CHAPTER 14: Basics of Functional Dependencies and Normalization for Relational Databases

Answers to Selected Exercises

14.19 - Suppose we have the following requirements for a university database that is used to keep track of students' transcripts:

(a) The university keeps track of each student's name (SNAME), student number (SNUM), social security number (SSSN), current address (SCADDR) and phone (SCPHONE), permanent address (SPADDR) and phone (SPPHONE), birthdate (BDATE), sex (SEX), class (CLASS) (freshman, sophomore, ..., graduate), major department (MAJORDEPTCODE), minor department (MINORDEPTCODE) (if any), and degree program (PROG) (B.A., B.S., ..., Ph.D.). Both ssn and student number have unique values for each student.

(b) Each department is described by a name (DEPTNAME), department code (DEPTCODE), office number (DEPTOFFICE), office phone (DEPTPHONE), and college (DEPTCOLLEGE). Both name and code have unique values for each department.

(c) Each course has a course name (CNAME), description (CDESC), code number (CNUM), number of semester hours (CREDIT), level (LEVEL), and offering department (CDEPT). The value of code number is unique for each course.

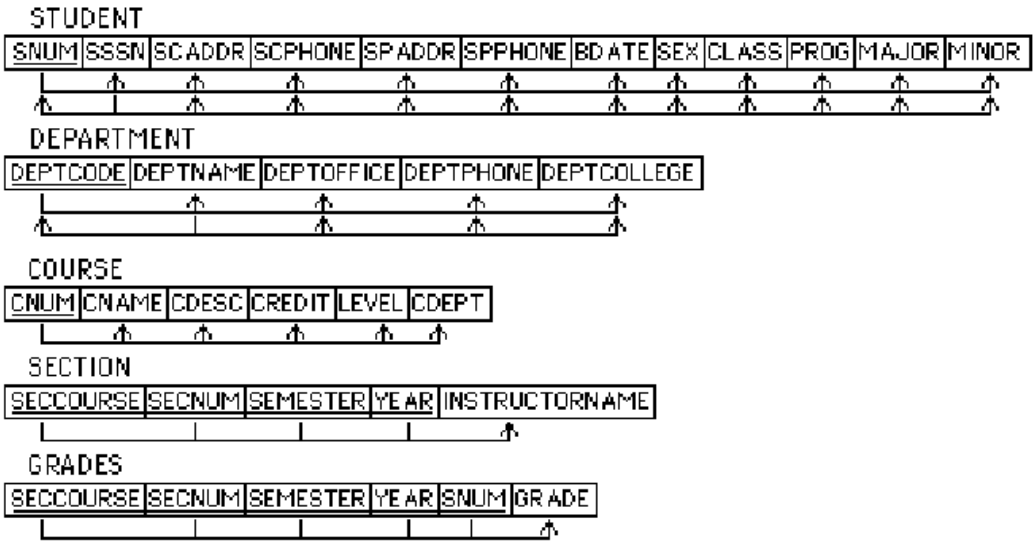
(d) Each section has an instructor (INSTUCTORNAME), semester (SEMESTER), year (YEAR), course (SECCOURSE), and section number (SECNUM). Section numbers distinguish different sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, ...; up to the number of sections taught during each semester.

(e) A grade record refers to a student (Ssn), refers to a particular section, and grade (GRADE).

Design an relational database schema for this database application. First show all the functional dependencies that should hold among the attributes. Then, design relation schemas for the database that are each in 3NF or BCNF. Specify the key attributes of each relation. Note any unspecified requirements, and make appropriate assumptions to make the specification complete.

Answer:
From the above description, we can presume that the following functional dependencies hold on the attributes:
FD1: {SSSN} -> {SNAME, SNUM, SCADDR, SCPHONE, SPADDR, SPPHONE, BDATE, SEX, CLASS,
MAJOR, MINOR, PROG}
FD2: {SNUM} -> {SNAME, SSSN, SCADDR, SCPHONE, SPADDR, SPPHONE, BDATE, SEX, CLASS,
MAJOR, MINOR, PROG}
FD3: {DEPTNAME} -> {DEPTCODE, DEPTOFFICE, DEPTPHONE, DEPTCOLLEGE}
FD4: {DEPTCODE} -> {DEPTNAME, DEPTOFFICE, DEPTPHONE, DEPTCOLLEGE}
FD5: {CNUM} -> {CNAME, CDESC, CREDIT, LEVEL, CDEPT}
FD6: {SECCOURSE, SEMESTER, YEAR, SECNUM} -> {INSTRUCTORNAME}

FD7: {SECCOURSE, SEMESTER, YEAR, SECNUM, SSSN} -> {GRADE}
These are the basic FDs that we can define from the given requirements; using inference rules IR1 to IR3, we can deduce many others. FD1 and FD2 refer to student attributes; we can define a relation STUDENT and choose either SSSN or SNUM as its primary key. Similarly, FD3 and FD4 refer to department attributes, with either DEPTNAME or DEPTCODE as primary key. FD5 defines COURSE attributes, and FD6 SECTION attributes. Finally, FD7 defines GRADES attributes. We can create one relation for each of STUDENT, DEPARTMENT, COURSE, SECTION, and GRADES as shown below, where the primary keys are underlined. The COURSE, SECTION, and GRADES relations are in 3NF and BCNF if no other dependencies exist. The STUDENT and DEPARTMENT relations are in 3NF and BCNF according to the general definition given in Sections 18.4 and 18.5, but not according to the definitions of Section 18.3 since both relations have secondary keys.



The foreign keys will be as follows:
STUDENT.MAJOR -> DEPARTMENT.DEPTCODE
STUDENT.MINOR -> DEPARTMENT.DEPTCODE
COURSE.CDEPT -> DEPARTMENT.DEPTCODE
SECTION.SECCOURSE -> COURSE.CNUM
GRADES.(SECCOURSE, SEMESTER, YEAR, SECNUM) ->
SECTION.(SECCOURSE, SEMESTER, YEAR, SECNUM)
GRADES.SNUM -> STUDENT.SNUM
Note: We arbitrarily chose SNUM over SSSN for primary key of STUDENT, and DEPTCODE over DEPTNAME for primary key of DEPARTMENT.

14.20 - What update anomalies occur in the EMP_PROJ and EMP_DEPT relations of Figure 14.3 and 14.4?

Answer:
In EMP_PROJ, the partial dependencies {SSN}->{ENAME} and {PNUMBER}->{PNAME, PLOCATION} can cause anomalies. For example, if a PROJECT temporarily has no

EMPLOYEEs working on it, its information (PNAME, PNUMBER, PLOCATION) will not be represented in the database when the last EMPLOYEE working on it is removed (deletion anomaly). A new PROJECT cannot be added unless at least one EMPLOYEE is assigned to work on it (insertion anomaly). Inserting a new tuple relating an existing EMPLOYEE to an existing PROJECT requires checking both partial dependencies; for example, if a different value is entered for PLOCATION than those values in other tuples with the same value for PNUMBER, we get an update anomaly. Similar comments apply to EMPLOYEE information. The reason is that EMP_PROJ represents the relationship between EMPLOYEEs and PROJECTs, and at the same time represents information concerning EMPLOYEE and PROJECT entities.
In EMP_DEPT, the transitive dependency {SSN}->{DNUMBER}->{DNAME, DMGRSSN} can cause anomalies. For example, if a DEPARTMENT temporarily has no EMPLOYEEs working for it, its information (DNAME, DNUMBER, DMGRSSN) will not be represented in the database when the last EMPLOYEE working on it is removed (deletion anomaly). A new DEPARTMENT cannot be added unless at least one EMPLOYEE is assigned to work on it
(insertion anomaly). Inserting a new tuple relating a new EMPLOYEE to an existing DEPARTMENT requires checking the transitive dependencies; for example, if a different value is entered for DMGRSSN than those values in other tuples with the same value for DNUMBER, we get an update anomaly. The reason is that EMP_DEPT represents the relationship between EMPLOYEEs and DEPARTMENTs, and at the same time represents information concerning EMPLOYEE and DEPARTMENT entities.

14.21 - In what normal form is the LOTS relation schema in Figure 14.12(a) with respect to the restrictive interpretations of normal form that take only the *primary key* into account? Would it be in the same normal form if the general definitions of normal form were used?

Answer:
If we only take the primary key into account, the LOTS relation schema in Figure 14.11 (a) will be in 2NF since there are no partial dependencies on the primary key .
However, it is not in 3NF, since there are the following two transitive dependencies on the primary key:
PROPERTY_ID# ->COUNTY_NAME ->TAX_RATE, and
PROPERTY_ID# ->AREA ->PRICE.
Now, if we take all keys into account and use the general definition of 2NF and 3NF, the LOTS relation schema will only be in 1NF because there is a partial dependency COUNTY_NAME ->TAX_RATE on the secondary key {COUNTY_NAME, LOT#}, which violates 2NF.

14.22 - Prove that any relation schema with two attributes is in BCNF.

Answer:
Consider a relation schema R={A, B} with two attributes. The only possible (non-trivial) FDs are {A} ->{B} and {B} ->{A}. There are four possible cases:
(i) No FD holds in R. In this case, the key is {A, B} and the relation satisfies BCNF.
(ii) Only {A} ->{B} holds. In this case, the key is {A} and the relation satisfies BCNF.
(iii) Only {B} ->{A} holds. In this case, the key is {B} and the relation satisfies BCNF.
(iv) Both {A} ->{B} and {B} ->{A} hold. In this case, there are two keys {A} and {B} and the relation satisfies BCNF.
Hence, any relation with two attributes is in BCNF.

14.23 - Why do spurious tuples occur in the result of joining the EMP_PROJ1 and EMPLOCS relations of Figure 14.5 (result shown in Figure 14.6)?

Answer:
In EMP_LOCS, a tuple (e, l) signifies that employee with name e works on some project located in location l. In EMP_PROJ1, a tuple (s, p, h, pn, l) signifies that employee with social security number s works on project p that is located at location l. When we join EMP_LOCS with EMP_PROJ1, a tuple (e, l) in EMP_LOCS can be joined with a tuple (s, p, h, pn, l) in EMP_PROJ1 where e is the name of some employee and s is the social security number of a different employee, resulting in spurious tuples. The lossless join property (see Chapter 13) can determine whether or not spurious tuples may result based on the FDs in the two relations being joined.

14.24 - Consider the universal relation R = {A, B, C, D, E, F, G, H, I} and the set of functional dependencies F = { {A, B} -> {C}, {A} -> {D, E}, {B} -> {F}, {F} -> {G, H}, {D} -> {I, J} }. What is the key for R? Decompose R into 2NF, then 3NF relations.

Answer:
A minimal set of attributes whose closure includes all the attributes in R is a key. (One can also apply algorithm 14.4a (see chapter 14 in the textbook)). Since the closure of {A, B}, {A, B}+ = R, one key of R is {A, B} (in this case, it is the only key).
To normalize R intuitively into 2NF then 3NF, we take the following steps (alternatively, we can apply the algorithms discussed in Chapter 14):
First, identify partial dependencies that violate 2NF. These are attributes that are functionally dependent on either parts of the key, {A} or {B}, alone. We can calculate the closures {A}+ and {B}+ to determine partially dependent attributes:
{A}+ = {A, D, E, I, J}. Hence {A} -> {D, E, I, J} ({A} -> {A} is a trivial dependency)
{B}+ = {B, F, G, H}, hence {A} -> {F, G, H} ({B} -> {B} is a trivial dependency)
To normalize into 2NF, we remove the attributes that are functionally dependent on part of the key (A or B) from R and place them in separate relations R1 and R2, along with the part of the key they depend on (A or B), which are copied into each of these relations but also remains in the original relation, which we call R3 below:
R1 = {A, D, E, I, J}, R2 = {B, F, G, H}, R3 = {A, B, C}
The new keys for R1, R2, R3 are underlined. Next, we look for transitive dependencies in R1, R2, R3. The relation R1 has the transitive dependency {A} -> {D} -> {I, J}, so we remove the transitively dependent attributes {I, J} from R1 into a relation R11 and copy the attribute D they are dependent on into R11. The remaining attributes are kept in a relation R12. Hence, R1 is decomposed into R11 and R12 as follows:
R11 = {D, I, J}, R12 = {A, D, E}
The relation R2 is similarly decomposed into R21 and R22 based on the transitive dependency {B} -> {F} -> {G, H}:
R2 = {F, G, H}, R2 = {B, F}
The final set of relations in 3NF are {R11, R12, R21, R22, R3}

14.25 - Repeat exercise 14.24 for the following different set of functional dependencies G = { {A, B} -> {C}, {B, D} -> {E, F}, {A, D} -> {G, H}, {A} -> {I}, {H} -> {J} }.

Answer:
To help in solving this problem systematically, we can first find the closures of all single attributes to see if any is a key on its own as follows:
{A}+ -> {A, I}, {B}+ -> {B}, {C}+ -> {C}, {D}+ -> {D}, {E}+ -> {E}, {F}+ -> {F},
{G}+ -> {G}, {H}+ -> {H, J}, {I}+ -> {I}, {J}+ -> {J}
Since none of the single attributes is a key, we next calculate the closures of pairs of attributes that are possible keys:
{A, B}+ -> {A, B, C, I}, {B, D}+ -> {B, D, E, F}, {A, D}+ -> {A, D, G, H, I, J}
None of these pairs are keys either since none of the closures includes all attributes. But the union of the three closures includes all the attributes:
{A, B, D}+ -> {A, B, C, D, E, F, G, H, I}
Hence, {A, B, D} is a key. (Note: Algorithm 14.4a (see chapter 14 in the textbook) can be used to determine a key).
Based on the above analysis, we decompose as follows, in a similar manner to problem 14.26, starting with the following relation R:
R = {A, B, D, C, E, F, G, H, I}
The first-level partial dependencies on the key (which violate 2NF) are:
{A, B} -> {C, I}, {B, D} -> {E, F}, {A, D}+ -> {G, H, I, J}
Hence, R is decomposed into R1, R2, R3, R4 (keys are underlined):
R1 = {A, B, C, I}, R2 = {B, D, E, F}, R3 = {A, D, G, H, I, J}, R4 = {A, B, D}
Additional partial dependencies exist in R1 and R3 because {A} -> {I}. Hence, we remove {I} into R5, so the following relations are the result of 2NF decomposition:
R1 = {A, B, C}, R2 = {B, D, E, F}, R3 = {A, D, G, H, J}, R4 = {A, B, D}, R5 = {A, I}
Next, we check for transitive dependencies in each of the relations (which violate 3NF). Only R3 has a transitive dependency {A, D} -> {H} -> {J}, so it is decomposed into R31 and R32 as follows:
R31 = {H, J}, R32 = {A, D, G, H}
The final set of 3NF relations is {R1, R2, R31, R32, R4, R5}

14.26 – No solution provided.

14.27 – Consider a relation *R(A,B,C,D,E)* with the following dependencies:
AB -> C
CD -> E
DE -> B

Is *AB* a candidate key of this relation? If not, is *ABD*? Explain your answer.

Answers:
No, AB+ = {A,B,C},a proper subset of {A,B,C,D,E}
Yes, ABD+ = {A,B,C,D,E}

14.28 - Consider the relation R, which has attributes that hold schedules of courses and sections at a university; R = {CourseNo, SecNo, OfferingDept, CreditHours, CourseLevel, InstructorSSN, Semester, Year, Days_Hours, RoomNo, NoOfStudents}. Suppose that the following functional dependencies hold on R:
{CourseNo} -> {OfferingDept, CreditHours, CourseLevel}
{CourseNo, SecNo, Semester, Year} ->
{Days_Hours, RoomNo, NoOfStudents, InstructorSSN}
{RoomNo, Days_Hours, Semester, Year} -> {InstructorSSN, CourseNo, SecNo}

Try to determine which sets of attributes form keys of R. How would you normalize this relation?

Answer:
Let us use the following shorthand notation:
C = CourseNo, SN = SecNo, OD = OfferingDept, CH = CreditHours, CL = CourseLevel,
I = InstructorSSN, S = Semester, Y = Year, D = Days_Hours, RM = RoomNo,
NS = NoOfStudents
Hence, R = {C, SN, OD, CH, CL, I, S, Y, D, RM, NS}, and the following functional dependencies hold:
{C} -> {OD, CH, CL}
{C, SN, S, Y} -> {D, RM, NS, I}
{RM, D, S, Y} -> {I, C, SN}
First, we can calculate the closures for each left hand side of a functional dependency, since these sets of attributes are the candidates to be keys:
(1) {C}+ = {C, OD, CH, CL}
(2) Since {C, SN, S, Y} -> {D, RM, NS, I}, and {C}+ = {C, OD, CH, CL}, we get:
{C, SN, S, Y}+ = {C, SN, S, Y, D, RM, NS, I, OD, CH, CL} = R
(3) Since {RM, D, S, Y} -> {I, C, SN}, we know that {RM, D, S, Y}+ contains {RM, D, S, Y, I, C, SN}. But {C}+ contains {OD, CH, CL} so these are also contained in {RM, D, S, Y}+ since C is already there. Finally, since {C, SN, S, Y} are now all in {RM, D, S, Y}+ and {C, SN, S, Y}+ contains {NS} (from (2) above), we get:
{RM, D, S, Y}+ = {RM, D, S, Y, I, C, SN, OD, CH, CL, NS} = R
Hence, both K1 = {C, SN, S, Y} and K2 = {RM, D, S, Y} are (candidate) keys of R. By applying the general definition of 2NF, we find that the functional dependency {C} -> {OD, CH, CL} is a partial dependency for K1 (since C is included in K1). Hence, R is normalized into R1 and R2 as follows:
R1 = {C, OD, CH, CL}
R2 = {RM, D, S, Y, I, C, SN, NS} with candidate keys K1 and K2
Since neither R1 nor R2 have transitive dependencies on either of the candidate keys, R1 and R2 are in 3NF also. They also both satisfy the definition of BCNF.

14.29 - Consider the following relations for an order-processing application database at ABC, Inc.

ORDER (O#, Odate, Cust#, Total_amount)
ORDER-ITEM (O#, I#, Qty_ordered, Total_price, Discount%)

Assume that each item has a different discount. The Total_price refers to one item, Odate is the date on which the order was placed, and the Total_amount is the amount of the order. If we apply a natural join on the relations Order-Item and Order in this database, what does the resulting relation schema look like? What will be its key? Show the FDs in this resulting relation. Is it in 2NF? Is it in 3NF? Why or why not? (State any assumptions you make.)

Answer:
Given relations
Order(O#, Odate, Cust#, Total_amt)
Order_Item(O#, I#, Qty_ordered, Total_price, Discount%),
the schema of Order * Order_Item looks like

T_1(O#,I#,Odate, Cust#, Total_amount, Qty_ordered, Total_price, Discount%)
and its key is O#,I#.
It has functional dependencies
O#I# . Qty_ordered
O#I# .Total_price
O#I# .Discount%
O# . Odate
O# .Cust#
O# .Total_amount
It is not in 2NF, as attributes Odate, Cut#, and Total_amount are only partially dependent on the primary key, O#I#
Nor is it in 3NF, as a 2NF is a requirement for 3NF.

14.30 - Consider the following relation:
CAR_SALE(Car#, Date_sold, Salesman#, Commision%, Discount_amt
Assume that a car may be sold by multiple salesmen and hence {CAR#, SALESMAN#} is the primary key. Additional dependencies are:
Date_sold ->Discount_amt
and
Salesman# ->commission%

Based on the given primary key, is this relation in 1NF, 2NF, or 3NF? Why or why not? How would you successively normalize it completely?

Answer:
Given the relation schema
Car_Sale(Car#, Salesman#, Date_sold, Commission%, Discount_amt)
with the functional dependencies
Date_sold □□Discount_amt
Salesman# □□Commission%
Car# □□Date_sold
This relation satisfies 1NF but not 2NF (Car# □□Date_sold and Car# □
Discount_amt
so these two attributes are not FFD on the primary key) and not 3NF.
To normalize,
2NF:
Car_Sale1(Car#, Date_sold, Discount_amt)
Car_Sale2(Car#, Salesman#)
Car_Sale3(Salesman#,Commission%)
3NF:
Car_Sale1-1(Car#, Date_sold)
Car_Sale1-2(Date_sold, Discount_amt)
Car_Sale2(Car#, Salesman#)
Car_Sale3(Salesman#,Commission%)

14.31 - Consider the following relation for published books:
BOOK (Book_title, Authorname, Book_type, Listprice, Author_affil, Publisher)
Author_affil referes to the affiliation of the author. Suppose the following dependencies exist:
Book_title -> Publisher, Book_type
Book_type -> Listprice
Author_name -> Author-affil

(a) What normal form is the relation in? Explain your answer.
(b) Apply normalization until you cannot decompose the relations further. State the reasons behind each decomposition.

Answer:
Given the relation
Book(Book_title, Authorname, Book_type, Listprice, Author_affil, Publisher)
and the FDs
Book_title □□Publisher, Book_type
Book_type □□Listprice
Authorname □Author_affil

(a)The key for this relation is Book_title,Authorname. This relation is in 1NF and not in 2NF as no attributes are FFD on the key. It is also not in 3NF.

(b) 2NF decomposition:
Book0(Book_title, Authorname)
Book1(Book_title, Publisher, Book_type, Listprice)
Book2(Authorname, Author_affil)
This decomposition eliminates the partial dependencies.
3NF decomposition:
Book0(Book_title, Authorname)
Book1-1(Book_title, Publisher, Book_type)
Book1-2(Book_type, Listprice)
Book2(Authorname, Author_affil)

This decomposition eliminates the transitive dependency of Listprice

14.32 - This exercise asks you to converting business statements into dependencies. Consider the following relation DiskDrive(serialNumber, manufacturer, model, batch, capacity, retailer). Each tuple in the relation DiskDrive contains information about a disk drive with a unique serialNumber, made by a manufacturer, with a particular model, released in a certain batch, which has a certain storage capacity, and is sold by a certain retailer. For example, the tuple DiskDrive(1978619, WesternDigital, A2235X, 765234, 500, CompUSA) specifies that WesternDigital made a disk drive with serial number 1978619, model number A2235X in batch 765235 with 500GB that is sold by CompUSA.

Write each of the following dependencies as an FD:
a. The manufacturer and serial number uniquely identifies the drive
b. A model number is registered by a manufacturer and hence can't be used by another manufacturer.
c. All disk drives in a particular batch are the same model.
d. All disk drives of a particular model of a particular manufacturer have exactly the same capacity.

Answer:
a. manufacturer, serialNumber → model, batch, capacity, retailer

b. model → manufacturer

c. manufacturer, batch → model

d. model → capacity (Comment: 10.35.d can be "model, manufacturer → capacity." There is nothing in the question that suggests we should "optimize" our FDs by assuming the other requirements)

14.33 - Consider the following relation:

R (Doctor#, Patient#, Date, Diagnosis, Treat_code, Charge)

In this relation, a tuple describes a visit of a patient to a doctor along with a treatment code and daily charge. Assume that diagnosis is determined (uniquely) for each patient by a doctor. Assume that each treatment code has a fixed charge (regardless of patient). Is this relation in 2NF? Justify your answer and decompose if necessary. Then argue whether further normalization to 3NF is necessary, and if so, perform it.

Answer:
From the question's text, we can infer the following functional dependencies:

{Doctor#, Patient#, Date}→{Diagnosis, Treat_code, Charge}
{Treat_code}→{Charge}

Because there are no partial dependencies, the given relation is in 2NF already. This however is not 3NF because the Charge is a nonkey attribute that is determined by another nonkey attribute, Treat_code. We must decompose further:

R (Doctor#, Patient#, Date, Diagnosis, Treat_code)
R1 (Treat_code, Charge)

We could further infer that the treatment for a given diagnosis is functionally dependant, but we should be sure to allow the doctor to have some flexibility when prescribing cures.

14.34 - Consider the following relation:
CAR_SALE (CarID, Option_type, Option_Listprice, Sale_date, Discounted_price)
This relation refers to options installed on cars (e.g.- cruise control) that were sold at a dealership and the list and discounted prices for the options.
If CarID → Sale_date and Option_type→ Option_Listprice, and
CarID, Option_type → Discounted_price, argue using the generalized definition of the 3NF that this relation is not in 3NF. Then argue from your knowledge of 2NF, why it is not in 2NF.

Answer:
For this relation to be in 3NF, all of the nontrivial functional dependencies must both be fully functional and nontransitive on every key of the relation. However, in this relation we have two dependencies (CarID → Sale_date and Option_type → Option_Listprice) that violate

these requirements. Both of these dependencies are partial and transitive.

For this relation to be in 2NF, all of the nontrivial functional dependencies must be fully functional on every key of the relation. Again, as was discussed about 3NF, this relation has two partial dependencies that violate this requirement.

14.35 - Consider the relation:
BOOK (Book_Name, Author, Edition, Year)
with the data:

Book_Name	Author	Edition	Year
DB_fundamentals	Navathe	4	2004
DB_fundamentals	Elmasri	4	2004
DB_fundamentals	Elmasri	5	2007
DB_fundamentals	Navathe	5	2007

a. Based on a common-sense understanding of the above data, what are the possible candidate keys of this relation?

b. Does the above have one or more functional dependency (do not list FDs by applying derivation rules)? If so, what is it? Show how you will remove it by decomposition.

c. Does the resulting relation have an MVD? If so, what is it?

d. What will the final decomposition look like?

Answer:
a. The only candidate key is {Book_Name, Author, Edition}. From the example, it would appear that {Book_Name, Author, Year} would also be a candidate key but we should consider that some books may have a release cycle which causes multiple editions to appear in a given year.

b. Yes, we have the following FD: Book_Name, Edition $\rightarrow$ Year. We can decompose to remove this FD in the following way:

BOOK (Book_Name, Author, Edition)
BOOK_YEAR (Book_Name, Edition, Year)

c. Yes, BOOK contains Book_Name $\rightarrow\rightarrow$ Author and Book_Name $\rightarrow\rightarrow$ Edition.

d. The final decomposition would look like:

BOOK (Book_Name, Edition)
BOOK_AUTHOR (Book_Name, Edition, Author)
BOOK_YEAR (Book_Name, Edition, Year)

14.36 - Consider the following relation:
TRIP (trip_id, start_date, cities_visited, cards_used)

This relation refers to business trips made by salesmen in a company. Suppose the trip has a single start_date but involves many cities and one may use multiple credit cards for that trip. Make up a mock-up population of the table.

a. Discuss what FDs and / or MVDs exist in this relation.
b. Show how you will go about normalizing it.

Answer:
a. The TRIP relation has the following FDs and MVDs:

trip_id $\rightarrow$ start_date
trip_id $\rightarrow\rightarrow$ cities_visited
trip_id $\rightarrow\rightarrow$ cards_used

b. Because there are no interdependencies, this relation can be trivially decomposed to conform to 4NF:

TRIP_DATE (trip_id, start_date)
TRIP_CITIES (trip_id, cities_visited)
TRIP_CARDS (trip_id, cards_used)
.

CHAPTER 15: Relational Database Design Algorithms and Further Dependencies

Answers to Selected Exercises

15.17 - Show that the relation schemas produced by Algorithm 15.4 are in 3NF.

Answer:
We give a proof by contradiction. Suppose that one of the relations R i resulting from Algorithm 15.1 is not in 3NF. Then a FD Y -> A holds R i in where: (a) Y is not a superkey of R, and (b) A is not a prime attribute. But according to step 2 of the algorithm, R i will contain a set of attributes X union A 1 union A 2 union ... union A n , where X -> A i for i=1, 2, ..., n, implying that X is a key of R i and the A i are the only non-prime attributes of R i . Hence, if an FD Y -> A holds in R i where A is non-prime and Y is not a superkey of R i , Y must be a proper subset of X (otherwise Y would contain X and hence be a superkey). If both Y -> A and X -> A hold and Y is a proper subset of X, this contradicts that X -> A is a FD in a minimal set of FDs that is input to the algorithm, since removing an attribute from X leaves a valid FD, thus violating one of the minimality conditions. This produces a contradiction of our assumptions. Hence, R i must be in 3NF.

15.18 - Show that, if the matrix S resulting from Algorithm 15.3 does not have a row that is all "a" symbols, then projecting S on the decomposition and joining it back will always produce at least one spurious tuple.

Answer:

The matrix S initially has one row for each relation R i in the decomposition, with "a" symbols under the columns for the attributes in R i . Since we never change an "a" symbol into a "b" symbol during the application of the algorithm, then projecting S on each R i at the end of applying the algorithm will produce one row consisting of all "a" symbols in each S(R i). Joining these back together again will produce at least one row of all "a" symbols (resulting from joining the all "a" rows in each projection S(R i)). Hence, if after applying the algorithm, S does not have a row that is all "a", projecting S over the R i 's and joining will result in at least one all "a" row, which will be a spurious tuple (since it did not exist in S but will exist after projecting and joining over the R i 's).

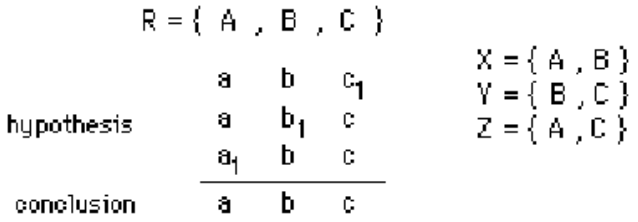
15.19 - Show that the relation schemas produced by Algorithm 15.5 are in BCNF.

Answer:
This is trivial, since the algorithm loop will continue to be applied until all relation schemas are in BCNF.

15.20 – **No Solution Provided**

15.21 - Specify a template dependency for join dependencies.

Answer:
The following template specifies a join dependency JD(X,Y,Z).



15.22 - Specify all the inclusion dependencies for the relational schema of Figure 3.5.

Answer:
The inclusion dependencies will correspond to the foreign keys shown in Figure 3.7.

15.23 - Prove that a functional dependency satisfies the formal definition of multi-valued dependency.

Answer:
Suppose that a functional dependency X -> Y exists in a relation R={X, Y, Z}, and suppose there are two tuples with the same value of X. Because of the functional dependency, they must also have the same value of Y. Suppose the tuples are t 1 = < x, y, z 1 > and t 2 = < x, y, z 2 >. Then, according to the definition of multivalued dependency, we must have two tuples t 3 and t 4 (not necessarily distinct from t 1 and t 2) satisfying: t 3 [X]= t 4 [X]= t 1 [X]= t 2 [X], t 3 [Y]= t 2 [Y], t 4 [Y]= t 1 [Y], t 3 [Z]= t 1 [Z], and t 4 [Z]= t 2 [Z]. Two tuples satisfying this are t 2 (satisfies conditions for t 4) and t 1 (satisfies conditions for t 3). Hence, whenever the condition for functional dependency holds, so does the condition for multivalued dependency.

15.24 - 15.31: No solutions provided.

15.32 - Consider the relation REFRIG(MODEL#, YEAR, PRICE, MANUF_PLANT, COLOR), which is abbreviated as REFRIG(M, Y, P, MP, C), and the following set of F of functional dependencies: F={M -> MP, {M,Y} -> P, MP -> C}

(a) Evaluate each of the following as a candidate key for REFRIG, giving reasons why it can or cannot be a key: {M}, {M,Y}, {M.C}

(b) Based on the above key determination, state whether the relation REFRIG is in 3NF and in BCNF, giving proper reasons.

(c) Consider the decomposition of REFRIG into D={R1(M,Y,P), R2(M,MP,C)}. Is this decomposition lossless? Show why. (You may consult the test under Property LJ1 in Section 15.2.4)

Answers:
(a)
- {M} IS NOT a candidate key since it does not functionally determine attributes Y or P.
- {M, Y} IS a candidate key since it functionally determines the remaining attributes P, MP, and C.
i.e.
{M, Y} P, But M MP
By augmentation {M, Y} MP
Since MP C, by transitivity M MP, MP C, gives M C
By augmentation {M, Y} C
Thus {M, Y} P, MP, C and {M, Y} can be a candidiate key
- {M, C} IS NOT a candidate key since it does not functionally determine attributes Y or P.

(b)
REFRIG is not in 2NF, due to the partial dependency {M, Y} MP (since {M} MP holds). Therefore REFRIG is neither in 3NF nor in BCNF.
Alternatively: BCNF can be directly tested by using all of the given dependencies and finding out if the left hand side of each is a superkey (or if the right hand side is a prime attribute). In the two fields in REFRIG: M MP and MP C. Since neither M nor MP is a superkey, we can conclude that REFRIG is is neither in 3NF nor in BCNF.

(c)

$R = \{M, Y, P, MP, C\}$
$R1 = \{M, Y, P\}$
$R2 = \{M, MP, C\}$

$F = \{M \quad MP, \ \{M, Y\} \quad P, \ MP \quad C\}$
$F^+ = \{ \{M\}^+ \quad \{M, MP, C\},$
$\{M, Y\}^+ \quad \{M, Y, P, MP, C\},$
$\{MP\}^+ \quad \{MP, C\} \}$

$R1 \cap R2 = M$
$R2 - R1 = \{MP, C\}$

D(R1, R2} has the lossless join property since
Property: *LJ1: FD ((R1 ∩ R2) (R2 – R1)) is in* F^+
is satisfied (due to M {MP, C}).

CHAPTER 16: Disk Storage, Basic File Structures, Hashing, and Modern Storage Architectures

Answers to Selected Exercises

16.27 - Consider a disk with the following characteristics (these are not parameters of any particular disk unit): block size B=512 bytes, interblock gap size G=128 bytes, number of blocks per track=20, number of tracks per surface=400. A disk pack consists of 15 double-sided disks.

(a) What is the total capacity of a track and what is its useful capacity (excluding interblock gaps)?

(b) How many cylinders are there?

(c) What is the total capacity and the useful capacity of a cylinder?

(d) What is the total capacity and the useful capacity of a disk pack?

(e) Suppose the disk drive rotates the disk pack at a speed of 2400 rpm (revolutions per minute); what is the transfer rate in bytes/msec and the block transfer time btt in msec? What is the average rotational delay rd in msec? What is the bulk transfer rate (see Appendix B)?

(f) Suppose the average seek time is 30 msec. How much time does it take (on the average) in msec to locate and transfer a single block given its block address?

(g) Calculate the average time it would take to transfer 20 random blocks and compare it with the time it would take to transfer 20 consecutive blocks using double buffering to save seek time and rotational delay.

Answer:
(a) Total track size = 20 * (512+128) = 12800 bytes = 12.8 Kbytes
Useful capacity of a track = 20 * 512 = 10240 bytes = 10.24 Kbytes

(b) Number of cylinders = number of tracks = 400

(c) Total cylinder capacity = 15*2*20*(512+128) = 384000 bytes = 384 Kbytes
Useful cylinder capacity = 15 * 2 * 20 * 512 = 307200 bytes = 307.2 Kbytes

(d) Total capacity of a disk pack = 15 * 2 * 400 * 20 * (512+128)
= 153600000 bytes = 153.6 Mbytes
Useful capacity of a disk pack = 15 * 2 * 400 * 20 * 512 = 122.88 Mbytes

(e) Transfer rate tr= (total track size in bytes)/(time for one disk revolution in msec)
tr= (12800) / ((60 * 1000) / (2400)) = (12800) / (25) = 512 bytes/msec
block transfer time btt = B / tr = 512 / 512 = 1 msec
average rotational delay rd = (time for one disk revolution in msec) / 2 = 25 / 2
= 12.5 msec
bulk transfer rate btr= tr * (B/(B+G)) = 512*(512/640) = 409.6 bytes/msec

(f) average time to locate and transfer a block = s+rd+btt = 30+12.5+1 = 43.5 msec

(g) time to transfer 20 random blocks = 20 * (s + rd + btt) = 20 * 43.5 = 870 msec
time to transfer 20 consecutive blocks using double buffering = s + rd + 20*btt
= 30 + 12.5 + (20*1) = 62.5 msec
(a more accurate estimate of the latter can be calculated using the bulk transfer rate as follows: time to transfer 20 consecutive blocks using double buffering
= s+rd+((20*B)/btr) = 30+12.5+ (10240/409.6) = 42.5+ 25 = 67.5 msec)

16.28 - A file has r=20,000 STUDENT records of fixed-length. Each record has the following fields: NAME (30 bytes), SSN (9 bytes), ADDRESS (40 bytes), PHONE (9 bytes), BIRTHDATE (8 bytes), SEX (1 byte), MAJORDEPTCODE (4 bytes), MINORDEPTCODE (4 bytes), CLASSCODE (4 bytes, integer), and DEGREEPROGRAM (3 bytes). An additional byte is used as a deletion marker. The file is stored on the disk whose parameters are given in Exercise 16.27.

(a) Calculate the record size R in bytes.

(b) Calculate the blocking factor bfr and the number of file blocks b assuming an unspanned organization.

(c) Calculate the average time it takes to find a record by doing a linear search on the file if (i) the file blocks are stored contiguously and double buffering is used, and (ii) the file blocks are not stored contiguously.

(d) Assume the file is ordered by SSN; calculate the time it takes to search for a record given its SSN value by doing a binary search.

Answer:
(a) R = (30 + 9 + 40 + 9 + 8 + 1 + 4 + 4 + 4 + 3) + 1 = 113 bytes

(b) bfr = floor(B / R) = floor(512 / 113) = 4 records per block
b = ceiling(r / bfr) = ceiling(20000 / 4) = 5000 blocks

(c) For linear search we search on average half the file blocks= 5000/2= 2500 blocks.
i. If the blocks are stored consecutively, and double buffering is used, the time to read 2500 consecutive blocks
= s+rd+(2500*(B/btr))= 30+12.5+(2500*(512/409.6))
= 3167.5 msec = 3.1675 sec
(a less accurate estimate is = s+rd+(2500*btt)= 30+12.5+2500*1= 2542.5 msec)
ii. If the blocks are scattered over the disk, a seek is needed for each block, so the time is: 2500 * (s + rd + btt) = 2500 * (30 + 12.5 + 1) = 108750 msec = 108.75 sec

(d) For binary search, the time to search for a record is estimated as:
ceiling(log 2 b) * (s +rd + btt)
= ceiling(log 2 5000) * (30 + 12.5 + 1) = 13 * 43.5 = 565.5 msec = 0.5655 sec

13.25 Suppose that only 80% of the STUDENT records from Exercise 16.28 have a value for PHONE, 85% for MAJORDEPTCODE, 15% for MINORDEPTCODE, and 90% for DEGREEPROGRAM, and we use a variable-length record file. Each record has a 1-byte field type for each field occurring in the record, plus the 1-byte deletion marker and a 1-byte end-of-record marker. Suppose we use a spanned record organization, where each block has a 5-byte pointer to the next block (this space is not used for record storage).

(a) Calculate the average record length R in bytes.

(b) Calculate the number of blocks needed for the file.

Answer:

(a) Assuming that every field has a 1-byte field type, and that the fields not mentioned above (NAME, SSN, ADDRESS, BIRTHDATE, SEX, CLASSCODE) have values in every record, we need the following number of bytes for these fields in each record, plus 1 byte for the deletion marker, and 1 byte for the end-of-record marker:
R fixed = (30+1) + (9+1) + (40+1) + (8+1) + (1+1) + (4+1) +1+1 = 100 bytes
For the fields (PHONE, MAJORDEPTCODE, MINORDEPTCODE DEGREEPROGRAM), the average number of bytes per record is:
R variable = ((9+1)*0.8)+((4+1)*0.85)+((4+1)*0.15)+((3+1)*0.9)
= 8+4.25+0.75+3.6= 16.6 bytes
The average record size R = R fixed + R variable = 100 + 16.6 = 116.6 bytes
The total bytes needed for the whole file = r * R = 20000 * 116.6 = 2332000 bytes

(b) Using a spanned record organization with a 5-byte pointer at the end of each block, the bytes available in each block are (B-5) = (512 - 5) = 507 bytes.
The number of blocks needed for the file are:
b = ceiling((r * R) / (B - 5)) = ceiling(2332000 / 507) = 4600 blocks
(compare this with the 5000 blocks needed for fixed-length, unspanned records in Problem 4.19(b))

16.30 - Suppose that a disk unit has the following parameters: seek time s=20 msec; rotational delay rd=10 msec; block transfer time btt=1 msec; block size B=2400 bytes;

interblock gap size G=600 bytes. An EMPLOYEE file has the following fields: SSN, 9 bytes; LASTNAME, 20 bytes; FIRSTNAME, 20 bytes; MIDDLE INIT, 1 byte; BIRTHDATE, 10 bytes; ADDRESS, 35 bytes); PHONE, 12 bytes); SUPERVISORSSN, 9 bytes; DEPARTMENT, 4 bytes; JOBCODE, 4 bytes; deletion marker, 1 byte. The EMPLOYEE file has r=30000 STUDENT records, fixed-length format, and unspanned blocking. Write down appropriate formulas and calculate the following values for the above EMPLOYEE file:

(a) The record size R (including the deletion marker), the blocking factor bfr, and the number of disk blocks b.

(b) Calculate the wasted space in each disk block because of the unspanned organization.

(c) Calculate the transfer rate tr and the bulk transfer rate btr for this disk (see Appendix B for definitions of tr and btr).

(d) Calculate the average number of block accesses needed to search for an arbitrary record in the file, using linear search.

(e) Calculate the average time needed in msec to search for an arbitrary record in the file, using linear search, if the file blocks are stored on consecutive disk blocks and double buffering is used.

(f) Calculate the average time needed in msec to search for an arbitrary record in the file, using linear search, if the file blocks are not stored on consecutive disk blocks.

(g) Assume that the records are ordered via some key field. Calculate the average number of block accesses and the average time needed to search for an arbitrary record in the file, using binary search.

Answer:
(a) R = (9 + 20 + 20 + 1 + 10 + 35 + 12 + 9 + 4 + 4) + 1 = 125 bytes
bfr = floor(B / R) = floor(2400 / 125) = 19 records per block
b = ceiling(r / bfr) = ceiling(30000 / 19) = 1579 blocks

(b) Wasted space per block = B - (R * Bfr) = 2400 - (125 * 19) = 25 bytes

(c) Transfer rate tr= B/btt = 2400 / 1 = 2400 bytes/msec
bulk transfer rate btr= tr * (B/(B+G))
= 2400*(2400/(2400+600)) = 1920 bytes/msec

(d) For linear search we have the following cases:
i. search on key field:
if record is found, half the file blocks are searched on average: b/2= 1579/2 blocks
if record is not found, all file blocks are searched: b = 1579 blocks
ii. search on non-key field:
all file blocks must be searched: b = 1579 blocks

(e) If the blocks are stored consecutively, and double buffering is used, the time to read n consecutive blocks= s+rd+(n*(B/btr))
i. if n=b/2: time = 20+10+((1579/2)*(2400/1920))= 1016.9 msec = 1.016 sec
(a less accurate estimate is = s+rd+(n*btt)= 20+10+(1579/2)*1= 819.5 msec)
ii. if n=b: time = 20+10+(1579*(2400/1920))= 2003.75 msec = 2.004 sec

(a less accurate estimate is = s+rd+(n*btt)= 20+10+1579*1= 1609 msec)

(f) If the blocks are scattered over the disk, a seek is needed for each block, so the time to search n blocks is: n * (s + rd + btt)
i. if n=b/2: time = (1579/2)*(20+10+1)= 24474.5 msec = 24.475 sec
ii. if n=b: time = 1579*(20+10+1)= 48949 msec = 48.949 sec

(g) For binary search, the time to search for a record is estimated as:
ceiling(log 2 b) * (s +rd + btt)
= ceiling(log 2 1579) * (20+10+1) = 11 * 31 = 341 msec = 0.341 sec

16.31 - A PARTS file with Part# as hash key includes records with the following Part# values: 2369, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428, 3943, 4750, 6975, 4981, 9208. The file uses 8 buckets, numbered 0 to 7. Each bucket is one disk block and holds two records. Load these records into the file in the given order using the hash function h(K)=K mod 8. Calculate the average number of block accesses for a random retrieval on Part#.

Answer:
The records will hash to the following buckets:
K h(K) (bucket number)
2369 1
3760 0
4692 4
4871 7
5659 3
1821 5
1074 2
7115 3
1620 4
2428 4 overflow
3943 7
4750 6
6975 7 overflow
4981 5
9208 0
9209

Two records out of 15 are in overflow, which will require an additional block access. The other records require only one block access. Hence, the average time to retrieve a random record is:
(1 * (13/15)) + (2 * (2/15)) = 0.867 + 0.266 = 1.133 block accesses

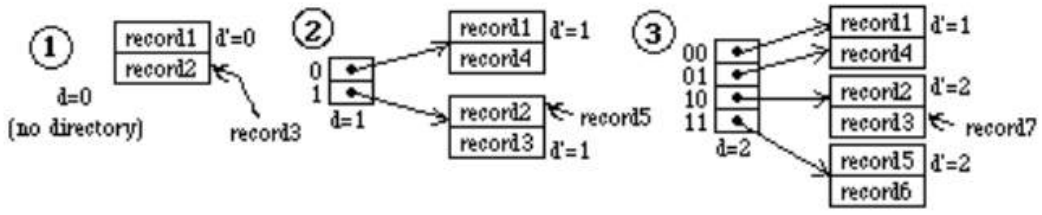
16.32 - Load the records of Exercise 16.31 into expandable hash files based on extendible hashing. Show the structure of the directory at each step. Show the directory at each step, and the global and local depths. Use the hash function h(k) = K mod 128.

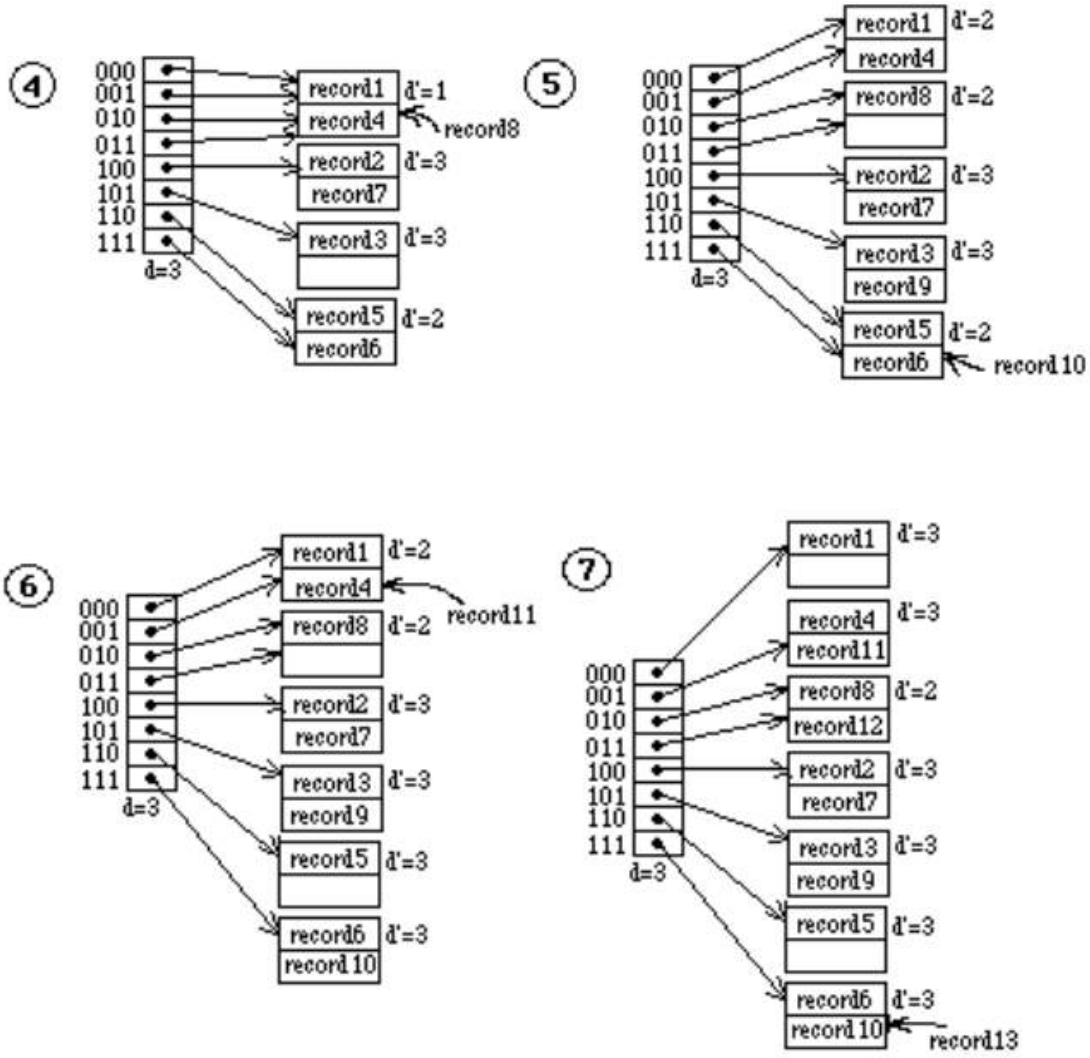
Answer:
Hashing the records gives the following result:

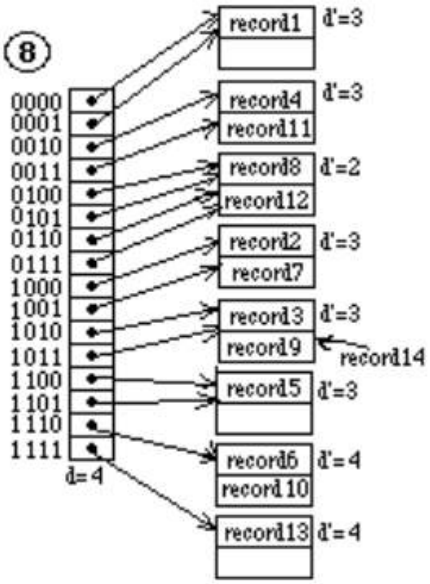
	K	h(K) (bucket number)	binary h(K)
record1	2369	1	00001
record2	3760	16	10000
record3	4692	20	10100
record4	4871	7	00111
record5	5659	27	11011
record6	1821	29	11101
record7	1074	18	10010

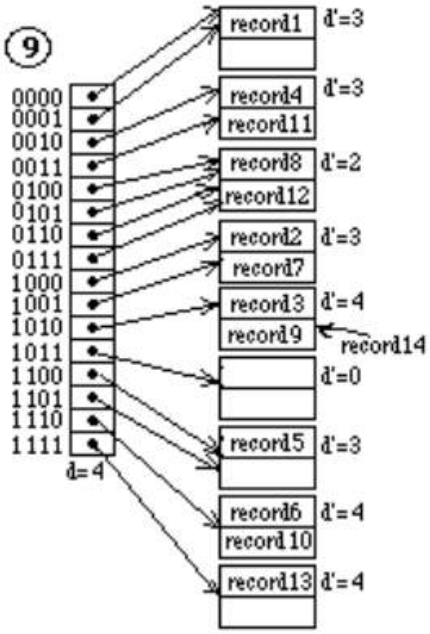
record8	7115	11	01011
record9	1620	20	10100
record10	2428	28	11100
record11	3943	7	00111
record12	4750	14	01110
record13	6975	31	11111
record14	4981	21	10101
record15	9208	24	11000

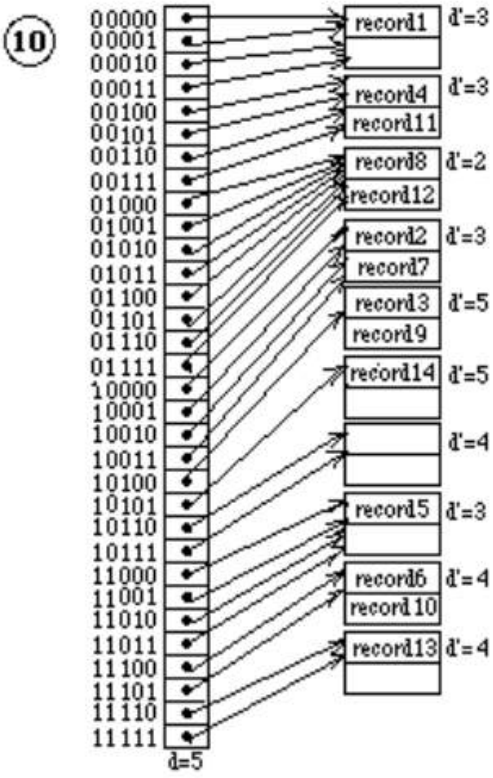
Extendible hashing:




4
000
001
010
011
100
101
110
111
d=3
record1
record4
record8
record2
record7
record3
record5
record6
d'=1
d'=3
d'=3
d'=2
5
record1
record4
record8
record2
record7
record3
record9
record5
record6
record10
d'=2
d'=2
d'=3
d'=3
d'=2
6
record1
record4
record8
record2
record7
record3
record9
record5
record6
record10
record11
d'=2
d'=2
d'=3
d'=3
d'=3
d'=3
7
record1
record4
record11
record8
record12
record2
record7
record3
record9
record5
record6
record10
record13
d'=3
d'=3
d'=2
d'=3
d'=3
d'=3
d'=3

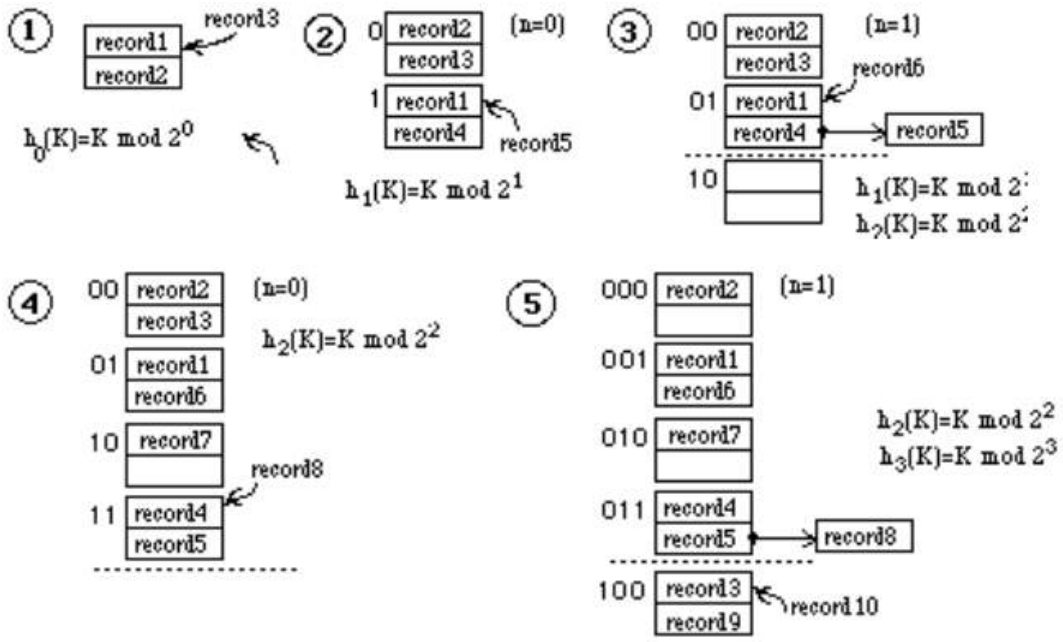

8
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
d=4
record1
d'=3
record4
record11
d'=3
record8
record12
d'=2
record2
record7
d'=3
record3
record9
d'=3
record14
record5
d'=3
record6
record10
d'=4
record13
d'=4

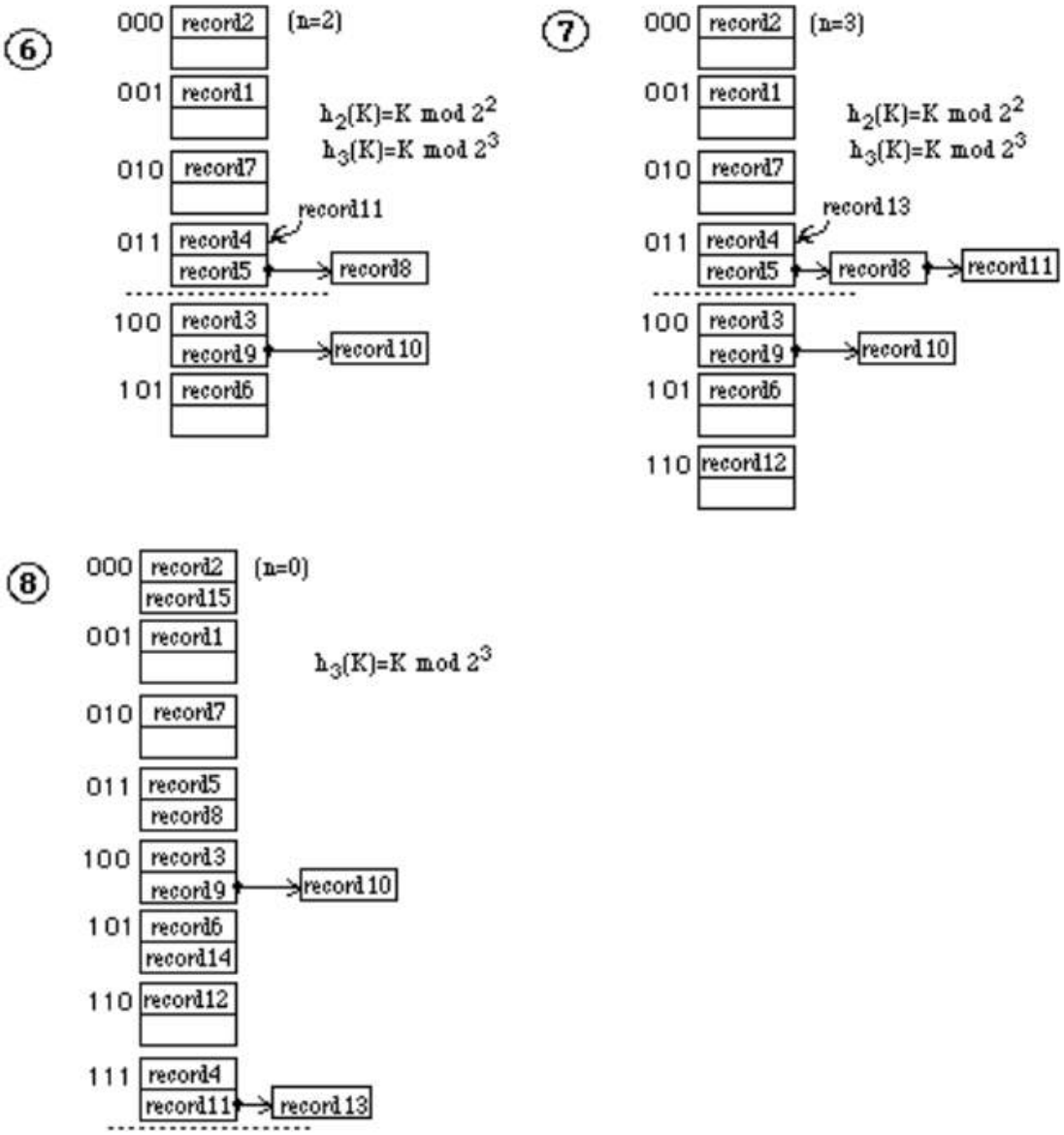

9
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
d=4
record1
d'=3
record4
record11
d'=3
record8
record12
d'=2
record2
record7
d'=3
record3
record9
d'=4
record14
d'=0
record5
d'=3
record6
record10
d'=4
record13
d'=4


10
00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111
d=5
record1
d'=3
record4
record11
d'=3
record8
record12
d'=2
record2
record7
d'=3
record3
record9
d'=5
record14
d'=5
d'=4
record5
d'=3
record6
record10
d'=4
record13
d'=4

16.33 - Load the records of Exercise 16.31 into expandable hash files based on linear hashing. Start with a single disk block, using the hash function *h ø = K* mod 2º, and show how the file grows and how the hash functions change as the records are inserted. Assume that blocks are split whenever an overflow occurs, and show the value of n at each stage

Answer:





Note: It is more common to specify a certain load factor for the file for triggering the splitting of buckets (rather than triggering the splitting whenever a new record being inserted is placed in overflow). The load factor lf could be defined as: lf = (r) / (b * Bfr) where r is the current number of records, b is the current number of buckets, and Bfr is the maximum

number of records per bucket. Whenever lf gets to be larger than some threshold, say 0.8, a split is triggered. It is also possible to merge brackets in the reverse order in which they were created; a merge operation would be triggered whenever lf becomes less than another threshold, say 0.6.

16.34 – 16.36: No solutions provided

16.37 - Can you think of techniques other than an unordered overflow file that can be used to make insertion in an ordered file more efficient?

Answer:
It is possible to use an overflow file in which the records are chained together in a manner similar to the overflow for static hash files. The overflow records that should be inserted in each block of the ordered file are linked together in the overflow file, and a pointer to the first record in the chain (linked list) is kept in the block of the main file.
The list may or may not be kept ordered.

16.38 - No solution provided.

16.39 - Can you think of techniques other than chaining to handle bucket overflow in external hashing?

Answer:
One can use techniques for handling collisions similar to those used for internal hashing. For example, if a bucket is full, the record which should be inserted in that bucket may be placed in the next bucket if there is space (open addressing). Another scheme is to use a whole overflow block for each bucket that is full. However, chaining seems to be the most appropriate technique for static external hashing.

16.40 – 16.41: No solutions provided.

16.42 - Suppose that a file initially contains r=120,000 records of R=200 bytes each in an unsorted (heap) file. The block size B=2400 bytes, the average seek time s=16 ms, the average rotational latency rd=8.3 ms and the block transfer time btt=0.8 ms. Assume that 1 record is deleted for every 2 records added until the total number of active records is 240,000.

(a) How many block transfers are needed to reorganize the file?

(b) How long does it take to find a record right before reorganization?

(c) How long does it take to find a record right after reorganization?

Answer:
Let X = # of records deleted
Hence 2X= # of records added.
Total active records
= 240,000 = 120,000 - X + 2X.
Hence, X = 120,000
Records before reorganization (i.e., before deleting any records physically) = 360,000.

(a) No. of blocks for Reorganization
= Blocks Read + Blocks Written.
-200 bytes/record and 2400 bytes/block gives us 12 records per block
-Reading involves 360,000 records; i.e. 360,000/12 = 30K blocks
-Writing involves 240,000 records; i.e., 240000/12 = 20K blocks.
Total blocks transferred during reorganization
= 30K + 20K = 50K blocks.

(b) Time to locate a record before reorganization. On an average we assume that half the file will be read.
Hence, Time = (b/2)* btt = 15000 * 0.8 ms = 12000 ms.
= 12 sec.

(c) Time to locate a record after reorganization
= (b/2) * btt = 10000 * 0.8 = 8 sec.

16.43 - Suppose we have a sequential (ordered) file of 100000 records where each record is 240 bytes. Assume that B=2400 bytes, s=16 ms, rd=8.3 ms, and btt=0.8 ms. Suppose we want to make X independent random records from the file. We could make X random block reads or we could perform one exhaustive read of the entire file looking for those X records. The question is to decide when it would be more efficient to perform one exhaustive read of the entire file than to perform X individual random reads. That is, what is the value for X when an exhaustive read of the file is more efficient than random X reads? Develop this function of X.

Answer:
Total blocks in file = 100000 records * 240 bytes/record divided by 2400 bytes/block = 10000 blocks.
Time for exhaustive read
= s + r + b.btt
= 16 + 8.3 + (10000) * 0.8
= 8024.3 msec
Let X be the # of records searched randomly that takes more time than exhaustive read time.
Hence, X (s + r + btt) > 8024.3
X (16+8.3+0.8) > 8024.3
X > 8024.3/25.1
Thus, X > 319.69
i.e. If at least 320 random reads are to be made, it is better to search the file exhaustively.

16.44 - Suppose that a static hash file initially has 600 buckets in the primary area and that records are inserted that create an overflow area of 600 buckets. If we reorganize the hash file, we can assume that the overflow is eliminated. If the cost of reorganizing the file is the cost of the bucket transfers (reading and writing al of the buckets) and the only periodic file operation is the fetch operation, then how many times would we have to perform a fetch (successfully) to make the reorganization cost-effective? That is, the reorganization cost and subsequent search cost are less than the search cost before reorganization. Assume s=16, rd=8.3 ms, btt=1 ms.

Primary Area = 600 buckets
Secondary Area (Overflow) = 600 buckets
Total reorganization cost = Buckets Read & Buckets Written for (600 & 600) +
1200 = 2400 buckets = 2400 (1 ms) = 2400 ms
Let X = number of random fetches from the file.
Average Search time per fetch = time to access (1 + 1/2) buckets where 50% of time we need to access the overflow bucket.
Access time for one bucket access = (S + r + btt)
= 16 + 8.3 + 0-8
= 25.1 ms
Time with reorganization for the X fetches
= 2400 + X (25.1) ms
Time without reorganization for X fetches = X (25.1) (1 + 1/2) ms
= 1.5 * X * (25.1) ms.
Hence, 2400 + X (25.1) < (25.1) * (1.5X)
2374.9/ 12.55 < X
Hence, 189.23 < X
If we make at least 190 fetches, the reorganization is worthwhile.

16.45 - Suppose we want to create a linear hash file with a file load factor of 0.7 and a blocking factor of 20 records per bucket, which is to contain 112,000 records initially.

(a) How many buckets should we allocate in primary areas?

(b) What should be the number of bits used for bucket addresses?

Answer:
(a) No. of buckets in primary area.
= 112000/(20*0.7) = 8000.

(b) K: the number of bits used for bucket addresses
2K < = 8000 < = 2 k+1
2 12 = 4096
2 13 = 8192
K = 12
Boundary Value = 8000 - 2 12
= 8000 - 4096
= 3904 -

CHAPTER 17: Indexing Structures for Files and Physical Database Design

Answers to Selected Exercises

17.18 - Consider a disk with block size B=512 bytes. A block pointer is P=6 bytes long, and a record pointer is P R =7 bytes long. A file has r=30,000 EMPLOYEE records of fixed-length. Each record has the following fields: NAME (30 bytes), SSN (9 bytes), DEPARTMENTCODE (9 bytes), ADDRESS (40 bytes), PHONE (9 bytes), BIRTHDATE (8 bytes), SEX (1 byte), JOBCODE (4 bytes), SALARY (4 bytes, real number). An additional byte is used as a deletion marker.

(a) Calculate the record size R in bytes.

(b) Calculate the blocking factor bfr and the number of file blocks b assuming an unspanned organization.

(c) Suppose the file is ordered by the key field SSN and we want to construct a primary index on SSN. Calculate (i) the index blocking factor bfr i (which is also the index fan-out fo); (ii) the number of first-level index entries and the number of first-level index blocks; (iii) the number of levels needed if we make it into a multi-level index; (iv) the total number of blocks required by the multi-level index; and (v) the number of block accesses needed to search for and retrieve a record from the file--given its SSN value--using the primary index.

(d) Suppose the file is not ordered by the key field SSN and we want to construct a secondary index on SSN. Repeat the previous exercise (part c) for the secondary index and compare with the primary index.

(e) Suppose the file is not ordered by the non-key field DEPARTMENTCODE and we want to construct a secondary index on SSN using Option 3 of Section 17.1.3, with an extra level of indirection that stores record pointers. Assume there are 1000 distinct values of DEPARTMENTCODE, and that the EMPLOYEE records are evenly distributed among these values. Calculate (i) the index blocking factor bfr i (which is also the index fan-out fo); (ii) the number of blocks needed by the level of indirection that stores record pointers; (iii) the number of first-level index entries and the number of first-level index blocks; (iv) the number of levels needed if we make it a multi-level index; (v) the total number of blocks required by the multi-level index and the blocks used in the extra level of indirection; and (vi) the approximate number of block accesses needed to search for and retrieve all records in the file having a specific DEPARTMENTCODE value using the index.

(f) Suppose the file is ordered by the non-key field DEPARTMENTCODE and we want to construct a clustering index on DEPARTMENTCODE that uses block anchors (every new value of DEPARTMENTCODE starts at the beginning of a new block). Assume there are 1000 distinct values of DEPARTMENTCODE, and that the EMPLOYEE records are evenly distributed among these values. Calculate (i) the index blocking factor bfr i (which is also the index fan-out fo); (ii) the number of first-level index entries and the number of first-level index blocks; (iii) the number of levels needed if we make it a multi-level index; (iv) the total number of blocks required by the multi-level index; and (v) the number of block accesses needed to search for and retrieve all records in the file having a specific DEPARTMENTCODE value using the clustering index (assume that multiple blocks in a cluster are either contiguous or linked by pointers).

(g) Suppose the file is not ordered by the key field Ssn and we want to construct a B + -tree

access structure (index) on SSN. Calculate (i) the orders p and p leaf of the B + -tree; (ii) the number of leaf-level blocks needed if blocks are approximately 69% full (rounded up for convenience); (iii) the number of levels needed if internal nodes are also 69% full (rounded up for convenience); (iv) the total number of blocks required by the B + -tree; and (v) the number of block accesses needed to search for and retrieve a record from the file--given its SSN value--using the B + -tree.

Answer:
(a) Record length R = (30 + 9 + 9 + 40 + 9 + 8 + 1 + 4 + 4) + 1 = 115 bytes

(b) Blocking factor bfr = floor(B/R) = floor(512/115) = 4 records per block
Number of blocks needed for file = ceiling(r/bfr) = ceiling(30000/4) = 7500

(c) i. Index record size R i = (V SSN + P) = (9 + 6) = 15 bytes
Index blocking factor bfr i = fo = floor(B/R i) = floor(512/15) = 34
ii. Number of first-level index entries r 1 = number of file blocks b = 7500 entries
Number of first-level index blocks b 1 = ceiling(r 1 /bfr i) = ceiling(7500/34)
= 221 blocks
iii. We can calculate the number of levels as follows:
Number of second-level index entries r 2 = number of first-level blocks b 1
= 221 entries
Number of second-level index blocks b 2 = ceiling(r 2 /bfr i) = ceiling(221/34)
= 7 blocks
Number of third-level index entries r 3 = number of second-level index blocks b 2
= 7 entries
Number of third-level index blocks b 3 = ceiling(r 3 /bfr i) = ceiling(7/34) = 1
Since the third level has only one block, it is the top index level.
Hence, the index has x = 3 levels
iv. Total number of blocks for the index b i = b 1 + b 2 + b 3 = 221 + 7 + 1
= 229 blocks
v. Number of block accesses to search for a record = x + 1 = 3 + 1 = 4

(d) i. Index record size R i = (V SSN + P) = (9 + 6) = 15 bytes
Index blocking factor bfr i = (fan-out) fo = floor(B/R i) = floor(512/15)
= 34 index records per block
(This has not changed from part (c) above)
(Alternative solution: The previous solution assumes that leaf-level index blocks contain block pointers; it is also possible to assume that they contain record pointers, in which case the index record size would be V SSN + P R = 9 + 7 = 16 bytes. In this case, the calculations for leaf nodes in (i) below would then have to use R i = 16 bytes rather than R i = 15 bytes, so we get:
Index record size R i = (V SSN + P R) = (9 + 7) = 15 bytes
Leaf-level ndex blocking factor bfr i = floor(B/R i) = floor(512/16)
= 32 index records per block
However, for internal nodes, block pointers are always used so the fan-out for internal nodes fo would still be 34.)
ii. Number of first-level index entries r 1 = number of file records r = 30000
Number of first-level index blocks b 1 = ceiling(r 1 /bfr i) = ceiling(30000/34)
= 883 blocks
(Alternative solution:
Number of first-level index entries r 1 = number of file records r = 30000
Number of first-level index blocks b 1 = ceiling(r 1 /bfr i) = ceiling(30000/32)

= 938 blocks)
iii. We can calculate the number of levels as follows:
Number of second-level index entries r 2 = number of first-level index blocks b 1
= 883 entries
Number of second-level index blocks b 2 = ceiling(r 2 /bfr i) = ceiling(883/34)
= 26 blocks
Number of third-level index entries r 3 = number of second-level index blocks b 2
= 26 entries
Number of third-level index blocks b 3 = ceiling(r 3 /bfr i) = ceiling(26/34) = 1
Since the third level has only one block, it is the top index level.
Hence, the index has x = 3 levels
(Alternative solution:
Number of second-level index entries r 2 = number of first-level index blocks b 1
= 938 entries
Number of second-level index blocks b 2 = ceiling(r 2 /bfr i) = ceiling(938/34)
= 28 blocks
Number of third-level index entries r 3 = number of second-level index blocks b 2
= 28 entries
Number of third-level index blocks b 3 = ceiling(r 3 /bfr i) = ceiling(28/34) = 1
Since the third level has only one block, it is the top index level.
Hence, the index has x = 3 levels)
iv. Total number of blocks for the index b i = b 1 + b 2 + b 3 = 883 + 26 + 1 = 910
(Alternative solution:
Total number of blocks for the index b i = b 1 + b 2 + b 3 = 938 + 28 + 1 = 987)
v. Number of block accesses to search for a record = x + 1 = 3 + 1 = 4

(e) i. Index record size R i = (V DEPARTMENTCODE + P) = (9 + 6) = 15 bytes
Index blocking factor bfr i = (fan-out) fo = floor(B/R i) = floor(512/15)
= 34 index records per block
ii. There are 1000 distinct values of DEPARTMENTCODE, so the average number of records for each value is (r/1000) = (30000/1000) = 30
Since a record pointer size P R = 7 bytes, the number of bytes needed at the level of indirection for each value of DEPARTMENTCODE is 7 * 30 =210 bytes, which fits in one block. Hence, 1000 blocks are needed for the level of indirection.
iii. Number of first-level index entries r 1
= number of distinct values of DEPARTMENTCODE = 1000 entries
Number of first-level index blocks b 1 = ceiling(r 1 /bfr i) = ceiling(1000/34)
= 30 blocks
iv. We can calculate the number of levels as follows:
Number of second-level index entries r 2 = number of first-level index blocks b 1
= 30 entries
Number of second-level index blocks b 2 = ceiling(r 2 /bfr i) = ceiling(30/34) = 1
Hence, the index has x = 2 levels
v. total number of blocks for the index b i = b 1 + b 2 + b indirection
= 30 + 1 + 1000 = 1031 blocks
vi. Number of block accesses to search for and retrieve the block containing the record pointers at the level of indirection = x + 1 = 2 + 1 = 3 block accesses
If we assume that the 30 records are distributed over 30 distinct blocks, we need an additional 30 block accesses to retrieve all 30 records. Hence, total block accesses needed on average to retrieve all the records with a given value for DEPARTMENTCODE = x + 1 + 30 = 33

(f) i. Index record size R i = (V DEPARTMENTCODE + P) = (9 + 6) = 15 bytes

Index blocking factor bfr i = (fan-out) fo = floor(B/R i) = floor(512/15)
= 34 index records per block
ii. Number of first-level index entries r 1
= number of distinct DEPARTMENTCODE values= 1000 entries
Number of first-level index blocks b 1 = ceiling(r 1 /bfr i)
= ceiling(1000/34) = 30 blocks
iii. We can calculate the number of levels as follows:
Number of second-level index entries r 2 = number of first-level index blocks b 1
= 30 entries
Number of second-level index blocks b 2 = ceiling(r 2 /bfr i) = ceiling(30/34) = 1
Since the second level has one block, it is the top index level.
Hence, the index has x = 2 levels
iv. Total number of blocks for the index b i = b 1 + b 2 = 30 + 1 = 31 blocks
v. Number of block accesses to search for the first block in the cluster of blocks
= x + 1 = 2 + 1 = 3
The 30 records are clustered in ceiling(30/bfr) = ceiling(30/4) = 8 blocks.
Hence, total block accesses needed on average to retrieve all the records with a given DEPARTMENTCODE = x + 8 = 2 + 8 = 10 block accesses

(g) i. For a B + -tree of order p, the following inequality must be satisfied for each internal tree node: (p * P) + ((p - 1) * V SSN) < B, or
(p * 6) + ((p - 1) * 9) < 512, which gives 15p < 521, so p=34
For leaf nodes, assuming that record pointers are included in the leaf nodes, the following inequality must be satisfied: (p leaf * (V SSN +P R)) + P < B, or
(p leaf * (9+7)) + 6 < 512, which gives 16p leaf < 506, so p leaf =31
ii. Assuming that nodes are 69% full on the average, the average number of key values in a leaf node is 0.69*p leaf = 0.69*31 = 21.39. If we round this up for convenience, we get 22 key values (and 22 record pointers) per leaf node. Since the file has 30000 records and hence 30000 values of SSN, the number of leaf-level nodes (blocks) needed is b 1 = ceiling(30000/22) = 1364 blocks
iii. We can calculate the number of levels as follows:
The average fan-out for the internal nodes (rounded up for convenience) is
fo = ceiling(0.69*p) = ceiling(0.69*34) = ceiling(23.46) = 24
number of second-level tree blocks b 2 = ceiling(b 1 /fo) = ceiling(1364/24)
= 57 blocks
number of third-level tree blocks b 3 = ceiling(b 2 /fo) = ceiling(57/24)= 3
number of fourth-level tree blocks b 4 = ceiling(b 3 /fo) = ceiling(3/24) = 1
Since the fourth level has only one block, the tree has x = 4 levels (counting the leaf level). Note: We could use the formula:
x = ceiling(log fo (b 1)) + 1 = ceiling(log 24 1364) + 1 = 3 + 1 = 4 levels
iv. total number of blocks for the tree b i = b 1 + b 2 + b 3 + b 4
= 1364 + 57 + 3 + 1 = 1425 blocks
v. number of block accesses to search for a record = x + 1 = 4 + 1 = 5

17.19 - A PARTS file with Part# as key field includes records with the following Part# values: 23, 65, 37, 60, 46, 92, 48, 71, 56, 59, 17, 21, 10, 74, 78, 15, 16, 20, 24, 28, 39, 43, 47, 50, 69, 75, 8, 49, 33, 38. Suppose the search field values are inserted in the given order in a B + -tree of order p=4 and p leaf =3; show how the tree will expand and what the final tree looks like.

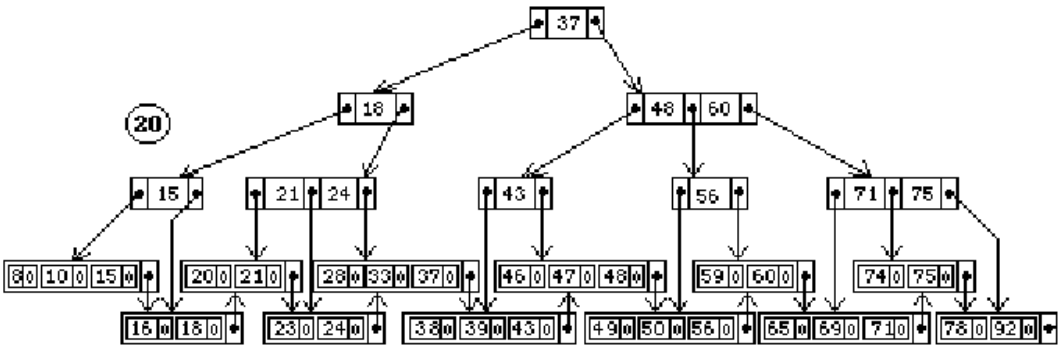
Answer:

A B + -tree of order p=4 implies that each internal node in the tree (except possibly the root) should have at least 2 keys (3 pointers) and at most 4 pointers. For p leaf =3, leaf nodes must have at least 2 keys and at most 3 keys. The figure on page 50 shows how the tree progresses as the keys are inserted. We will only show a new tree when insertion causes a split of one of the leaf nodes, and then show how the split propagates up the tree.
Hence, step 1 below shows the tree after insertion of the first 3 keys 23, 65, and 37, and before inserting 60 which causes overflow and splitting. The trees given below show how the keys are inserted in order. Below, we give the keys inserted for each tree:
1 :23, 65, 37; 2:60; 3:46; 4:92; 5:48, 71; 6:56; 7:59, 17; 8:21; 9:10; 10:7 4 ;
11:78; 12:15; 13:16; 14:20; 15:24; 16:28, 39; 17:43, 47; 17:50, 69; 19:7 5 ;
20:8, 49, 33, 38;

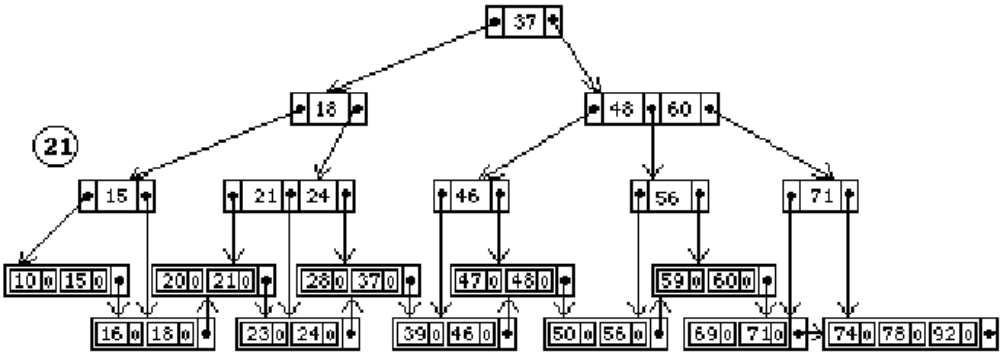
17.20: No solution provided.

17.21 - Suppose that the following search field values are deleted, in the given order, from the B + -tree of Exercise 17.19, show how the tree will shrink and show the final tree. The deleted values are: 65, 75, 43, 17, 20, 92, 59, 37.

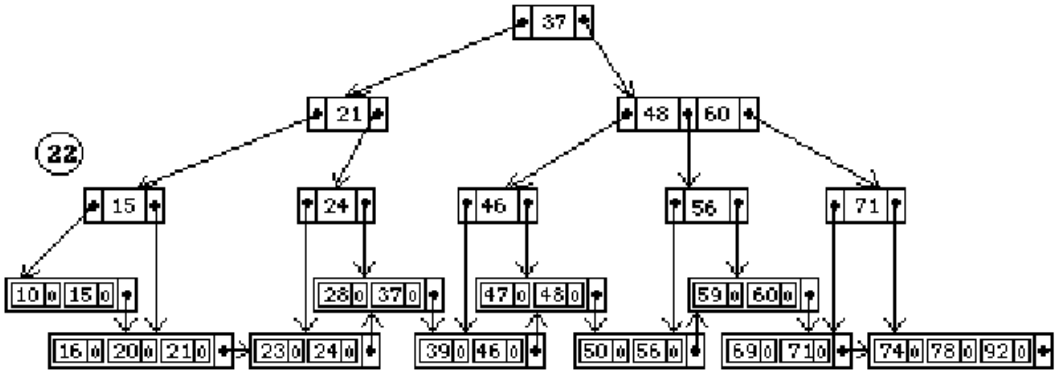
Answer:
An important note about a deletion algorithm for a B + -tree is that deletion of a key value from a leaf node will result in a reorganization of the tree if: (i) The leaf node is less than half full; in this case, we will combine it with the next leaf node (other algorithms combine it with either the next or the previous leaf nodes, or both), (ii) If the key value deleted is the rightmost (last) value in the leaf node, in which case its value will appear in an internal node; in this case, the key value to the left of the deleted key in the left node replaces the deleted key value in the internal node. Following is what happens to the tree number 19 after the specified deletions (not tree number 20):
Deleting 65 will only affect the leaf node. Deleting 75 will cause a leaf node to be less than half full, so it is combined with the next node; also, 75 is removed from the internal node leading to the following tree:



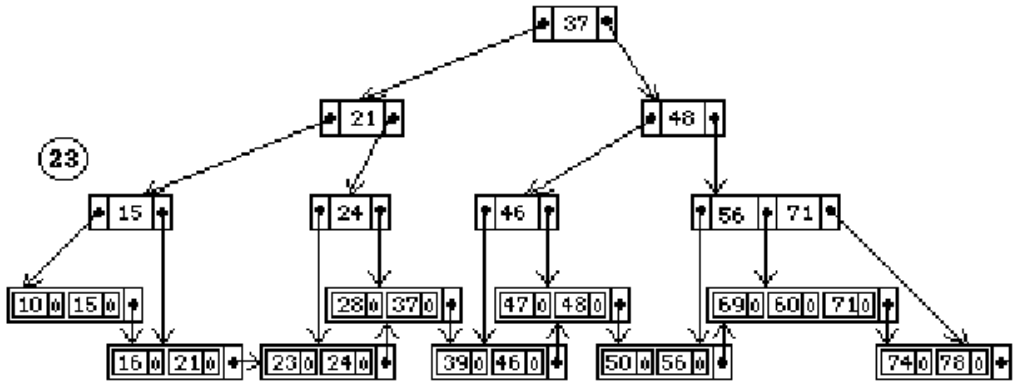
Deleting 43 causes a leaf node to be less than half full, and it is combined with the next node. Since the next node has 3 entries, its rightmost (first) entry 46 can replace 43 in both the leaf and internal nodes, leading to the following tree:



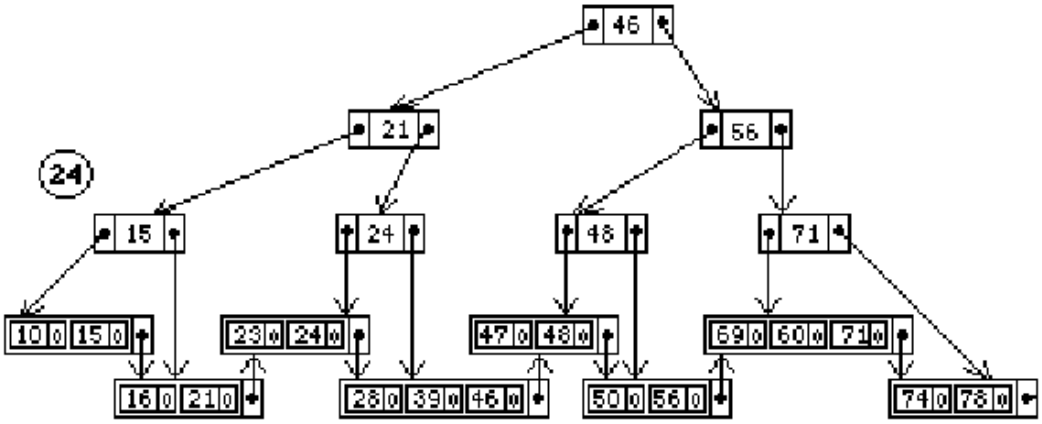
Next, we delete 17, which is a rightmost entry in a leaf node and hence appears in an internal node of the B + -tree. The leaf node is now less than half full, and is combined with the next node. The value 17 must also be removed from the internal node, causing underflow in the internal node. One approach for dealing with underflow in internal nodes is to reorganize the values of the underflow node with its child nodes, so 21 is moved up into the underflow node leading to the following tree:



Deleting 20 and 92 will not cause underflow. Deleting 59 causes underflow, and the remaining value 60 is combined with the next leaf node. Hence, 60 is no longer a rightmost entry in a leaf node and must be removed from the internal node. This is normally done by moving 56 up to replace 60 in the internal node, but since this leads to underflow in the node that used to contain 56, the nodes can be reorganized as follows:



Finally, removing 37 causes serious underflow, leding to a reorganization of the whole tree. One approach to deleting the value in the root node is to use the rightmost value in the next leaf node (the first leaf node in the right subtree) to replace the root, and move this leaf node to the left subtree. In this case, the resulting tree may look as follows:



17.22 - 17.28: No solutions provided.

CHAPTER 18: Strategies for Query Processing

Answers to Selected Exercises

18.13 - Consider SQL queries Q1, Q8, Q1B, Q4, Q27 in Chapter 5.

(a) Draw at least two query trees that can represented each of these queries.

Under what circumstances would you use each of your query trees?

(b) Draw the initial query tree for each of these queries; then show how the query tree is optimized by the algorithm outlined in section 18.7.

(c) For each query, compare your on query trees of part (a) and the initial and final query trees of part (b).

Answer:
Below are possible answers for Q8 and Q27.
Q8: SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME
FROM EMPLOYEE E, EMPLOYEE S
WHERE E.SUPERSSN = S.SSN
Q27: SELECT FNAME, LNAME, 1.1*SALARY
FROM EMPLOYEE, WORKS_ON, PROJECT
WHERE SSN = ESSN AND PNO = PNUMBER AND PNAME = 'ProductX'
Q8's tree1:
PROJECT E.FNAME, E.LNAME, S.FNAME, S.LNAME
E.SUPERSSN=S.SSN JOIN
EMPLOYEE E EMPLOYEE S
Q8'S tree2:
PROJECT
CARTESIAN PRODUCT
EMPLOYEE E EMPLOYEE S
E.FNAME, E.LNAME, S.FNAME, S.LNAME
SELECT E.SUPERSSN=S.SSN
The initial query tree for Q8 is the same as tree2 above; the only change made by the optimization algorithm is to replace the selection and Cartesian product by the join in tree1. Thus, tree 1 is the result after optimization.
Q27's tree1:
PROJECT FNAME, LNAME, SALARY
PNO=PNUMBER JOIN
EMPLOYEE PROJECT
SSN=ESSN JOIN SELECT PNAME="ProductX"
WORKS_ON
Q27's tree2:
PROJECT FNAME, LNAME, SALARY
PNO=PNUMBER AND SSN=ESSN AND PNAME="ProductX" SELECT
EMPLOYEE
PROJECT
CARTESIAN PRODUCT
WORKS_ON
CARTESIAN PRODUCT
The initial query tree of Q27 is tree2 above, but the result of the heuristic optimization process will NOT be the same as tree1 in this case. It can be optimized more thoroughly, as follows:
PROJECT FNAME, LNAME, SALARY
PNO=PNUMBER JOIN EMPLOYEE
PROJECT
SSN=ESSN JOIN
SELECT PNAME="ProductX" WORKS_ON
The reason is that the leaf nodes could be arranged (in Step 3 of the algorithm outlined on page 613) so that the more restrictive selects are executed first.

18.14 - A file of 4096 blocks is to be sorted with an available buffer space of 64 blocks. How many passes will be needed in the merge phase of the external sort-merge algorithm?

Answer:

We first need to compute the number of runs, n , in the merge phase.
R
Using the formula in the text, we have

$$n_R = \left\lceil \frac{b}{n_B} \right\rceil$$

where b = 4096 (the number of blocks in the file)

$n_B = 64$ (buffer blocks)

so $n_R = 64$ (number of sorted runs on disk)

In the merge phase , the number of passes is dependent on the degree of merging, d_M

where $d_M = MIN(n_B - 1, n_R) = 63$

$$\text{The number of passes} = \left\lceil \log_{d_M} (n_R) \right\rceil = 2$$

$$= \left\lceil \log_{63} 64 \right\rceil = 2$$

18.15 - Develop cost functions for the PROJECT, UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT algorithms discussed in section 18.4.

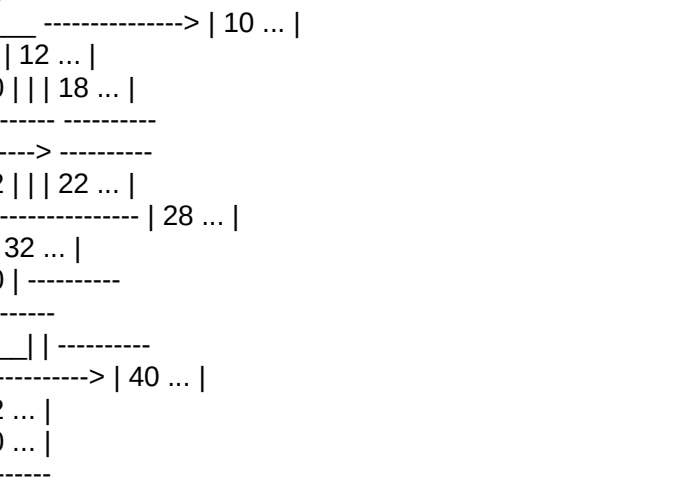
Answer:

Assume relations R and S are stored in b R and b S disk blocks, respectively. Also, assume that the file resulting from the operation is stored in b RESULT disk blocks (if the size cannot be otherwise determined).
PROJECT operation: if <attribute list> includes a key of R, then the cost is 2*b R since the read–in and write–out files have the same size, which is the size of R itself; if <attribute list> does not include a key of R, then we must sort the intermediate result file before eliminating duplicates, which costs another scan; thus the latter cost is 3*b R + k*b R *log 2 b R (assuming PROJECT is implemented by sort and eliminate duplicates).
SET OPERATIONS (UNION, DIFFERENCE, INTERSECTION): According to the algorithms where the usual sort and scan/merge of each table is done, the cost of any of these is: k*[(b R *log 2 b R) + (b S *log 2 b S)] + b R + b S + b RESULT
CARTESIAN PRODUCT: The join selectivity of Cartesian product is js = 1, and the typical way of doing it is the nested loop method, since there are no conditions to match. We first assume two memory buffers; the other quantities are as discussed for the JOIN analysis in Section 18.2.3. We get:
J1: C R X S = b R + (b R *b S) + (|R|*|S|)/bfr RS
Next, suppose we have n B memory buffers, as in Section 16.1.2. Assume file R is smaller and is used in the outer loop. We get:
J1': C R X S = b R + ceiling(b R /(n B - 1)) * b S)+ (|R|*|S|)/bfr RS , which is better than J1, per its middle term.

18.16 - No solution provided.

18.17 - Can a nondense (sparse) index be used in the implementation of an aggregate operator? Why or why not?

Answer:
A nondense (sparse) index contains entries for only some of the search values. A primary index is an example of a nondense index which includes an entry for each disk block of the data file rather than for every record.



If the keys in the index correspond to the smallest key value in the data block then the sparse index could be used to compute the MIN function. However, the MAX function could not be determined from the index. In addition, since all values do not appear in the index, AVG, SUM and COUNT could not be determined from just the index.

18.18 - Calculate the cost functions for different options of executing the JOIN operation OP7 discussed in section 18.3.2.

Answer:
The operation is
OP7: DEPARTMENT |x| MGRSSN=SSN EMPLOYEE.
As in section 18.2.3 we assume the secondary index on MGRSSN of DEPARTMENT, with selection cardinality s=1 and level x=1; also the join selectivity of OP7 is js = 1/125 = 1/|DEPARTMENT|, because MGRSSN acts as a key of the DEPARTMENT table. (Note: There is exactly one manager per department.)
Finally, we assume the same blocking factor as the OP6 join of these two tables:
bfr = 4 records/block, as the results involve the same number of attributes. Thus the applicable methods are J1 and J2 with either table as the outer loop; the quantities parallel those of OP6:
J1 with EMPLOYEE as outer loop:
CJ1 = 2000 + (2000*13) + (((1/125)*10000*125)/4) = 30,500
J1 with DEPARTMENT as outer loop:
CJ1' = 13 + (13*2000) + (((1/125)*10000*125)/4) = 28,513
J2 with EMPLOYEE as outer loop, and MGRSSN as secondary key for S:
[EMPLOYEE as outer loop and primary index on SSN gives the same result.]
CJ2a = b R + (|R|*(x S + s)) + ((js*|R|*|S|)/bfr
= 2000 + (2000*(1+1)) + (((1/125)*10000*125)/4)
= 24,500
J2 with DEPARTMENT as outer loop:
CJ2c = b S + (|S|*(x R + 1)) + ((js*|R|*|S|)/bfr
= 13 + (125*2) + (((1/125)*10000*125)/4)
= 13 + 250 + 2500 = 2,763 [!]
Obviously this optimization was worthwhile, to get the latter minimum.

18.18 – 18.20: No solution provided.

15.21 - Extend the sort-merge join algorithm to implement the LEFT OUTER JOIN operation.

Answer:
The left outer join of R and S would produce the rows from R and S that join as well as the rows from R that do not join any row from S. The sort-merge join algorithm would only need minor modifications during the merge phase. During the first step, relation R would be sorted on the join column(s).
During the second step, relation S would be sorted on the join columns(s).
During the third step, the sorted R relation and S relation would be merged. That is, rows would be combined if the R row and S row have the same value for the join column(s).
In addition, if no matching S row was found for an R row, then that R row would also be placed in the left outer join result, except that the corresponding attributes from the S relation would be set to NULL values. (How a null value is to be represented is a matter of choice out of the options provided in a DBMS.)

18.22 - Compare the cost of two different query plans for the following query:

salary > 40000 select (EMPLOYEE |X| DNO=DNUMBER DEPARTMENT)
Use the database statistics in Figure 15.8

Answer:
One plan might be for the following query tree

We can use the salary index on EMPLOYEE for the select operation:
The table in Figure 18.8(a) indicates that there are 500 unique salary values, with a low value of 1 and a high value of 500. (It might be in reality that salary is in units of 1000 dollars, so 1 represents $1000 and 500 represents $500,000.)
The selectivity for (Salary > 400) can be estimated as
(500 - 400)/500 = 1/5
This assumes that salaries are spread evenly across employees.
So the cost (in block accesses) of accessing the index would be
Blevel + (1/5) * (LEAF BLOCKS) = 1 + (1/5)* 50 = 11
Since the index is nonunique, the employees can be stored on any of the data blocks.
So the the number of data blocks to be accessed would be
(1/5) * (NUM_ROWS) = (1/5) * 10,000 = 2000
Since 10,000 rows are stored in 2000 blocks, we have that
2000 rows can be stored in 400 blocks. So the TEMPORARY table (i.e., the result of the selection operator) would contain 400 blocks.
The cost of writing the TEMPORARY table to disk would be
400 blocks.
Now, we can do a nested loop join of the temporary table and the DEPARTMENT table. The cost of this would be
b + (b * b)
DEPARTMENT DEPARTMENT TEMPORARY
We can ignore the cost of writing the result for this comparision, since the cost would be the same for both plans, we will consider.
We have 5 + (5 * 400) = 2005 block accesses
18.22 (continued)
Therefore, the total cost would be
11 + 2000 + 400 + 2005 = 4416 block accesses
NOTE: If we have 5 main memory buffer pages available during the join, then we could store all 5 blocks of the DEPARTMENT table there. This would reduce the cost of the join to 5 + 400 = 405 and the total cost would be reduced to 11 + 2000 + 400 + 405 = 2816. A second plan might be for the following query tree

Again, we could use a nested loop for the join but instead of creating a temporary table for the result we can use a pipelining approach and pass the joining rows to the select operator as they are computed. Using a nested loop join algorithm would yield the following 50 + (50 * 2000) = 100,050 blocks We would pipeline the result to the selection operator and it would choose only those rows whose salary value was greater than 400.
NOTE: If we have 50 main memory buffer pages available during the join, then we could store the entire DEPARTMENT table there. This would reduce the cost of the join and the pipelined select to 50 + 2000 = 2050.

CHAPTER 20: INTRODUCTION TO TRANSACTION PROCESSING CONCEPTS and THEORY

Answers to Selected Exercises

20.14 - Change transaction T 2 in Figure 20.2b to read:
read_item(X);
X:= X+M;
if X > 90 then exit
else write_item(X);

Discuss the final result of the different schedules in Figure 20.3 (a) and (b), where M = 2 and N = 2, with respect to the following questions. Does adding the above condition change the final outcome? Does the outcome obey the implied consistency rule (that the capacity of X is 90)?

Answer:
The above condition does not change the final outcome unless the initial value of X > 88. The outcome, however, does obey the implied consistency rule that X < 90, since the value of X is not updated if it becomes greater than 90.

20.15 - Repeat Exercise 20.14 adding a check in T 1 so that Y does not exceed 90.

Answer:
T1 T2
read_item(X);
X := X-N;

read_item(X);
X := X+M;
write_item(X);
read_item(Y);
if X > 90 then exit
else write_item(X);
Y := Y+N;
if Y> 90 then
exit
else write_item(Y);
The above condition does not change the final outcome unless the initial value of X > 88 or Y > 88. The outcome obeys the implied consistency rule that X < 90 and Y < 90.

20.16 - Add the operation commit at the end of each of the transactions T 1 and T 2 from Figure 20.2; then list all possible schedules for the modified transactions.
Determine which of the schedules are recoverable, which are cascadeless, and which are strict.

Answer:
T 1 T 2
read_item(X); read_item(X);
X := X - N X := X + M;
write_item(X); write_item(X);
read_item(Y); commit T 2
Y := Y + N;
write_item(Y);
commit T 1

The transactions can be written as follows using the shorthand notation:
T 1 : r 1 (X); w 1 (X); r 1 (Y); w 1 (Y); C 1 ;
T 2 : r 2 (X); w 2 (X); C 2 ;

In general, given m transactions with number of operations n1, n2, ..., nm, the number of possible schedules is: (n1 + n2 + ... + nm)! / (n1! * n2! * ... * nm!), where ! is the factorial function. In our case, m =2 and n1 = 5 and n2 = 3, so the number of possible schedules is:
(5+3)! / (5! * 3!) = 8*7*6*5*4*3*2*1/ 5*4*3*2*1*3*2*1 = 56.

Below are the 56 possible schedules, and the type of each schedule:
S 1 : r 1 (X); w 1 (X); r 1 (Y); w 1 (Y); C 1 ; r 2 (X); w 2 (X); C 2 ; strict (and hence cascadeless)
S 2 : r 1 (X); w 1 (X); r 1 (Y); w 1 (Y); r 2 (X); C 1 ; w 2 (X); C 2 ; recoverable
S 3 : r 1 (X); w 1 (X); r 1 (Y); w 1 (Y); r 2 (X); w 2 (X); C 1 ; C 2 ; recoverable
S 4 : r 1 (X); w 1 (X); r 1 (Y); w 1 (Y); r 2 (X); w 2 (X); C 2 ; C 1 ; non-recoverable
S 5 : r 1 (X); w 1 (X); r 1 (Y); r 2 (X); w 1 (Y); C 1 ; w 2 (X); C 2 ; recoverable
S 6 : r 1 (X); w 1 (X); r 1 (Y); r 2 (X); w 1 (Y); w 2 (X); C 1 ; C 2 ; recoverable
S 7 : r 1 (X); w 1 (X); r 1 (Y); r 2 (X); w 1 (Y); w 2 (X); C 2 ; C 1 ; non-recoverable
S 8 : r 1 (X); w 1 (X); r 1 (Y); r 2 (X); w 2 (X); w 1 (Y); C 1 ; C 2 ; recoverable
S 9 : r 1 (X); w 1 (X); r 1 (Y); r 2 (X); w 2 (X); w 1 (Y); C 2 ; C 1 ; non-recoverable
S 10 : r 1 (X); w 1 (X); r 1 (Y); r 2 (X); w 2 (X); C 2 ; w 1 (Y); C 1 ; non-recoverable
S 11 : r 1 (X); w 1 (X); r 2 (X); r 1 (Y); w 1 (Y); C 1 ; w 2 (X); C 2 ; recoverable

S 12 : r 1 (X); w 1 (X); r 2 (X); r 1 (Y); w 1 (Y); w 2 (X); C 1 ; C 2 ; recoverable
S 13 : r 1 (X); w 1 (X); r 2 (X); r 1 (Y); w 1 (Y); w 2 (X); C 2 ; C 1 ; non-recoverable
S 14 : r 1 (X); w 1 (X); r 2 (X); r 1 (Y); w 2 (X); w 1 (Y); C 1 ; C 2 ; recoverable
S 15 : r 1 (X); w 1 (X); r 2 (X); r 1 (Y); w 2 (X); w 1 (Y); C 2 ; C 1 ; non-recoverable
S 16 : r 1 (X); w 1 (X); r 2 (X); r 1 (Y); w 2 (X); C 2 ; w 1 (Y); C 1 ; non-recoverable
S 17 : r 1 (X); w 1 (X); r 2 (X); w 2 (X); r 1 (Y); w 1 (Y); C 1 ; C 2 ; recoverable
S 18 : r 1 (X); w 1 (X); r 2 (X); w 2 (X); r 1 (Y); w 1 (Y); C 2 ; C 1 ; non-recoverable
S 19 : r 1 (X); w 1 (X); r 2 (X); w 2 (X); r 1 (Y); C 2 ; w 1 (Y); C 1 ; non-recoverable
S 20 : r 1 (X); w 1 (X); r 2 (X); w 2 (X); C 2 ; r 1 (Y); w 1 (Y); C 1 ; non-recoverable
S 20 : r 1 (X); r 2 (X); w 1 (X); r 1 (Y); w 1 (Y); C 1 ; w 2 (X); C 2 ; strict (and hence cascadeless)
S 22 : r 1 (X); r 2 (X); w 1 (X); r 1 (Y); w 1 (Y); w 2 (X); C 1 ; C 2 ; cascadeless
S 23 : r 1 (X); r 2 (X); w 1 (X); r 1 (Y); w 1 (Y); w 2 (X); C 2 ; C 1 ; cascadeless
S 24 : r 1 (X); r 2 (X); w 1 (X); r 1 (Y); w 2 (X); w 1 (Y); C 1 ; C 2 ; cascadeless
S 25 : r 1 (X); r 2 (X); w 1 (X); r 1 (Y); w 2 (X); w 1 (Y); C 2 ; C 1 ; cascadeless
S 26 : r 1 (X); r 2 (X); w 1 (X); r 1 (Y); w 2 (X); C 2 ; w 1 (Y); C 1 ; cascadeless
S 27 : r 1 (X); r 2 (X); w 1 (X); w 2 (X); r 1 (Y); w 1 (Y); C 1 ; C 2 ; cascadeless
S 28 : r 1 (X); r 2 (X); w 1 (X); w 2 (X); r 1 (Y); w 1 (Y); C 2 ; C 1 ; cascadeless
S 29 : r 1 (X); r 2 (X); w 1 (X); w 2 (X); r 1 (Y); C 2 ; w 1 (Y); C 1 ; cascadeless
S 30 : r 1 (X); r 2 (X); w 1 (X); w 2 (X); C 2 ; r 1 (Y); w 1 (Y); C 1 ; cascadeless
S 31 : r 1 (X); r 2 (X); w 2 (X); w 1 (X); r 1 (Y); w 1 (Y); C 1 ; C 2 ; cascadeless
S 32 : r 1 (X); r 2 (X); w 2 (X); w 1 (X); r 1 (Y); w 1 (Y); C 2 ; C 1 ; cascadeless
S 33 : r 1 (X); r 2 (X); w 2 (X); w 1 (X); r 1 (Y); C 2 ; w 1 (Y); C 1 ; cascadeless
S 34 : r 1 (X); r 2 (X); w 2 (X); w 1 (X); C 2 ; r 1 (Y); w 1 (Y); C 1 ; cascadeless
S 35 : r 1 (X); r 2 (X); w 2 (X); C 2 ; w 1 (X); r 1 (Y); w 1 (Y); C 1 ; strict (and hence cascadeless)
S 36 : r 2 (X); r 1 (X); w 1 (X); r 1 (Y); w 1 (Y); C 1 ; w 2 (X); C 2 ; strict (and hence cascadeless)
S 37 : r 2 (X); r 1 (X); w 1 (X); r 1 (Y); w 1 (Y); w 2 (X); C 1 ; C 2 ; cascadeless
S 38 : r 2 (X); r 1 (X); w 1 (X); r 1 (Y); w 1 (Y); w 2 (X); C 2 ; C 1 ; cascadeless
S 39 : r 2 (X); r 1 (X); w 1 (X); r 1 (Y); w 2 (X); w 1 (Y); C 1 ; C 2 ; cascadeless
S 40 : r 2 (X); r 1 (X); w 1 (X); r 1 (Y); w 2 (X); w 1 (Y); C 2 ; C 1 ; cascadeless
S 41 : r 2 (X); r 1 (X); w 1 (X); r 1 (Y); w 2 (X); C 2 ; w 1 (Y); C 1 ; cascadeless
S 42 : r 2 (X); r 1 (X); w 1 (X); w 2 (X); r 1 (Y); w 1 (Y); C 1 ; C 2 ; cascadeless
S 43 : r 2 (X); r 1 (X); w 1 (X); w 2 (X); r 1 (Y); w 1 (Y); C 2 ; C 1 ; cascadeless
S 44 : r 2 (X); r 1 (X); w 1 (X); w 2 (X); r 1 (Y); C 2 ; w 1 (Y); C 1 ; cascadeless
S 45 : r 2 (X); r 1 (X); w 1 (X); w 2 (X); C 2 ; r 1 (Y); w 1 (Y); C 1 ; cascadeless
S 46 : r 2 (X); r 1 (X); w 2 (X); w 1 (X); r 1 (Y); w 1 (Y); C 1 ; C 2 ; cascadeless
S 47 : r 2 (X); r 1 (X); w 2 (X); w 1 (X); r 1 (Y); w 1 (Y); C 2 ; C 1 ; cascadeless
S 48 : r 2 (X); r 1 (X); w 2 (X); w 1 (X); r 1 (Y); C 2 ; w 1 (Y); C 1 ; cascadeless
S 49 : r 2 (X); r 1 (X); w 2 (X); w 1 (X); C 2 ; r 1 (Y); w 1 (Y); C 1 ; cascadeless
S 50 : r 2 (X); r 1 (X); w 2 (X); C 2 ; w 1 (X); r 1 (Y); w 1 (Y); C 1 ; cascadeless
S 51 : r 2 (X); w 2 (X); r 1 (X); w 1 (X); r 1 (Y); w 1 (Y); C 1 ; C 2 ; non-recoverable
S 52 : r 2 (X); w 2 (X); r 1 (X); w 1 (X); r 1 (Y); w 1 (Y); C 2 ; C 1 ; recoverable
S 53 : r 2 (X); w 2 (X); r 1 (X); w 1 (X); r 1 (Y); C 2 ; w 1 (Y); C 1 ; recoverable
S 54 : r 2 (X); w 2 (X); r 1 (X); w 1 (X); C 2 ; r 1 (Y); w 1 (Y); C 1 ; recoverable
S 55 : r 2 (X); w 2 (X); r 1 (X); C 2 ; w 1 (X); r 1 (Y); w 1 (Y); C 1 ; recoverable
S 56 : r 2 (X); w 2 (X); C 2 ; r 1 (X); w 1 (X); r 1 (Y); w 1 (Y); C 1 ; strict (and hence cascadeless)

20.17 - List all possible schedules for transactions T 1 and T 2 from figure 20.2, and determine which are conflict serializable (correct) and which are not.

Answer:
T 1 T 2
read_item(X); read_item(X);
X := X - N X := X + M;
write_item(X); write_item(X);
read_item(Y);
Y := Y + N;
write_item(Y);

The transactions can be written as follows using shorthand notation:
T 1 : r 1 (X); w 1 (X); r 1 (Y); w 1 (Y);
T 2 : r 2 (X); w 2 (X);

In this case, m =2 and n1 = 4 and n2 = 2, so the number of possible schedules is:
(4+2)! / (4! * 2!) = 6*5*4*3*2*1/ 4*3*2*1*2*1 = 15.

Below are the 15 possible schedules, and the type of each schedule:
S 1 : r 1 (X); w 1 (X); r 1 (Y); w 1 (Y); r 2 (X); w 2 (X); serial (and hence also serializable)
S 2 : r 1 (X); w 1 (X); r 1 (Y); r 2 (X); w 1 (Y); w 2 (X); (conflict) serializable
S 3 : r 1 (X); w 1 (X); r 1 (Y); r 2 (X); w 2 (X); w 1 (Y); (conflict) serializable
S 4 : r 1 (X); w 1 (X); r 2 (X); r 1 (Y); w 1 (Y); w 2 (X); (conflict) serializable
S 5 : r 1 (X); w 1 (X); r 2 (X); r 1 (Y); w 2 (X); w 1 (Y); (conflict) serializable
S 6 : r 1 (X); w 1 (X); r 2 (X); w 2 (X); r 1 (Y); w 1 (Y); (conflict) serializable
S 7 : r 1 (X); r 2 (X); w 1 (X); r 1 (Y); w 1 (Y); w 2 (X); not (conflict) serializable
S 8 : r 1 (X); r 2 (X); w 1 (X); r 1 (Y); w 2 (X); w 1 (Y); not (conflict) serializable
S 9 : r 1 (X); r 2 (X); w 1 (X); w 2 (X); r 1 (Y); w 1 (Y); not (conflict) serializable
S 10 : r 1 (X); r 2 (X); w 2 (X); w 1 (X); r 1 (Y); w 1 (Y); not (conflict) serializable
S 11 : r 2 (X); r 1 (X); w 1 (X); r 1 (Y); w 1 (Y); w 2 (X); not (conflict) serializable
S 12 : r 2 (X); r 1 (X); w 1 (X); r 1 (Y); w 2 (X); w 1 (Y); not (conflict) serializable
S 13 : r 2 (X); r 1 (X); w 1 (X); w 2 (X); r 1 (Y); w 1 (Y); not (conflict) serializable
S 14 : r 2 (X); r 1 (X); w 2 (X); w 1 (X); r 1 (Y); w 1 (Y); not (conflict) serializable
S 15 : r 2 (X); w 2 (X); r 1 (X); w 1 (X); r 1 (Y); w 1 (Y); serial (and hence also serializable)

20.18 - How many *serial* schedules exist for the three transactions in Figure 20.8 (a)? What are they? What is the total number of possible schedules?

Partial Answer:
T1 T2 T3 T3 T2 T1
T2 T3 T1 T2 T1 T3
T3 T1 T2 T1 T3 T2

Total number of serial schedules for the three transactions = 6
In general, the number of serial schedules for n transactions is n! (i.e. factorial(n))

20.19 - No solution provided.

20.20 - Why is an explicit transaction end statement needed in SQL but not an explicit begin statement?

Answer:
A transaction is an atomic operation. It has only one way to begin, that is, with "Begin Transaction" command but it could end up in two ways: Successfully installs its updates to the database (i.e., commit) or Removes its partial updates (which may be incorrect) from the database (abort). Thus, it is important for the database systems to identify the right way of ending a transaction. It is for this reason an "End" command is needed in SQL2 query.

20.20 Describe situations where each of the different isolation levels would be useful for transaction processing.

Answer:
Transaction isolation levels provide a measure of the influence of other concurrent transactions on a given transaction. This affects the level of concurrency, that is, the level of concurrency is the highest in Read Uncommitted and the lowest in Serializable.

Isolation level Serializable: This isolation level preserves consistency in all situations, thus it is the safest execution mode. It is recommended for execution environment where every update is crucial for a correct result. For example, airline reservation, debit credit, salary increase, and so on.

Isolation level Repeatable Read: This isolation level is similar to Serializable except Phantom problem may occur here. Thus, in record locking (finer granularity), this isolation level must be avoided. It can be used in all types of environments, except in the environment where accurate summary information (e.g., computing total sum of a all different types of account of a bank customer) is desired.

Isolation level Read Committed: In this isolation level a transaction may see two different values of the same data items during its execution life. A transaction in this level applies write lock and keeps it until it commits. It also applies a read (shared) lock but the lock is released as soon as the data item is read by the transaction. This isolation level may be used for making balance, weather, departure or arrival times, and so on.

Isolation level Read Uncommitted: In this isolation level a transaction does not either apply a shared lock or a write lock. The transaction is not allowed to write any data item, thus it may give rise to dirty read, unrepeatable read, and phantom. It may be used in the environment where statistical average of a large number of data is required.

20.22 - Which of the following schedules is (conflict) serializable? For each serializable schedule, determine the equivalent serial schedules.

(a) r1 (X); r3 (X); w1(X); r2(X); w3(X)

(b) r1 (X); r3 (X); w3(X); w1(X); r2(X)

(c) r3 (X); r2 (X); w3(X); r1(X); w1(X)

(d) r3 (X); r2 (X); r1(X); w3(X); w1(X)

Answer:
Let there be three transactions T1, T2, and T3. They are executed concurrently and produce a schedule S. S is serializable if it can be reproduced as at least one serial schedule (T1 □□T2
□□T3 or T1 □□T3 □□T2 or T2 □□T1 □□T3 or T2 □□T3 □□T1 or T3 □□T1 □□T2 or T3 □□T2
□□T1).

(a) This schedule is not serializable because T1 reads X (r1(X)) before T3 but T3 reads X (r3(X)) before T1 writes X (w1(X)), where X is a common data item. The operation r2(X) of T2 does not affect the schedule at all so its position in the schedule is irrelevant. In a serial schedule T1, T2, and T3, the operation w1(X) comes after r3(X), which does not happen in the question.

(b) This schedule is not serializable because T1 reads X (r1(X)) before T3 but T3 writes X (w3(X)) before T1 writes X (w1(X)). The operation r2(X) of T2 does not affect the schedule at all so its position in the schedule is irrelevant. In a serial schedule T1, T3, and T2, r3(X) and w3(X) must come after w1(X), which does not happen in the question.

(c) This schedule is **serializable** because all conflicting operations of T3 happens before all conflicting operation of T1. T2 has only one operation, which is a read on X (r2(X)), which does not conflict with any other operation. Thus this serializable schedule is equivalent to r2(X); r3(X); w3(X); r1(X); w1(X) (e.g., T2 □□T3 □□T1) serial schedule.

(d) This is not a serializable schedule because T3 reads X (r3(X)) before T1 reads X (r1(X)) but r1(X) happens before T3 writes X (w3(X)). In a serial schedule T3, T2, and T1, r1(X) will happen after w3(X), which does not happen in the question.

20.23 - Consider the three transactions T1, T2, and T3, and the schedules S1 and S2 given below. Draw the serializibility (precedence) graphs for S1 and S2 and state whether each schedule is serializable or not. If a schedule is serializable, write down the equivalent serial schedule(s).

T1: r1(x); r1(z); w1(x)
T2: r2(z); r2(y); w2(z); w2(y)
T3: r3(x); r3(y); w3(y)
S1: r1(x); r2(z); r1(x); r3(x); r3(y); w1(x); w3(y); r2(y); w2(z); w2(y)
S2: r1(x); r2(z); r3(x); r1(z); r2(y); r3(y); w1(x); w2(z); w3(y); w2(y)

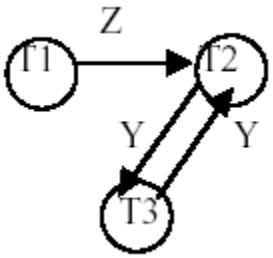
Answer:
Schedule S1: It is a serializable schedule because

- □T1 only reads X (r1(X)), which is not modified either by T2 or T3,
- T3 reads X (r3(X)) before T1 modifies it (w1(X)),
- □T2 reads Y (r2(Y)) and writes it (w2(Y)) only after T3 has written to it (w3(Y))
- Thus, the serializability graph is

Schedule is not a serializable schedule because

- T2 reads Y (r2(Y)), which is then read and modified by T3 (w3(Y))
- T3 reads Y (r3(Y)), which then modified before T2 modifies Y (w2(Y))

In the above order T3 interferes in the execution of T2, which makes the schedule nonserializable.



20.24 - Consider schedules S3, S4, and S5 below. Determine whether each schedule is strict, cascadeless, recoverable, or nonrecoverable. (Determine the strictest recoverability condition that each schedule satisfies.)
S3: r1(x); r2(z); r1(z); r3(x); r3(y); w1(x); c1; w3(y); c3; r2(y); w2(z); w2(y);c2
S4: r1(x); r2(z); r1(z); r3(x); r3(y); w1(x); w3(y); r2(y); w2(z); w2(y); c1; c2; c3;
S5: r1(x); r2(z); r3(x); r1(z); r2(y); r3(y); w1(x); w2(z); w3(y); w2(y); c3; c2;

Answer:
Strict schedule: A schedule is strict if it satisfies the following conditions:
1. Tj reads a data item X ***after*** Ti has written to X and Ti is terminated (aborted or committed)
2. Tj writes a data item X ***after*** Ti has written to X and Ti is terminated (aborted or committed)
Schedule S3 is not strict because T3 reads X (r3(X)) ***before*** T1 has written to X (w1(X)) but T3 commits ***after*** T1. In a strict schedule T3 must read X ***after*** C1.
Schedule S4 is not strict because T3 reads X (r3(X)) ***before*** T1 has written to X (w1(X)) but T3 commits ***after*** T1. In a strict schedule T3 must read X ***after*** C1.
Schedule S5 is not strict because T3 reads X (r3(X)) ***before*** T1 has written to X (w1(X)) but T3 commits ***after*** T1. In a strict schedule T3 must read X ***after*** C1.
Cascadeless schedule: A schedule is cascadeless if the following condition is satisfied:
Tj reads X only ***after*** Ti has written to X and terminated (aborted or committed).
Schedule S3 is ***not cascadeless*** because T3 reads X (r3(X)) before T1 commits.
Schedule S4 is ***not cascadeless*** because T3 reads X (r3(X)) before T1 commits.
Schedule S5 is ***not cascadeless*** because T3 reads X (r3(X)) ***before*** T1 commits or T2 reads
Y (r2(Y)) ***before*** T3 commits.

NOTE: According to the definition of cascadeless schedules S3, S4, and S4 are not cascadeless. However, T3 is not affected if T1 is rolled back in any of the schedules, that is, T3 does not have to roll back if T1 is rolled back. The problem occurs because these schedules are not serializable.

Recoverable schedule: A schedule is recoverable if the following condition is satisfied:
□□Tj commits after Ti if Tj has read any data item written by Ti.
NOTE: Ci > Cj means Ci happens ***before*** Cj. Ai denotes abort Ti. To test if a schedule is recoverable one has to include abort operations. Thus in testing the recoverability abort operations will have to used in place of commit one at a time. Also the strictest condition is where a transaction neither reads nor writes to a data item, which was written to by a transaction that has not committed yet.

□□If A1>C3>C2, then S3 is ***recoverable*** because rolling back of T1 does not affect T2 and T3. If C1>A3>C2. S3 is ***not recoverable*** because T2 read the value of Y (r2(Y)) ***after*** T3 wrote X (w3(Y)) and T2 committed but T3 rolled back. Thus, T2 used non- existent value of Y. If C1>C3>A3, then S3 is ***recoverable*** because roll back of T2 does not affect T1 and T3. Strictest condition of S3 is C3>C2.
□□If A1>C2>C3, then S4 is ***recoverable*** because roll back of T1 does not affect T2 and T3. If C1>A2>C3, then S4 is ***recoverable*** because the roll back of T2 will restore the value of Y that was read and written to by T3 (w3(Y)). It will not affect T1. If C1>C2>A3, then S4 is ***not recoverable*** because T3 will restore the value of Y which was not read by T2. Strictest condition of S4 is C3>C2, but it is not satisfied by S4.
□□If A1>C3>C2, then S5 is ***recoverable*** because neither T2 nor T3 writes to X, which is written by T1. If C1>A3>C2, then S5 is ***not recoverable*** because T3 will restore the value of Y, which was not read by T2. Thus, T2 committed with a non-existent value of Y. If C1>C3>A2, then S5 is ***recoverable*** because it will restore the value of Y to the value, which was read by T3. Thus, T3 committed with the right value of Y. Strictest condition of S3 is C3>C2, but it is not satisfied by S5.

CHAPTER 21: CONCURRENCY CONTROL TECHNIQUES

Answers to Selected Exercises

21.20 - Prove that the basic two-phase locking protocol guarantees conflict serializability of schedules. (Hint: Show that, if a serializability graph for a schedule has a cycle, then at least one of the transactions participating in the schedule does not obey the two-phase locking protocol.)

Answer:
(This proof is by contradiction, and assumes binary locks for simplicity. A similar proof can be made for shared/exclusive locks.)
Suppose we have n transactions T1, T2, ..., Tn such that they all obey the basic two-phase locking rule (i.e. no transaction has an unlock operation followed by a lock operation). Suppose that a non-(conflict)-serializable schedule S for T1, T2, ..., Tn does occur; then, according to Section 17.5.2, the precedence (serialization) graph for S must have a cycle. Hence, there must be some sequence within the schedule of the form:
S: ...; [o1(X); ...; o2(X);] ...; [o2(Y); ...; o3(Y);] ... ; [on(Z); ...; o1(Z);]...
where each pair of operations between square brackets [o,o] are conflicting (either [w, w], or [w, r], or [r,w]) in order to create an arc in the serialization graph. This implies that in transaction T1, a sequence of the following form occurs:
T1: ...; o1(X); ... ; o1(Z); ...
Furthermore, T1 has to unlock item X (so T2 can lock it before applying o2(X) to follow the rules of locking) and has to lock item Z (before applying o1(Z), but this must occur after Tn has unlocked it). Hence, a sequence in T1 of the following form occurs:
T1: ...; o1(X); ...; unlock(X); ... ; lock(Z); ...; o1(Z); ...

This implies that T1 does not obey the two-phase locking protocol (since lock(Z) follows unlock(X)), contradicting our assumption that all transactions in S follow the two-phase locking protocol.

21.21 - Modify the data structures for multiple-mode locks and the algorithms for read_lock(X), write_lock(X), and unlock(X) so that upgrading and downgrading of locks are possible. (Hint: The lock needs to keep track of the transaction id(s) that hold the lock, if any.)

Answer:
We assume that a List of transaction ids that have read-locked an item is maintained, as well as the (single) transaction id that has write-locked an item. Only read_lock and write_lock are shown below.

```
read_lock (X, Tn):
B: if lock (X) = "unlocked"
then begin lock (X) <- "read_locked, List(Tn)";
no_of_reads (X) <- 1
end
else if lock(X) = "read_locked, List"
then begin
(* add Tn to the list of transactions that have read_lock on X *)
lock (X) <- "read_locked, Append(List,Tn)";
no_of_reads (X) <- no_of_reads (X) + 1
end
else if lock (X) = "write_locked, Tn"
(* downgrade the lock if write_lock on X is held by Tn itself *)
then begin lock (X) <- "read_locked, List(Tn)";
no_of_reads (X) <- 1
end
else begin
wait (until lock (X) = "unlocked" and
the lock manager wakes up the transaction);
goto B;
end;
write_lock (X,Tn);
B: if lock (X) = "unlocked"
then lock (X) <- "write_locked, Tn"
else
if ( (lock (X) = "read_locked, List") and (no_of_reads (X) = 1)
and (transaction in List = Tn) )
(* upgrade the lock if read_lock on X is held only by Tn itself *)
then lock (X) = "write_locked, Tn"
else begin
wait (until ( [ lock (X) = "unlocked" ] or
[ (lock (X) = "read_locked, List") and (no_of_reads (X) = 1)
and (transaction in List = Tn) ] ) and
the lock manager wakes up the transaction);
goto B;
end;
```

21.21 - Prove that strict two-phase locking guarantees strict schedules.

Answer:
Since no other transaction can read or write an item written by a transaction T until T has committed, the condition for a strict schedule is satisfied.

21.23 – No solution provided.

21.24 - Prove that cautious waiting avoids deadlock.

Answer:
In cautious waiting, a transaction Ti can wait on a transaction Tj (and hence Ti becomes blocked) only if Tj is not blocked at that time, say time b(Ti), when Ti waits. Later, at some time b(Tj) > b(Ti), Tj can be blocked and wait on another transaction Tk only if Tk is not blocked at that time. However, Tj cannot be blocked by waiting on an already blocked transaction since this is not allowed by the protocol. Hence, the wait-for graph among the blocked transactions in this system will follow the blocking times and will never have a cycle, and so deadlock cannot occur.

21.25 - Apply the timestamp ordering algorithm to the schedules of Figure 21.8 (b) and (c), and determine whether the algorithm will allow the execution of the schedules.

Answer:
Let us assume a clock with linear time points 0, 1, 2, 3, ..., and that the original read and write timestamps of all items are 0 (without loss of generality).
read_TS(X) = read_TS(Y) = read_TS(Z) = 0
write_TS(X) = write_TS(Y) = write_TS(Z) = 0
Let us call the schedules in Figure 17.8(b) Schedule E or SE, and that in Figure 17.8(c) Schedule F or SF. The two schedules can be written as follows in shorthand notation:
SE:
r2(Z); r2(Y); w2(Y); r3(Y); r3(Z); r1(X); w1(X); w3(Y); w3(Z); r2(X); r1(Y); w1(Y); w2(X);
1 2 3 4 5 6 7 8 9 10 11 12 13
SF:
r3(Y); r3(Z); r1(X); w1(X); w3(Y); w3(Z); r2(Z); r1(Y); w1(Y); r2(Y); w2(Y); r2(X); w2(X);
1 2 3 4 5 6 7 8 9 10 11 12 13
Assume that each operation takes one time unit, so that the numbers under the operations indicate the time when each operation occurred. Also assume that each transaction timestamp corresponds to the time of its first operations in each schedule, so the transaction timestamps are as follows (Note: These values do not change during the schedule, since they are assigned as unique identifiers to the transactions):
Schedule E Schedule F
TS(T1) = 6 TS(T1) = 3
TS(T2) = 1 TS(T2) = 7

TS(T3) = 4 TS(T3) = 1

(a) Applying timestamp ordering to Schedule E:
Initial values (new values are shown after each operation):
read_TS(X)=0,read_TS(Y)=0,read_TS(Z)=0,write_TS(X)=0,write_TS(Y)=0,write_TS(Z)=0
TS(T1)=6, TS(T2)=1, TS(T3)=4 (These do not change)
T2: read_item(Z)
TS(T2) > write_TS(Z)
Execute read_item(Z)
Set read_TS(Z) <- max(read_TS(Z),TS(T2)) = 1
read_TS(X)=0,read_TS(Y)=0,read_TS(Z)=1,write_TS(X)=0,write_TS(Y)=0,write_TS(Z)=0
T2: read_item(Y)
TS(T2) > write_TS(Y)
Execute read_item(Y)
Set read_TS(Y) <- max(read_TS(Y),TS(T2)) = 1
read_TS(X)=0,read_TS(Y)=1,read_TS(Z)=1,write_TS(X)=0,write_TS(Y)=0,write_TS(Z)=0
T2: write_item(Y)
TS(T2) = read_TS(Y) and TS(T2) > write_TS(Y)
Execute write_item(Y)
write_TS(Y) <- max(write_TS(Y),TS(T2)) = 1
read_TS(X)=0,read_TS(Y)=1,read_TS(Z)=1,write_TS(X)=0,write_TS(Y)=1,write_TS(Z)=0
T3: read_item(Y)
TS(T3) > write_TS(Y)
Execute read_item(Y)
read_TS(Y) <- max(read_TS(Y),TS(T3)) = 4
read_TS(X)=0,read_TS(Y)=4,read_TS(Z)=1,write_TS(X)=0,write_TS(Y)=1,write_TS(Z)=0
T3: read_item(Z)
TS(T3) > write_TS(Z)
Execute read_item(Z)
read_TS(Z) <- max(read_TS(Z),TS(T3)) = 4
read_TS(X)=0,read_TS(Y)=4,read_TS(Z)=1,write_TS(X)=0,write_TS(Y)=1,write_TS(Z)=0
T1: read_item(X)
TS(T1) > write_TS(X)
Execute read_item(X)
read_TS(X) <- max(read_TS(X),TS(T1)) = 6
read_TS(X)=6,read_TS(Y)=4,read_TS(Z)=1,write_TS(X)=0,write_TS(Y)=1,write_TS(Z)=0
T1: write_item(X)
TS(T1) = read_TS(X) and TS(T1) > write_TS(X)
Execute write_item(X)
write_TS(X) <- max(write_TS(X),TS(T1)) = 6
read_TS(X)=6,read_TS(Y)=4,read_TS(Z)=1,write_TS(X)=6,write_TS(Y)=1,write_TS(Z)=0
T3: write_item(Y)
TS(T3) = read_TS(Y) and TS(T3) > write_TS(Y)
Execute write_item(Y)
write_TS(Y) <- max(write_TS(Y),TS(T3)) = 4
read_TS(X)=6,read_TS(Y)=4,read_TS(Z)=1,write_TS(X)=6,write_TS(Y)=4,write_TS(Z)=0
T3: write_item(Z)
TS(T3) > read_TS(Z) and TS(T3) > write_TS(Z)
Execute write_item(Z)
write_TS(Z) <- max(write_TS(Z),TS(T3)) = 4
read_TS(X)=6,read_TS(Y)=4,read_TS(Z)=1,write_TS(X)=6,write_TS(Y)=4,write_TS(Z)=4
T2: read_item(X)
TS(T2) < write_TS(X)

Abort and Rollback Y2, Reject read_item(X)

Result: Since T3 had read the value of Y that was written by T2, T3 should also be aborted and rolled by the recovery technique (because of cascading rollback); hence, all effects of T2 and T3 would also be erased and only T1 would finish execution.

(b) Applying timestamp ordering to Schedule F:
Initial values (new values are shown after each operation):
read_TS(X)=0,read_TS(Y)=0,read_TS(Z)=0,write_TS(X)=0,write_TS(Y)=0,write_TS(Z)=0
TS(T1)=3, TS(T2)=7, TS(T3)=1 (These do not change)
T3: read_item(Y)
TS(T3) > write_TS(Y)
Execute read_item(Y)
Set read_TS(Y) <- max(read_TS(Y),TS(T3)) = 1
read_TS(X)=0,read_TS(Y)=1,read_TS(Z)=0,write_TS(X)=0,write_TS(Y)=0,write_TS(Z)=0
T3: read_item(Z)
TS(T3) > write_TS(Z)
Execute read_item(Z)
Set read_TS(Z) <- max(read_TS(Z),TS(T3)) = 1
read_TS(X)=0,read_TS(Y)=1,read_TS(Z)=1,write_TS(X)=0,write_TS(Y)=0,write_TS(Z)=0
T1: read_item(X)
TS(T1) > write_TS(X)
Execute read_item(X)
read_TS(X) <- max(read_TS(X),TS(T1)) = 3
read_TS(X)=3,read_TS(Y)=1,read_TS(Z)=1,write_TS(X)=0,write_TS(Y)=0,write_TS(Z)=0
T1: write_item(X)
TS(T1) = read_TS(X) and TS(T1) > write_TS(X)
Execute write_item(X)
write_TS(X) <- max(write_TS(X),TS(T1)) = 3
read_TS(X)=3,read_TS(Y)=1,read_TS(Z)=1,write_TS(X)=3,write_TS(Y)=0,write_TS(Z)=0
T3: write_item(Y)
TS(T3) = read_TS(Y) and TS(T3) > write_TS(Y)
Execute write_item(Y)
write_TS(Y) <- max(write_TS(Y),TS(T3)) = 1
read_TS(X)=3,read_TS(Y)=1,read_TS(Z)=1,write_TS(X)=3,write_TS(Y)=1,write_TS(Z)=0
T3: write_item(Z)
TS(T3) = read_TS(Z) and TS(T3) > write_TS(Z)
Execute write_item(Z)
write_TS(Z) <- max(write_TS(Z),TS(T3)) = 1
read_TS(X)=3,read_TS(Y)=1,read_TS(Z)=1,write_TS(X)=3,write_TS(Y)=1,write_TS(Z)=1
T2: read_item(Z)
TS(T2) > write_TS(Z)
Execute read_item(Z)
Set read_TS(Z) <- max(read_TS(Z),TS(T2)) = 7
read_TS(X)=3,read_TS(Y)=1,read_TS(Z)=7,write_TS(X)=3,write_TS(Y)=1,write_TS(Z)=1
T1: read_item(Y)
TS(T1) > write_TS(Y)
Execute read_item(Y)
Set read_TS(Y) <- max(read_TS(Y),TS(T1)) = 3
read_TS(X)=3,read_TS(Y)=3,read_TS(Z)=7,write_TS(X)=3,write_TS(Y)=1,write_TS(Z)=1
T1: write_item(Y)
TS(T1) = read_TS(Y) and TS(T1) > write_TS(Y)
Execute write_item(Y)

write_TS(Y) <- max(read_TS(Y),TS(T1)) = 3
read_TS(X)=3,read_TS(Y)=3,read_TS(Z)=7,write_TS(X)=3,write_TS(Y)=3,write_TS(Z)=1
T2: read_item(Y)
TS(T2) > write_TS(Y)
Execute read_item(Y)
Set read_TS(Y) <- max(read_TS(Y),TS(T2)) = 7
read_TS(X)=3,read_TS(Y)=7,read_TS(Z)=7,write_TS(X)=3,write_TS(Y)=3,write_TS(Z)=1
T2: write_item(Y)
TS(T2) = read_TS(Y) and TS(T2) > write_TS(Y)
Execute write_item(Y)
write_TS(Y) <- max(write_TS(Y),TS(T2)) = 7
read_TS(X)=3,read_TS(Y)=7,read_TS(Z)=7,write_TS(X)=3,write_TS(Y)=7,write_TS(Z)=1
T2: read_item(X)
TS(T2) > write_TS(X)
Execute read_item(X)
Set read_TS(X) <- max(read_TS(X),TS(T2)) = 7
read_TS(X)=7,read_TS(Y)=7,read_TS(Z)=7,write_TS(X)=3,write_TS(Y)=3,write_TS(Z)=1
T2: write_item(X)
TS(T2) = read_TS(X) and TS(T2) > write_TS(X)
Execute write_item(X)
write_TS(X) <- max(write_TS(X),TS(T2)) = 7
read_TS(X)=7,read_TS(Y)=7,read_TS(Z)=7,write_TS(X)=7,write_TS(Y)=7,write_TS(Z)=1

Result: Schedule F executes successfully.

21.26 - Repeat Exercise 21.25, but use the multiversion timestamp ordering method.

Answer:
Let us assume the same timestamp values as in the solution for Exercise 18.21 above. To refer to versions, we use X, Y, Z to reference the original version (value) of each item, and then use indexes (1, 2, ...) to refer to newly written version (for example, X1, X2, ...).

(a) Applying multiversion timestamp ordering to Schedule E:
Initial values (new values are shown after each operation):
read_TS(X)=0,read_TS(Y)=0,read_TS(Z)=0,write_TS(X)=0,write_TS(Y)=0,write_TS(Z)=0
TS(T1)=6, TS(T2)=1, TS(T3)=4 (These do not change)
T2: read_item(Z)
Execute read_item(Z)
Set read_TS(Z) <- max(read_TS(Z),TS(T2)) = 1
read_TS(X)=0,read_TS(Y)=0,read_TS(Z)=1,write_TS(X)=0,write_TS(Y)=0,write_TS(Z)=0
T2: read_item(Y)
Execute read_item(Y)
Set read_TS(Y) <- max(read_TS(Y),TS(T2)) = 1
read_TS(X)=0,read_TS(Y)=1,read_TS(Z)=1,write_TS(X)=0,write_TS(Y)=0,write_TS(Z)=0
T2: write_item(Y)
TS(T2) = read_TS(Y)
Execute write_item(Y) (by creating a new version Y1 of Y)
write_TS(Y1) <- TS(T2) = 1,
read_TS(Y1) <- TS(T2) = 1
read_TS(X)=0,read_TS(Y)=1,read_TS(Y1)=1,read_TS(Z)=1,
write_TS(X)=0,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Z)=0

T3: read_item(Y)
Execute read_item(Y) by reading the value of version Y1
read_TS(Y1) <- max(read_TS(Y1),TS(T3)) = 4
read_TS(X)=0,read_TS(Y)=1,read_TS(Y1)=4,read_TS(Z)=1,
write_TS(X)=0,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Z)=0
T3: read_item(Z)
Execute read_item(Z)
read_TS(Z) <- max(read_TS(Z),TS(T3)) = 4
read_TS(X)=0,read_TS(Y)=1,read_TS(Y1)=4,read_TS(Z)=4,
write_TS(X)=0,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Z)=0
T1: read_item(X)
Execute read_item(X)
read_TS(X) <- max(read_TS(X),TS(T1)) = 6
read_TS(X)=6,read_TS(Y)=1,read_TS(Y1)=4,read_TS(Z)=4,
write_TS(X)=0,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Z)=0
T1: write_item(X)
Execute write_item(X) (by creating a new version X1 of X)
write_TS(X) <- TS(T1) = 6,
read_TS(X) <- TS(T1) = 6
read_TS(X)=6,read_TS(X1)=6,read_TS(Y)=1,read_TS(Y1)=4,read_TS(Z)=4,
write_TS(X)=0,write_TS(X1)=6,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Z)=0
T3: write_item(Y)
Execute write_item(Y) (by creating a new version Y2 of Y)
write_TS(Y2) <- TS(T3) = 4,
read_TS(Y2) <- TS(T3) = 4
read_TS(X)=6,read_TS(X1)=6,read_TS(Y)=1,read_TS(Y1)=4,read_TS(Y2)=4,
read_TS(Z)=4,
write_TS(X)=0,write_TS(X1)=6,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Y2)=4,
write_TS(Z)=0
T3: write_item(Z)
Execute write_item(Z) (by creating a new version Z1 of Z)
write_TS(Z1) <- TS(T3) = 4,
read_TS(Z1) <- TS(T3) = 4
read_TS(X)=6,read_TS(X1)=6,read_TS(Y)=1,read_TS(Y1)=4,read_TS(Y2)=4,
read_TS(Z)=4,read_TS(Z1)=4,
write_TS(X)=0,write_TS(X1)=6,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Y2)=4,
write_TS(Z)=0,write_TS(Z1)=4
T2: read_item(X)
Execute read_item(X) by reading the value of the initial version X
read_TS(X) <- max(read_TS(X),TS(T3)) = 6
read_TS(X)=6,read_TS(X1)=6,read_TS(Y)=1,read_TS(Y1)=4,read_TS(Y2)=4,
read_TS(Z)=4,read_TS(Z1)=4,
write_TS(X)=0,write_TS(X1)=6,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Y2)=4,
write_TS(Z)=0,write_TS(Z1)=4
T1: read_item(Y)
Execute read_item(Y) by reading the value of version Y2
read_TS(Y2) <- max(read_TS(Y2),TS(T3)) = 6
read_TS(X)=6,read_TS(X1)=6,read_TS(Y)=1,read_TS(Y1)=4,read_TS(Y2)=6,
read_TS(Z)=4,read_TS(Z1)=4,
write_TS(X)=0,write_TS(X1)=6,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Y2)=4,
write_TS(Z)=0,write_TS(Z1)=4
T1: write_item(Y)
Execute write_item(Y) (by creating a new version Y3 of Y)

write_TS(Y3) <- TS(T3) = 4,
read_TS(Y2) <- TS(T3) = 4
read_TS(X)=6,read_TS(X1)=6,read_TS(Y)=1,read_TS(Y1)=4,read_TS(Y2)=6,
read_TS(Y3)=6,read_TS(Z)=4,read_TS(Z1)=4,
write_TS(X)=0,write_TS(X1)=6,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Y2)=4,
write_TS(Y2)=6,write_TS(Z)=0,write_TS(Z1)=4
T2: write_item(X)
Abort and Rollback T2 since read_TS(X) >TS(T2)

Result: Since T3 had read the value of Y that was written by T2, T3 should also be aborted and rolled by the recovery technique (because of cascading rollback); hence, all effects of T2 and T3 would also be erased and only T1 would finish execution.

(b) Applying timestamp ordering to Schedule F:
Initial values (new values are shown after each operation):
read_TS(X)=0,read_TS(Y)=0,read_TS(Z)=0,write_TS(X)=0,write_TS(Y)=0,write_TS(Z)=0
TS(T1)=3, TS(T2)=7, TS(T3)=1 (These do not change)
T3: read_item(Y)
Execute read_item(Y)
Set read_TS(Y) <- max(read_TS(Y),TS(T3)) = 1
read_TS(X)=0,read_TS(Y)=1,read_TS(Z)=0,write_TS(X)=0,write_TS(Y)=0,write_TS(Z)=0
T3: read_item(Z)
Execute read_item(Z)
Set read_TS(Z) <- max(read_TS(Z),TS(T3)) = 1
read_TS(X)=0,read_TS(Y)=1,read_TS(Z)=1,write_TS(X)=0,write_TS(Y)=0,write_TS(Z)=0
T1: read_item(X)
Execute read_item(X)
read_TS(X) <- max(read_TS(X),TS(T1)) = 3
read_TS(X)=3,read_TS(Y)=1,read_TS(Z)=1,write_TS(X)=0,write_TS(Y)=0,write_TS(Z)=0
T1: write_item(X)
Execute write_item(X) by creating a new version X1 of X
write_TS(X1) <- TS(T1) = 3, read_TS(X1) <- TS(T1) = 3
read_TS(X)=3,read_TS(X1)=3,read_TS(Y)=1,read_TS(Z)=1,
write_TS(X)=0,write_TS(X1)=3,write_TS(Y)=0,write_TS(Z)=0
T3: write_item(Y)
Execute write_item(Y) by creating a new version Y1 of Y
write_TS(Y1) <- TS(T3) = 1, read_TS(Y1) <- TS(T3) = 1
read_TS(X)=3,read_TS(X1)=3,read_TS(Y)=1,read_TS(Y1)=1,read_TS(Z)=1,
write_TS(X)=0,write_TS(X1)=3,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Z)=0
T3: write_item(Z)
Execute write_item(Z) by creating a new version Z1 of Z
write_TS(Z1) <- TS(T3) = 1, read_TS(Z1) <- TS(T3) = 1
read_TS(X)=3,read_TS(X1)=3,read_TS(Y)=1,read_TS(Y1)=1,read_TS(Z)=1,
read_TS(Z1)=1,
write_TS(X)=0,write_TS(X1)=3,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Z)=0,
write_TS(Z1)=1
T2: read_item(Z)
Execute read_item(Z) by reading the value of version Z1
Set read_TS(Z1) <- max(read_TS(Z1),TS(T2)) = 7
read_TS(X)=3,read_TS(X1)=3,read_TS(Y)=1,read_TS(Y1)=1,read_TS(Z)=1,
read_TS(Z1)=7,
write_TS(X)=0,write_TS(X1)=3,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Z)=0,
write_TS(Z1)=1

T1: read_item(Y)
Execute read_item(Y) by reading the value of version Y1
Set read_TS(Y1) <- max(read_TS(Y1),TS(T1)) = 3
read_TS(X)=3,read_TS(X1)=3,read_TS(Y)=3,read_TS(Y1)=1,read_TS(Z)=1,
read_TS(Z1)=7,
write_TS(X)=0,write_TS(X1)=3,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Z)=0,
write_TS(Z1)=1
T1: write_item(Y)
Execute write_item(Y) by creating a new version Y2 of Y
write_TS(Y2) <- TS(T1) = 3, read_TS(Y2) <- TS(T1) = 3
read_TS(X)=3,read_TS(X1)=3,read_TS(Y)=3,read_TS(Y1)=1,read_TS(Y2)=3,
read_TS(Z)=1,read_TS(Z1)=7,
write_TS(X)=0,write_TS(X1)=3,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Y2)=3,
write_TS(Z)=0,write_TS(Z1)=1
T2: read_item(Y)
Execute read_item(Y) by reading the value of version Y2
Set read_TS(Y2) <- max(read_TS(Y2),TS(T2)) = 7
read_TS(X)=3,read_TS(X1)=3,read_TS(Y)=3,read_TS(Y1)=1,read_TS(Y2)=7,
read_TS(Z)=1,read_TS(Z1)=7,
write_TS(X)=0,write_TS(X1)=3,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Y2)=3,
write_TS(Z)=0,write_TS(Z1)=1
T2: write_item(Y)
Execute write_item(Y) by creating a new version Y3 of Y
write_TS(Y3) <- TS(T2) = 7, read_TS(Y3) <- TS(T2) = 7
read_TS(X)=3,read_TS(X1)=3,read_TS(Y)=3,read_TS(Y1)=1,read_TS(Y2)=7,
read_TS(Y3)=7,read_TS(Z)=1,read_TS(Z1)=7,
write_TS(X)=0,write_TS(X1)=3,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Y2)=3,
write_TS(Y3)=7,write_TS(Z)=0,write_TS(Z1)=1
T2: read_item(X)
Execute read_item(X) by reading the value of version X1
Set read_TS(X1) <- max(read_TS(X1),TS(T2)) = 7
read_TS(X)=3,read_TS(X1)=7,read_TS(Y)=3,read_TS(Y1)=1,read_TS(Y2)=7,
read_TS(Y3)=7,read_TS(Z)=1,read_TS(Z1)=7,
write_TS(X)=0,write_TS(X1)=3,write_TS(Y)=0,write_TS(Y1)=1,write_TS(Y2)=3,
write_TS(Y3)=7,write_TS(Z)=0,write_TS(Z1)=1
T2: write_item(X)
Execute write_item(X) by creating a new version X2 of X
write_TS(X2) <- TS(T2) = 7, read_TS(X2) <- TS(T2) = 7
read_TS(X)=3,read_TS(X1)=7,read_TS(X2)=7,read_TS(Y)=3,read_TS(Y1)=1,
read_TS(Y2)=7,read_TS(Y3)=7,read_TS(Z)=1,read_TS(Z1)=7,
write_TS(X)=0,write_TS(X1)=3,write_TS(X2)=7,write_TS(Y)=0,write_TS(Y1)=1,
write_TS(Y2)=3,write_TS(Y3)=7,write_TS(Z)=0,write_TS(Z1)=1

Result: Schedule F executes successfully.