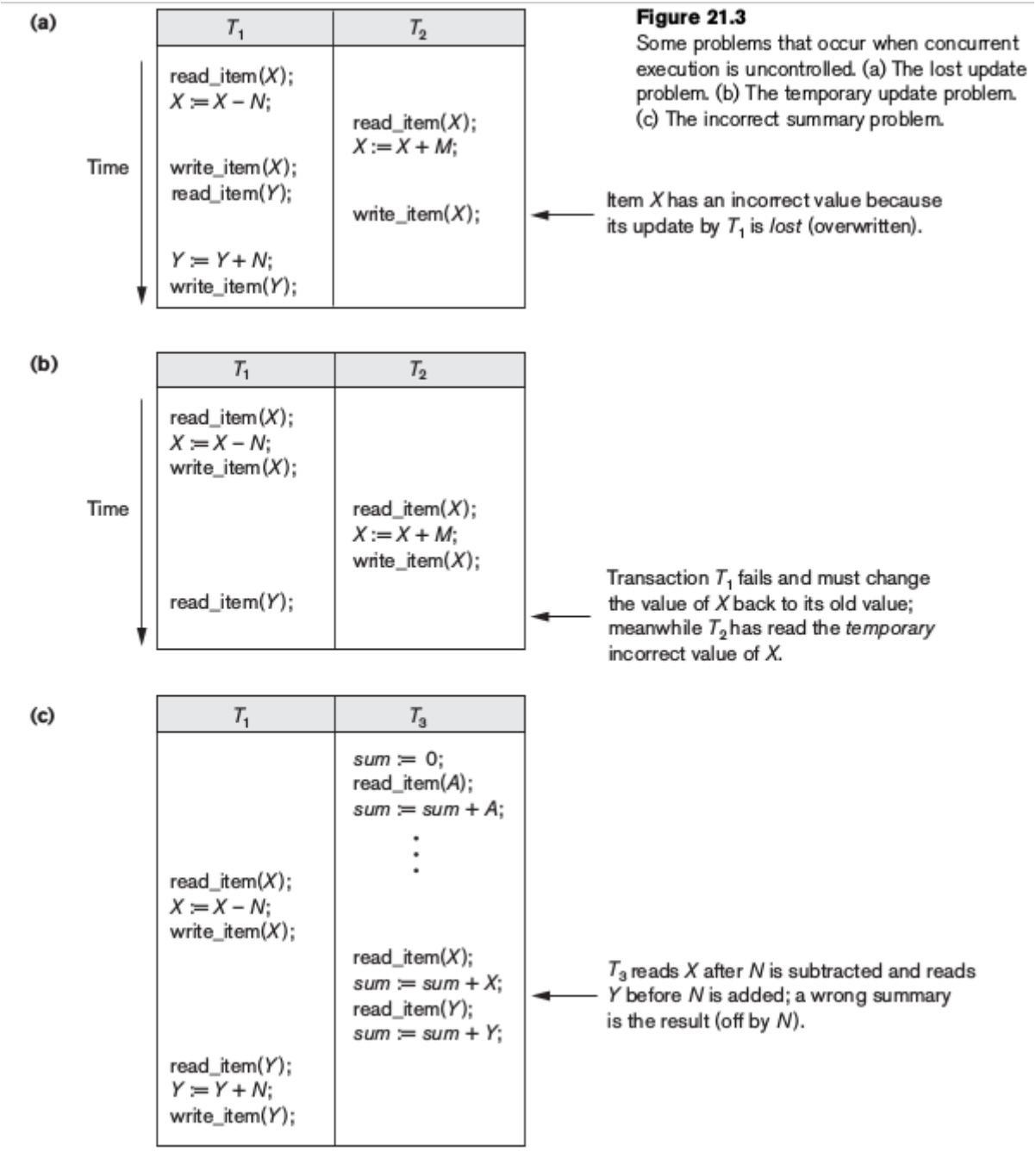


Figure 21.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

The Temporary Update (or Dirty Read) Problem. This problem occurs when one transaction updates a database item and then the transaction fails for some reason. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back to its

original value. Figure (b) above shows an example where T_1 updates item X and then fails before completion, so the system must change X back to its original value. Before it can do so, however, transaction T_2 reads the temporary value of X, which will not be recorded permanently in the database because of the failure of T_1. The value of item X that is read by T_2 is called dirty data because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the dirty read problem.

The Incorrect Summary Problem. If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction T_3 is calculating the total number of reservations on all the flights; meanwhile, transaction T_1 is executing. If the interleaving of operations shown in Figure (c) above occurs, the result of T_3 will be off by an amount N because T_3 reads the value of X after N seats have been subtracted from it but reads the value of Y before those N seats have been added to it.

The Unrepeatable Read Problem. Another problem that may occur is called unrepeatable read, where a transaction T reads the same item twice and the item is changed by another transaction T_J between the two reads. Hence, T receives different values for its two reads of the same item. This may occur, for example, if during an airline reservation transaction, a customer inquires about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

Why Recovery Is Needed

If a transaction fails after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

Types of Failures. Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

1. A computer failure (system crash). A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.
2. A transaction or system error. Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. Additionally, the user may interrupt the transaction during its execution.
3. Local errors or exception conditions detected by the transaction. During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. An exception condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled.
4. Concurrency control enforcement. The concurrency control method may decide to abort a transaction because it violates serializability, or it may abort one or more transactions to resolve a state of deadlock among several transactions.
5. Disk failure. Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash.
6. Physical problems and catastrophes. This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Desirable Properties of Transactions

Transactions should possess several properties, often called the ACID properties; they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

•

The atomicity property requires that we execute a transaction to completion. It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database. On the other hand, write operations of a committed transaction must be eventually written to disk.

A consistent state of the database satisfies the constraints specified in the schema as well as any other constraints on the database that should hold. A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the complete execution of the transaction, assuming that no interference with other transactions occurs.

The isolation property is enforced by the concurrency control subsystem of the DBMS. If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks but does not eliminate all other problems.

And last, the durability property is the responsibility of the recovery subsystem of the DBMS.

Characterizing Schedules Based on Serializability

Now we characterize the types of schedules that are always considered to be correct when concurrent transactions are executing. Such schedules are known as serializable schedules. Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions T_1 and T_2 at approximately the same time. If no interleaving of operations is permitted, there are only two possible outcomes:

1. Execute all the operations of transaction T_1(in sequence) followed by all the operations of transaction T_2 (in sequence).
2. Execute all the operations of transaction T_2(in sequence) followed by all the operations of transaction T_1(in sequence).

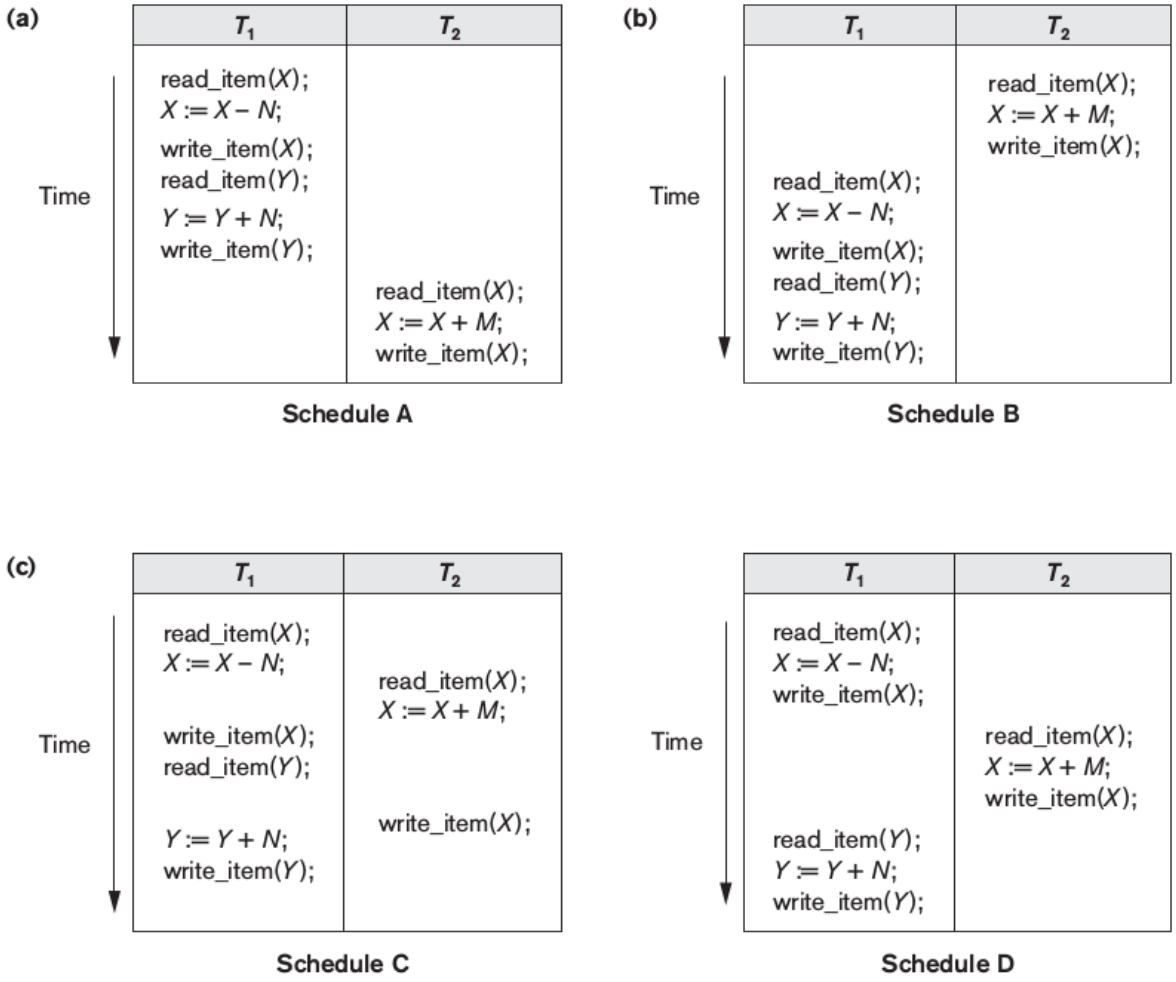
These two schedules—called serial schedules—are shown in Figure (a) and (b) below, respectively. If interleaving of operations is allowed, there will be many possible orders in which the system can execute the individual operations of the transactions. The concept of serializability of schedules is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules.

Serial, Nonserial, and Conflict-Serializable Schedules

Schedules A and B in Figure (a) and (b) below are called serial because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction. Schedules C and D in Figure (c) are called nonserial because each sequence interleaves operations from the two transactions. Formally, a schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule; otherwise, the schedule is called nonserial.

The problem with serial schedules is that they limit concurrency by prohibiting interleaving of operations. In a serial schedule, if a transaction waits for an I/O operation to

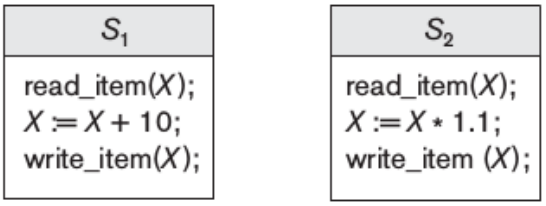
complete, we cannot switch the CPU to another transaction, thus wasting valuable CPU processing time. Hence, serial schedules are considered unacceptable in practice. However, if we can determine which other schedules are equivalent to a serial schedule, we can allow these schedules to occur.



To illustrate our discussion, consider the schedules in Figure above, and assume that the initial values of database items are X = 90 and Y = 90 and that N = 3 and M = 2. After executing transactions T_1 and T_2, we would expect the database values to be X = 89 and Y = 93, according to the meaning of the transactions. Executing either of the serial schedules A or B gives the correct results. Now consider the nonserial schedules C and D. Schedule C gives the results X = 92 and Y = 93, in which the X value is erroneous, whereas schedule D gives the correct results.

Schedule C gives an erroneous result because of the lost update problem; transaction T_2 reads the value of X before it is changed by transaction T_1, so only the effect of T_2 on X is reflected in the database. The effect of T_1 on X is lost, overwritten by T_2, leading to the incorrect result for item X. However, some nonserial schedules give the correct expected result, such as schedule D. We would like to determine which of the nonserial schedules always give a correct result and which may give erroneous results. The concept used to characterize schedules in this manner is that of **serializability** of a schedule.

The definition of serializable schedule is as follows: A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions.



Two schedules are called result equivalent if they produce the same final state of the database. However, two different schedules may accidentally produce the same final state. For example, in Figure above, schedules S_1 and S_2 will produce the same final database state if they execute on a database with an initial value of X = 100; however, for other initial values of X, the schedules are not result equivalent. Hence, result equivalence alone cannot be used to define equivalence of schedules.

Two definitions of equivalence of schedules are generally used: conflict equivalence and view equivalence. The definition of conflict equivalence of schedules is as follows: Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

For example, if a read and write operation occur in the order $r_1(X)$, $w_2(X)$ in schedule S_1, and in the reverse order $w_2(X)$, $r_1(X)$ in schedule S_2, the value read by $r_1(X)$ can be different in the two schedules. Similarly, if two write operations occur in the order $w_1(X)$, $w_2(X)$ in S_1, and in the reverse order $w_2(X)$, $w_1(X)$ in S_2, the next r(X) operation in the two schedules will read potentially different values; or if these are the last operations writing item X in the schedules, the final value of item X in the database will be different.

Using the notion of conflict equivalence, we define a schedule S to be conflict serializable if it is (conflict) equivalent to some serial schedule S_J. In such a case, we can reorder the nonconflicting operations in S until we form the equivalent serial schedule S_J.

According to this definition, schedule D in Figure (c) above is equivalent to the serial schedule A in Figure (a). In both schedules, the read_item (X) of T_2 reads the value of X written by T_1, while the other read_item operations read the database values from the initial database state. Additionally, T_1 is the last transaction to write Y, and T_2 is the last transaction to write X in both schedules. Because A is a serial schedule and schedule D is equivalent to A, D is a serializable schedule. Notice that the operations $r_1(Y)$ and $w_1(Y)$ of schedule D do not conflict with the operations $r_2(X)$ and $w_2(X)$, since they access different data items. Therefore, we can move $r_1(Y)$, $w_1(Y)$ before $r_2(X)$, $w_2(X)$, leading to the equivalent serial schedule T_1, T_2.

Schedule C in Figure (c) is not equivalent to either of the two possible serial schedules A and B, and hence is not serializable. Trying to reorder the operations of schedule C to find an equivalent serial schedule fails because $r_2(X)$ and $w_1(X)$ conflict, which means that we cannot move $r_2(X)$ down to get the equivalent serial schedule T_1, T_2. Similarly, because $w_1(X)$ and $w_2(X)$ conflict, we cannot move $w_1(X)$ down to get the equivalent serial schedule T_2, T_1.

Testing for Conflict Serializability of a Schedule

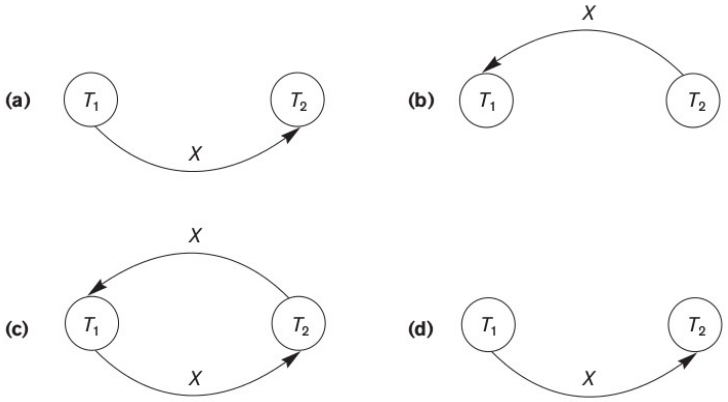
Algorithm can be used to test a schedule for conflict serializability. The algorithm looks at only the read_item and write_item operations in a schedule to construct a precedence graph (or serialization graph), which is a directed graph G = (N, E) that consists of a set of nodes N = {T_1, T_2, ..., T_n} and a set of directed edges E = {e_1, e_2, ..., e_m}.

Algorithm Testing Conflict Serializability of a Schedule S

1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
4. For each case in S where T_j executes a write_item (X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

In the precedence graph, an edge from T_i to T_j means that transaction T_i must come before transaction T_j in any serial schedule that is equivalent to S, because two conflicting operations appear in the schedule in that order. If there is no cycle in the precedence graph, we can create an equivalent serial schedule S_J that is equivalent to S, by ordering the transactions that participate in S as follows: Whenever an edge exists in the precedence graph from T_i to T_j, T_i must appear before T_j in the equivalent serial schedule S_J.

In general, several serial schedules can be equivalent to S if the precedence graph for S has no cycle. However, if the precedence graph has a cycle, it is easy to show that we cannot create any equivalent serial schedule, so S is not serializable. The precedence graphs created for schedules A to D, respectively, in Figure above appear in Figure (a) to (d) below. The graph for schedule C has a cycle, so it is not serializable. The graph for schedule D has no cycle, so it is serializable, and the equivalent serial schedule is T_1 followed by T_2. The graphs for schedules A and B have no cycles, as expected, because the schedules are serial and hence serializable.



(a)

Transaction T_1	Transaction T_2	Transaction T_3
read_item(X);	read_item(Z);	read_item(Y);
write_item(X);	read_item(Y);	read_item(Z);
read_item(Y);	write_item(Y);	write_item(Y);
write_item(Y);	read_item(X);	write_item(Z);
	write_item(X);	

(b)

Time

Transaction T_1	Transaction T_2	Transaction T_3
	read_item(Z);	
	read_item(Y);	
	write_item(Y);	
		read_item(Y);
		read_item(Z);
read_item(X);		
write_item(X);		
		write_item(Y);
		write_item(Z);
	read_item(X);	
read_item(Y);		
write_item(Y);		
	write_item(X);	

Schedule E

(c)

Time

Transaction T_1	Transaction T_2	Transaction T_3
		read_item(Y);
		read_item(Z);
read_item(X);		
write_item(X);		
		write_item(Y);
		write_item(Z);
	read_item(Z);	
read_item(Y);		
write_item(Y);		
	read_item(Y);	
	write_item(Y);	
	read_item(X);	
	write_item(X);	

Schedule F

Another example, in which three transactions participate, is shown in Figure above. Figure (a) shows the read_item and write_item operations in each transaction. Two schedules E and F for these transactions are shown in (b) and (c), respectively, and the precedence graphs for schedules E and F are shown in Figure (d) and (e) below. Schedule E is not serializable because the corresponding precedence graph has cycles. Schedule F is serializable, and the serial schedule equivalent to F is shown in Figure(e). Figure (f) below shows a precedence graph representing a schedule that has two equivalent serial schedules. To find an equivalent serial schedule, start with a node that does not have any incoming edges, and then make sure that the node order for every edge is not violated.

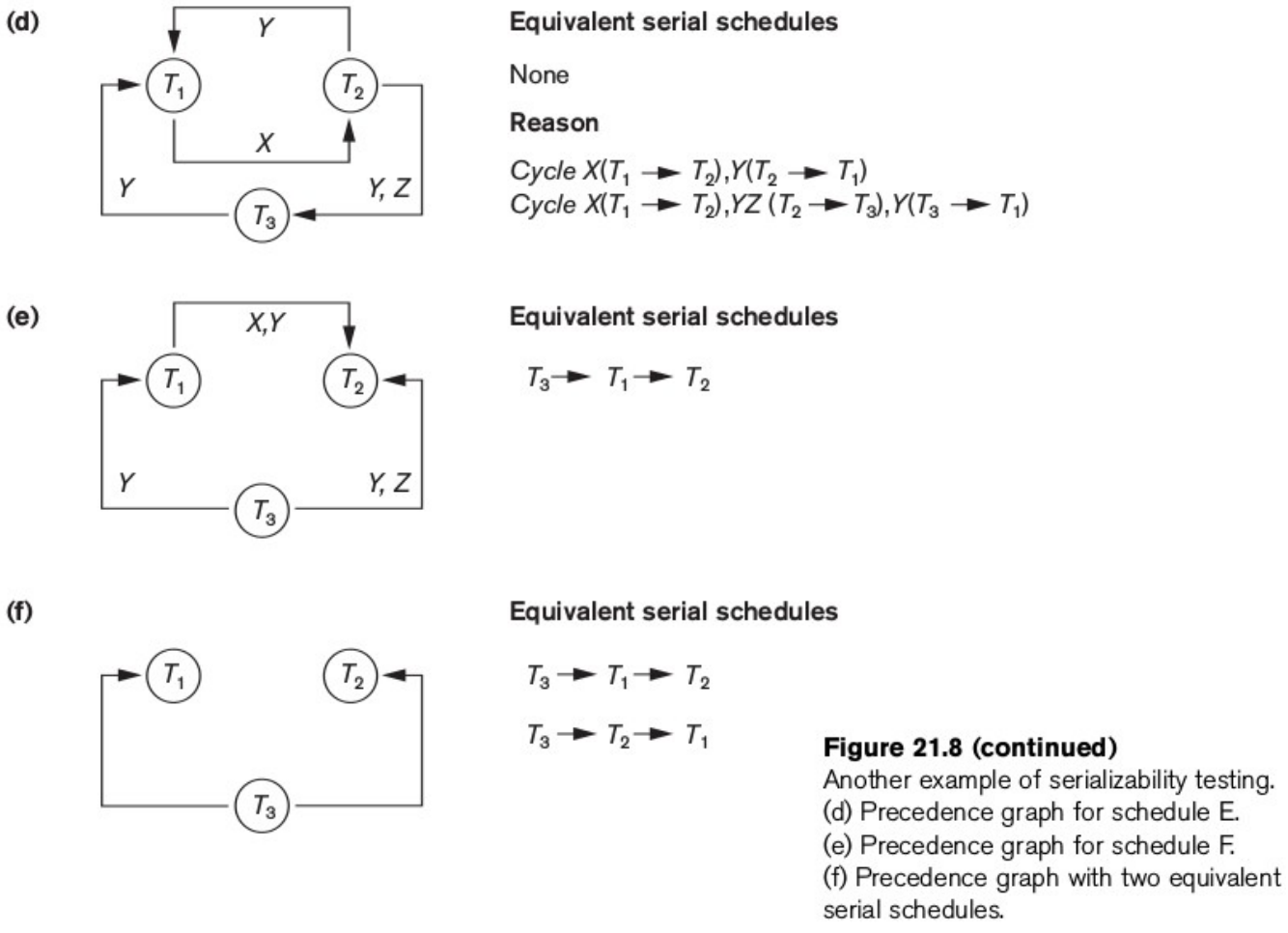


Figure 21.8 (continued)
Another example of serializability testing.
(d) Precedence graph for schedule E.
(e) Precedence graph for schedule F.
(f) Precedence graph with two equivalent serial schedules.

How Serializability Is Used for Concurrency Control

As we discussed earlier, saying that a schedule S is (conflict) serializable—that is, S is (conflict) equivalent to a serial schedule—is equivalent to saying that S is correct. Being serializable is distinct from being serial, however. A serializable schedule gives the benefits of concurrent execution without giving up any correctness.

When transactions are submitted continuously to the system, it is difficult to determine when a schedule begins and when it ends. Serializability theory can be adapted to deal with this problem by considering only the committed projection of a schedule S. Committed projection C(S) of a schedule S includes only the operations in S that belong to committed transactions. We can theoretically define a schedule S to be serializable if its committed projection C(S) is equivalent to some serial schedule, since only committed transactions are guaranteed by the DBMS.

View Equivalence and View Serializability

Another less restrictive definition of equivalence of schedules is called view equivalence. This

leads to another definition of serializability called view serializability. Two schedules S and S_J are said to be view equivalent if the following three conditions hold:

1. The same set of transactions participates in S and S_J, and S and S_J include the same operations of those transactions.
2. For any operation $r_i(X)$ of T_i in S, if the value of X read by the operation has been written by an operation $w_j(X)$ of T_j(or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation $r_i(X)$ of T_i in S_J.
3. If the operation $w_k(Y)$ of T_k is the last operation to write item Y in S, then $w_k(Y)$ of T_k must also be the last operation to write item Y in S_J.

The idea behind view equivalence is that, as long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results. The read operations are hence said to see the same view in both schedules. Condition 3 ensures that the final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules. A schedule S is said to be view serializable if it is view equivalent to a serial schedule.

The definitions of conflict serializability and view serializability are similar if a condition known as the constrained write assumption (or no blind writes) holds on all transactions in the schedule. This condition states that any write operation $w_i(X)$ in T_i is preceded by a $r_i(X)$ in T_i and that the value written by $w_i(X)$ in T_i depends only on the value of X read by $r_i(X)$. This assumes that computation of the new value of X is a function f(X) based on the old value of X read from the database. A blind write is a write operation in a transaction T on an item X that is not dependent on the value of X, so it is not preceded by a read of X in the transaction T.

The definition of view serializability is less restrictive than that of conflict serializability under the unconstrained write assumption, where the value written by an operation $w_i(X)$ in T_i can be independent of its old value from the database. This is possible when blind writes are allowed, and it is illustrated by the following schedule S_g of three transactions T_1: $r_1(X)$; $w_1(X)$; T_2: $w_2(X)$; and T_3: $w_3(X)$:

S_g: $r_1(X)$; $w_2(X)$; $w_1(X)$; $w_3(X)$; c_1; c_2; c_3;

In S_g the operations $w_2(X)$ and $w_3(X)$ are blind writes, since T_2 and T_3 do not read the value of X. The schedule S_g is view serializable, since it is view equivalent to the serial schedule T_1, T_2, T_3. However, S_g is not conflict serializable, since it is not conflict equivalent to any serial schedule. It has been shown that any conflict serializable schedule is also view serializable but not vice versa, as illustrated by the preceding example.

Other Types of Equivalence of Schedules

Serializability of schedules is sometimes considered to be too restrictive as a condition for ensuring the correctness of concurrent executions. Some applications can produce schedules that are correct by satisfying conditions less stringent than either conflict serializability or view serializability. An example is the type of transactions known as debit-credit transactions—for example, those that apply deposits and withdrawals to a data item whose value is the current balance of a bank account. The semantics of debit-credit operations is that they update the value of a data item X by either subtracting from or adding to the value of the data item. Because addition and subtraction operations are commutative—that is, they can be applied in any order—it is possible to produce correct schedules that are not serializable. For example, consider the following transactions, each of which may be used to

transfer an amount of money between two bank accounts:

T_1 : $r_1(X)$; X := X – 10; $w_1(X)$; $r_1(Y)$; Y := Y + 10; $w_1(Y)$;

T_2 : $r_2(Y)$; Y := Y – 20; $w_2(Y)$; $r_2(X)$; X := X + 20; $w_2(X)$;

Consider the following nonserializable schedule S_h for the two transactions: S_h: $r_1(X)$; $w_1(X)$; $r_2(Y)$; $w_2(Y)$; $r_1(Y)$; $w_1(Y)$; $r_2(X)$; $w_2(X)$;

With the additional knowledge, or semantics, that the operations between each $r_i(I)$ and $w_i(I)$ are commutative, we know that the order of executing the sequences consisting of (read, update, write) is not important as long as each (read, update, write) sequence by a particular transaction T_i on a particular item I is not interrupted by conflicting operations. Hence, the schedule S_h is considered to be correct even though it is not serializable.

Two-Phase Locking Techniques for Concurrency Control

Types of Locks and System Lock Tables

To introduce locking concepts gradually, first we discuss binary locks, which are simple, but are also too restrictive for database concurrency control purposes, and so are not used in practice. Then we discuss shared/exclusive locks—also known as read/write locks—which provide more general locking capabilities and are used in practical database locking schemes.

Binary Locks. A binary lock can have two states or values: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1. We refer to the current value (or state) of the lock associated with item X as lock(X). Two operations, lock_item and unlock_item, are used with binary locking. A transaction requests access to an item X by first issuing a lock_item(X) operation. If LOCK (X) = 1, the transaction is forced to wait. If LOCK (X) = 0, it is set to 1 (the transaction locks the item) and the transaction is allowed to access item X. When the transaction is through using the item, it issues an unlock_item(X) operation, which sets LOCK(X) back to 0 (unlocks the item) so that X may be accessed by other transactions. Hence, a binary lock enforces mutual exclusion on the data item.

lock_item(X):

```
B:  if LOCK(X) = 0              (* item is unlocked *)
        then LOCK(X) ← 1        (* lock the item *)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
```

unlock_item(X):

```
    LOCK(X) ← 0;        (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

Notice that the lock_item and unlock_item operations must be implemented as indivisible units; that is, no interleaving should be allowed once a lock or unlock operation is started

until the operation terminates or the transaction waits. It is quite simple to implement a binary lock; all that is needed is a binary-valued variable, LOCK , associated with each data item X in the database. In its simplest form, each lock can be a record with three fields: <Data_item_name, LOCK, Locking_transaction> plus a queue for transactions that are waiting to access the item.

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

1. A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.
2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.
3. A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X.
4. A transaction T will not issue an unlock_item (X) operation unless it already holds the lock on item X.

Shared/Exclusive (or Read/Write) Locks. The preceding binary locking scheme is too restrictive for database items because at most, one transaction can hold a lock on a given item. We should allow several transactions to access the same item X if they all access X for reading purposes only. This is because read operations on the same item by different transactions are not conflicting. However, if a transaction is to write an item X, it must have exclusive access to X.

For this purpose, a different type of lock called a multiple-mode lock is used. In this scheme—called shared/exclusive or read/write locks—there are three locking operations: read_lock(X), write_lock(X), and unlock(X). A lock associated with an item X, LOCK(X), now has three possible states: read-locked, write-locked, or unlocked. A read-locked item is also called share-locked because other transactions are allowed to read the item, whereas a write-locked item is called exclusive-locked because a single transaction exclusively holds the lock on the item.

Each record in the lock table will have four fields: <Data_item_name, LOCK, No_of_reads, Locking_transaction(s)>. Again, to save space, the system needs to maintain lock records only for locked items in the lock table. The value (state) of LOCK is either read-locked or write-locked, suitably coded (if we assume no records are kept in the lock table for unlocked items). If LOCK (X)=write-locked, the value of locking_transaction(s) is a single transaction that holds the exclusive (write) lock on X. If LOCK (X)=read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on X. The three operations read_lock(X), write_lock(X), and unlock(X) are described below, each of the three locking operations should be considered indivisible; no interleaving should be allowed once one of the operations is started until either the operation terminates by granting the lock or the transaction is placed in a waiting queue for the item.

read_lock(X):

```
B:    if LOCK(X) = "unlocked"
            then begin LOCK(X) ← "read-locked";
                  no_of_reads(X) ← 1
                  end
      else if LOCK(X) = "read-locked"
            then no_of_reads(X) ← no_of_reads(X) + 1
```

```
        else begin
                wait (until LOCK(X) = "unlocked"
                        and the lock manager wakes up the transaction);
                go to B
                end;
write_lock(X):
B:      if LOCK(X) = "unlocked"
                then LOCK(X) ← "write-locked"
        else begin
                wait (until LOCK(X) = "unlocked"
                        and the lock manager wakes up the transaction);
        go to B
        end;
unlock (X):
        if LOCK(X) = "write-locked"
                then begin LOCK(X) ← "unlocked";
                        wakeup one of the waiting transactions, if any
                        end
        else if LOCK(X) = "read-locked"
                then begin
                        no_of_reads(X) ← no_of_reads(X) –1;
                        if no_of_reads(X) = 0
                                then begin LOCK(X) = "unlocked";
                                        wakeup one of the waiting transactions, if any
                                        end
                        end;
```

When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
2. A transaction T must issue the operation write_lock (X) before any write_item(X) operation is performed in T.
3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item (X) operations are completed in T.
4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.
5. A transaction T will not issue a write_lock(X) operation if it already holds a read (shared)lock or write(exclusive) lock on item X.
6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Conversion of Locks. Sometimes it is desirable to relax conditions 4 and 5 in the preceding list in order to allow lock conversion; that is, a transaction that already holds a lock on item X is allowed under certain conditions to convert the lock from one locked state to another. For example, it is possible for a transaction T to issue a read_lock(X) and then later to upgrade

the lock by issuing a write_lock(X) operation. If T is the only transaction holding a read lock on X at the time it issues the write_lock(X) operation, the lock can be upgraded; otherwise, the transaction must wait. It is also possible for a transaction T to issue a write_lock(X) and then later to downgrade the lock by issuing a read_lock(X) operation. When upgrading and downgrading of locks is used, the lock table must include transaction identifiers in the record structure for each lock to store the information on which transactions hold locks on the item.

Using binary locks or read/write locks in transactions, as described earlier, does not guarantee serializability of schedules on its own. Figure 22.3 shows an example where the preceding locking rules are followed but a nonserializable schedule may result. This is because in Figure 22.3(a) the items Y in T_1 and X in T_2 were unlocked too early. This allows a schedule such as the one shown in Figure 22.3(c) to occur, which is not a serializable schedule and hence gives incorrect results. To guarantee serializability, we must follow an additional protocol concerning the positioning of locking and unlocking operations in every transaction. The best-known protocol, two-phase locking, is described in the next section.

Guaranteeing Serializability by Two-Phase Locking

A transaction is said to follow the two-phase locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction. Such a transaction can be divided into two phases: an expanding or growing (first) phase, during which new locks on items can be acquired but none can be released; and a shrinking (second) phase, during which existing locks can be released but no new locks can be acquired. If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase. Hence, a read_lock(X) operation that downgrades an already held write lock on X can appear only in the shrinking phase. Transactions T_1 and T_2 in Figure 22.3(a) do not follow the two-phase locking protocol because the write_lock(X) operation follows the unlock(Y) operation in T_1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T_2. If we enforce two-phase locking, the transactions can be rewritten as T_1' and T_2', as shown in Figure 22.4.

(a)

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
X := X + Y;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	Y := X + Y;
	write_item(Y);
	unlock(Y);

(b) Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

(c)

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);	

	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	Y := X + Y;
	write_item(Y);
	unlock(Y);

write_lock(X);
read_item(X);
X := X + Y;
write_item(X);
unlock(X);	

Time

Result of schedule S:
X=50, Y=50
(nonserializable)

Figure 22.3
Transactions that do not obey two-phase locking. (a) Two transactions T_1 and T_2. (b) Results of possible serial schedules of T_1 and T_2. (c) A nonserializable schedule S that uses locks.

T_1'	T_2'

read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
read_item(X);
X := X + Y;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	write_lock(Y);
	unlock(X)
	read_item(Y);
	Y := X + Y;
	write_item(Y);
	unlock(Y);

Figure 22.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 22.3, but follow the two-phase locking protocol. Note that they can produce a deadlock.

Now, the schedule shown in Figure 22.3(c) is not permitted for T_1' and T_2' (with their modified order of locking and unlocking operations)

It can be proved that, if every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable. The locking protocol, by enforcing two-phase locking rules, also enforces serializability. Two-phase locking may limit the amount of concurrency that can occur in a schedule, X must remain locked by T until all items that the

transaction needs to read or write have been locked; only then can X be released by T. This is the price for guaranteeing serializability of all schedules without having to check the schedules themselves.

Basic, Conservative, Strict, and Rigorous Two-Phase Locking. There are a number of variations of two-phase locking (2PL). The technique just described is known as basic 2PL. A variation known as conservative 2PL (or static 2PL) requires a transaction to lock all the items it accesses before the transaction begins execution, by predeclaring its read-set and write-set. If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking. Conservative 2PL is a deadlock-free protocol. However, it is difficult to use in practice because of the need to predeclare the read-set and write-set, which is not possible in many situations.

The most popular variation of 2PL is strict 2PL, which guarantees strict schedules. In this variation, a transaction T does not release any of its exclusive(write) locks until after it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability. Strict 2PL is not deadlock-free.

A more restrictive variation of strict 2PL is rigorous 2PL, which also guarantees strict schedules. In this variation, a transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL.

Dealing with Deadlock and Starvation

Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T_J in the set. Hence, each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item. But because the other transaction is also waiting, it will never release the lock.

A simple example is shown in Figure 22.5(a), where the two transactions $T_1'$and T_2' are deadlocked in a partial schedule; T_1' is in the waiting queue for X, which is locked by T_2', while T_2' is in the waiting queue for Y, which is locked by T_1'. Meanwhile, neither T_1' nor T_2' nor any other transaction can access items X and Y.

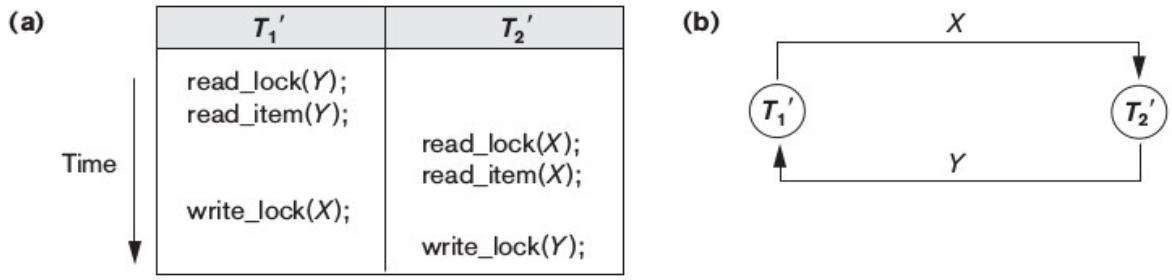


Figure 22.5
Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

Deadlock Prevention Protocols. One way to prevent deadlock is to use a deadlock prevention protocol. One deadlock prevention protocol, which is used in conservative two-phase locking, requires that every transaction lock all the items it needs in advance (which is generally not a practical assumption)—if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. Obviously this solution further limits concurrency.

A number of other deadlock prevention schemes have been proposed that make a decision about what to do with a transaction involved in a possible deadlock situation. Some

of these techniques use the concept of transaction timestamp TS(T), which is a unique identifier assigned to each transaction. The timestamps are typically based on the order in which transactions are started; hence, if transaction T_1 starts before transaction T_2, then $TS(T_1) < TS(T_2)$. Notice that the older transaction (which starts first) has the smaller timestamp value. Two schemes that prevent deadlock are called wait-die and wound- wait. Suppose that transaction T_i tries to lock an item X but is not able to because X is locked by some other transaction T_j with a conflicting lock. The rules followed by these schemes are:

- **Wait-die**. If $TS(T_i) < TS(T_j)$, then (T_i older than T_j) T_i is allowed to wait; otherwise (T_i younger than T_j) abort T_i (T_i dies) and restart it later with the same timestamp.
- **Wound-wait.** If $TS(T_i) < TS(T_j)$, then (T_i older than T_j) abort T_j (T_i wounds T_j) and restart it later with the same timestamp; otherwise (T_i younger than T_j) T_i is allowed to wait.

In wait-die, an older transaction is allowed to wait for a younger transaction, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted. The wound-wait approach does the opposite: A younger transaction is allowed to wait for an older one, whereas an older transaction requesting an item held by a younger transaction preempts the younger transaction by aborting it. Both schemes end up aborting the younger of the two transactions (the transaction that started later) that may be involved in a deadlock, assuming that this will waste less processing.

It can be shown that these two techniques are deadlock-free, since in wait-die, transactions only wait for younger transactions so no cycle is created. Similarly, in wound-wait, transactions only wait for older transactions so no cycle is created.

Another group of protocols that prevent deadlock do not require timestamps. These include the no waiting (NW) and cautious waiting (CW) algorithms. In the no waiting algorithm, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not. In this case, no transaction ever waits, so no deadlock will occur. However, this scheme can cause transactions to abort and restart needlessly. The cautious waiting algorithm was proposed to try to reduce the number of needless aborts/restarts. Suppose that transaction T_i tries to lock an item X but is not able to do so because X is locked by some other transaction T_j with a conflicting lock. The cautious waiting rules are as follows:

- Cautious waiting. If T_j is not blocked (not waiting for some other locked item), then T_i is blocked and allowed to wait; otherwise abort T_i.

It can be shown that cautious waiting is deadlock-free, because no transaction will ever wait for another blocked transaction. By considering the time b(T) at which each blocked transaction T was blocked, if the two transactions T_i and T_j above both become blocked, and T_i is waiting for T_j, then $b(T_i) < b(T_j)$, since T_i can only wait for T_j at a time when T_j is not blocked itself. Hence, the blocking times form a total ordering on all blocked transactions, so no cycle that causes deadlock can occur.

Deadlock Detection. A second, more practical approach to dealing with deadlock is deadlock detection, where the system checks if a state of deadlock actually exists. A simple way to detect a state of deadlock is for the system to construct and maintain a wait-for graph. One node is created in the wait-for graph for each transaction that is currently executing. Whenever a transaction T_i is waiting to lock an item X that is currently locked by a transaction T_j, a directed edge ($T_i \rightarrow T_j$) is created in the wait-for graph. We have a state of deadlock if and only if the wait-for graph has a cycle. One problem with this approach is the matter of determining when the system should check for a deadlock. One possibility is to check for a cycle every time an edge is added to the wait-for graph, but this may cause excessive overhead.

Figure 22.5(b) shows the wait-for graph for the (partial) schedule shown in Figure 22.5(a). If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted. Choosing which transactions to abort is known as victim selection. The algorithm for victim selection should generally avoid selecting transactions that have been running for a long time and that have performed many updates, and it should try instead to select transactions that have not made many changes (younger transactions).

Timeouts. Another simple scheme to deal with deadlock is the use of timeouts. This method is practical because of its low overhead and simplicity. In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists or not.

Starvation. Another problem that may occur when we use locking is starvation, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others. One solution for starvation is to have a fair waiting scheme, such as using a first-come-first-served queue; transactions are enabled to lock an item in the order in which they originally requested the lock. Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds. Starvation can also occur because of victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution.