**北京邮电大学 2022—2023 学年第一学期**

**《数据库系统原理》**期末考试试题（**A 卷**）

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 考试注意事项 | 一、学生参加考试须带学生证或学院证明，未带者不准进入考场。学生必须按照监考教师指定座位就坐。  二、书本、参考资料、书包等物品一律放到考场指定位置。  三、学生不得另行携带、使用稿纸，要遵守《北京邮电大学考场规则》，有考场违纪或作弊行为者，按相应规定严肃处理。  四、学生必须将答题内容做在试题答卷上，做在试题及草稿纸上一律无效。  五、填空题用英文答，中文答对得一半分。 | | | | | | | | | | |
| 考试  课程 | 数据库系统原理 | | | | 考试时间 | | | 2022 年 12 月 20 日  13:30~15:30 | | | |
| 题号 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 九 |  | 总分 |
| 满分 | 22 | 20 | 15 | 13 | 12 | 18 |  |  |  |  | 100 |
| 得分 |  |  |  |  |  |  |  |  |  |  |  |
| 阅卷  教师 |  |  |  |  |  |  |  |  |  |  |  |

**1.** (22 points) Here is the schema diagram for the **Banking** database(银行数据库). The table *branch* describes the name, the city located and the assets(资产) of the bank’s branches (支行) . The customers of branches are represented in the table *customer*. A customer may have an *account* (存款账户) in a branch. His account is uniquely identified by the attribute *account\_number*, and the attribute *balance* (存款额) records the amount of money in this account; A customer may also have a *loan* (借款账户) in a branch. His loan is solely identified by *loan\_number*, and the amount of money he loans is given by the attribute *amount* (借款额). The relationships between customers and their accounts or loans are modelled as *depositor* or *borrower* respectively.

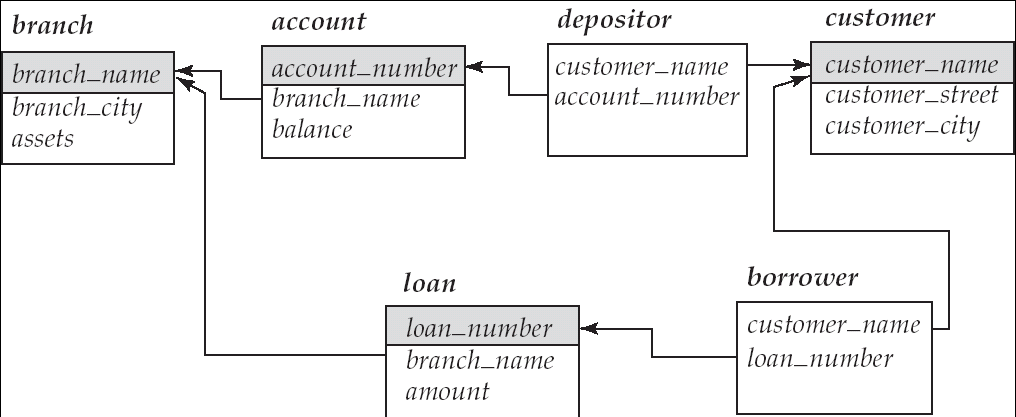


Figure 1 Schema diagram for Banking database

For the following queries, give **relational algebra** expressions for (1), and **SQL statements** for (2)~(5)

1. Find the names of all customers who have an account at the “Haidian” branch, but at any branch have no loan whose amount is greater than 1000. (4 points)
2. Create the table *account*, in which *account\_number* is the primary key, *branch\_name* is a foreign key that defines a referential integrity constraint from *account* to *branch*. It is also required that *branch\_name* is not permitted to be null, and the balance of an account is not below 0. (5 points)

Create table account(

Account\_number int not null,

Branch\_name varchar(50) not null,

Balance int check(balance>=0),

Primary key (account\_number),

Foreign key(branch\_name) references brach(branch\_name)

)

1. Use one or more SQL statements to verify whether or not the functional dependency

*account\_number*→*balance* is satisfied by the table *account*. (4 points)

select account\_number from account a join account b where a.account\_number = b.account\_number and a.balance <> b.balance

1. A customer BUPT paid off all of his loans in Haidian branch, and accordingly, use SQL statements to delete related information in database. (4 points)

DELETE FROM borrower

WHERE customer\_name = 'BUPT'

AND loan\_number IN (

SELECT loan\_number

FROM loan

WHERE branch\_name = 'Haidian'

);

DELETE FROM loan

WHERE branch\_name = 'Haidian'

AND loan\_number IN (

SELECT loan\_number

FROM borrower

WHERE customer\_name = 'BUPT'

);

1. For each branch located in Beijing, calculate the total number of the customers who have accounts in the branch. For the branch that have more than 100 account customers (i.e. depositors), list its name, the total number of the depositors, as well as the sum of the balances, in descending order of balance sum. (5 points)

SELECT b.branch\_name, COUNT(DISTINCT d.customer\_name) AS num\_customers, SUM(a.balance) AS total\_balance

FROM branch b JOIN account a ON b.branch\_name = a.branch\_name

JOIN depositor d ON a.account\_number = d.account\_number

WHERE b.city = 'Beijing'

GROUP BY b.branch\_name

HAVING COUNT(DISTINCT d.customer\_name) > 100

ORDER BY total\_balance DESC;

**2**.(20 points) A cake chain store(蛋糕连锁店) needs to manage its stores, products(cakes), employees and customers, and needs to manage the following information:

1. Stores（店铺）: including Store\_ID， Address （including province, city, street, zip code), Contact telephone number and other information. Store\_ID is unique, and a Store can have multiple contact numbers;
2. Cakes（蛋糕）: including Cakes\_ID , Name, Price, and the Cakes\_ID is unique;
3. Employees(员工): including Employee\_ID, Name, Gender, Age, Telephone Number and Employment Date, and the Employee\_ID is unique;
4. Customers(客户): including the Customer\_ID, Name, Gender, Age, Contact number. The Customer\_ID is unique;
5. Customer types: including the Type ID, Type Description. The Type ID is unique;
6. One store has many employees, and one employee can only work in one store；
7. A customer can only belong to one customer type，and one customer type can contain multiple customers;

(9) Sales information: Record the information about customers' purchase of cakes in a store, including the date and quantity of purchase.

Design a relational database to managing the information mentioned above:

1. Design the E/R diagram. It is required that mapping cardinality of each relationship and participation of each entity to the relationship should be given in the diagram. (10 points)
2. Convert the E-R diagram to the proper relational schemas, and give the primary key of each relation schema by underlines. (10 points)

Store ( store\_ID , address\_province, address\_city, address\_street, address\_zipcode)

Phone (store\_ID , phone\_number)

Cake ( cake\_ID , name, price)

Employee ( employee\_ID , name, gender, age, phone\_number, employment\_date)

Customer ( customer\_ID , name, gender, age)

Customer\_Type ( type\_ID , type\_description)

Belongs\_To ( customer\_ID , type\_ID )

Sale ( sale\_ID , customer\_ID , cake\_ID , date)

Account ( account\_ID , customer\_ID , store\_ID , balance)

Loan ( loan\_ID , customer\_ID , store\_ID , amount)

Contains ( store\_ID , cake\_ID , quantity)

1. (15 points) The functional dependency set F={A→C, B→A, C→DE, D→AC, B→E} holds on the relation schema R = (A, B, C, D, E),
2. Compute (AB)+. (2 points)

Start with (AB)+ = AB

Repeat until no more attributes can be added to (AB)+:

For each functional dependency A -> B in F, if A is a subset of (AB)+, then add B to (AB)+

A -> C is in F, and A is a subset of (AB)+, so add C to (AB)+. Now (AB)+ = ABC

B -> A is in F, but A is already in (AB)+, so no change

C -> DE is in F, and C is a subset of (AB)+, so add DE to (AB)+. Now (AB)+ = ABCDE

D -> AC is in F, but AC are already in (AB)+, so no change

B -> E is in F, and B is a subset of (AB)+, so add E to (AB)+. Now (AB)+ = ABCDEE

No more functional dependencies can be applied, so stop the loop

Return (AB)+ = ABCDE as the final result

1. List all the candidate keys of R. (1 points)

A candidate key of a relation is a minimal set of attributes that can uniquely identify each tuple in the relation. To find all the candidate keys of R, we can use the following steps:

Find a superkey of R, which is a set of attributes that can uniquely identify each tuple in R. One way to find a superkey is to take the union of all the left-hand sides of the functional dependencies in F. In this case, a superkey of R is {A, B, D}.

Check if the superkey is minimal, meaning that no proper subset of it is also a superkey. If it is minimal, then it is a candidate key. If not, then try to remove one attribute at a time and check if the remaining set is still a superkey by computing its closure and seeing if it contains all the attributes of R. If the remaining set is still a superkey, then repeat this step until no more attributes can be removed. If the remaining set is not a superkey, then restore the removed attribute and try another one.

Repeat steps 1 and 2 for all possible combinations of attributes until all candidate keys are found.

For your question, you need to list all the candidate keys of R = (A, B, C, D, E) with respect to F={A→C, B→A, C→DE, D→AC, B→E}.

One possible list of all the candidate keys of R is:

{A, B}

{B, D}

These are the only two minimal sets of attributes that can uniquely identify each tuple in R using F. We can verify this by computing their closures:

(AB)+ = ABCDE

(BD)+ = ABCDE

Both closures contain all the attributes of R, so they are superkeys. Moreover, no proper subset of them is also a superkey. For example:

(A)+ = AC

(B)+ = ABE

(D)+ = DAC

None of these closures contain all the attributes of R, so they are not superkeys. Therefore, {A, B} and {B, D} are minimal superkeys or candidate keys.

1. What is the highest normal form of R, and why? (3 points)

Check if R is in 1NF. 1NF requires that every attribute of R is atomic, meaning that it cannot be further decomposed into smaller parts. In this case, all attributes of R are atomic, so R is in 1NF.

Check if R is in 2NF. 2NF requires that every non-prime attribute of R is fully functionally dependent on every candidate key of R. A non-prime attribute is an attribute that is not part of any candidate key. A candidate key is a minimal set of attributes that can uniquely identify each tuple in R. A full functional dependency means that the dependency cannot be reduced by removing any attribute from the determinant. In this case, the candidate keys of R are {A, B} and {B, D}. The non-prime attributes of R are C and E. C and E are fully functionally dependent on both candidate keys, since A -> C and B -> E are in F and they cannot be reduced. Therefore, R is in 2NF.

Check if R is in 3NF. 3NF requires that every non-prime attribute of R is non-transitively dependent on every candidate key of R. A non-transitive dependency means that the dependency cannot be derived from another dependency using transitivity rule. In this case, C and E are non-transitively dependent on both candidate keys, since there is no other dependency in F that can imply A -> C or B -> E using transitivity rule. Therefore, R is in 3NF.

Check if R is in BCNF. BCNF requires that every determinant of R is a candidate key. A determinant is an attribute or a set of attributes that functionally determines another attribute or a set of attributes. In this case, A and B are determinants of R, but they are not candidate keys by themselves. Therefore, R is not in BCNF.

The highest normal form of R is 3NF, because it satisfies all the conditions for 1NF, 2NF and 3NF, but not for BCNF.

1. Is R1=( A, B, C) and R2=( C, D, E) a lossless-join decomposition of R, and Why? (3 points)

Method 1: Compute the natural join of the decomposed relations and compare it with the original relation. If they are equal, then the decomposition is lossless-join. This method is simple but inefficient, especially for large relations.

Method 2: Use the functional dependencies of R and check if one of the following conditions holds: (R1 ∩ R2) -> R1 or (R1 ∩ R2) -> R2, where R1 and R2 are the decomposed relations and -> denotes functional dependency. This method is based on the fact that if a common attribute of two relations is a superkey of one of them, then their natural join will not produce any spurious tuples. This method is more efficient but requires knowledge of functional dependencies.

For your question, you need to check if R1=( A, B, C) and R2=( C, D, E) is a lossless-join decomposition of R = (A, B, C, D, E) with respect to F={A→C, B→A, C→DE, D→AC, B→E}.

One possible way to check if R1 and R2 is a lossless-join decomposition of R is:

Use Method 2 and check if one of the following conditions holds: (R1 ∩ R2) -> R1 or (R1 ∩ R2) -> R2

The common attribute of R1 and R2 is C. So we need to check if C -> (A, B, C) or C -> (C, D, E) are in F+ (the closure of F).

To compute F+, we can use the following algorithm:

Start with F+ = F

Repeat until no more functional dependencies can be added to F+:

For each functional dependency X -> Y in F+, if Z -> W is in F+ and Y ∩ W = ∅, then add XZ -> YW to F+

For each functional dependency X -> Y in F+, if Z -> W is in F+ and Y ⊆ W, then add XZ -> W to F+

For each functional dependency X -> Y in F+, if Z -> W is in F+ and W ⊆ Y, then add XZ -> Y to F+

Return F+ as the final result

Using this algorithm, we can compute F+ as follows:

Start with F+ = {A→C, B→A, C→DE, D→AC, B→E}

Repeat until no more functional dependencies can be added to F+:

For each functional dependency X -> Y in F+, if Z -> W is in F+ and Y ∩ W = ∅, then add XZ -> YW to F+

A -> C and B -> A are in F+, and C ∩ A = ∅, so add AB -> CA to F+

A -> C and D -> AC are in F+, and C ∩ AC = ∅, so add AD -> CAC to F+

B -> A and C -> DE are in F+, and A ∩ DE = ∅, so add BC -> ADE to F+

B -> A and D -> AC are in F+, and A ∩ AC = ∅, so add BD -> AAC to F+

C -> DE and D -> AC are in F+, and DE ∩ AC = ∅, so add CD -> DEAC to F+

For each functional dependency X -> Y in F+, if Z -> W is in F+ and Y ⊆ W, then add XZ -> W to F+

A -> C and D -> AC are in F+, and C ⊆ AC, so add AD -> AC to F+

B -> A and D -> AC are in F+, and A ⊆ AC, so add BD -> AC to F+

C -> DE and D -> AC are in F+, and DE ⊆ DEAC, so add CD -> DEAC to F+

For each functional dependency X -> Y in F+, if Z -> W is in F+ and W ⊆ Y, then add XZ -> Y to F+

A -> C and D -> AC are in F+, and AC ⊆ CAC, so add AD -> CAC to F+

B -> A and D -> AC are in F+, and AC ⊆ AAC, so add BD -> AAC to F+

C -> DE and D -> AC are in F+, and DEAC ⊆ DEAC, so add CD -> DEAC to F+

No more functional dependencies can be added to F+, so stop the loop

Return F+ = {A→C, B→A, C→DE, D→AC, B→E, AB → CA, AD → CAC, BC → ADE, BD → AAC, CD → DEAC, AD → AC, BD → AC} as the final result

Now we can check if C -> (A, B, C) or C -> (C, D, E) are in F+. We can see that neither of them are in F+, so the decomposition is not lossless-join.

The answer is that R1=( A, B, C) and R2=( C, D, E) is not a lossless-join decomposition of R = (A, B, C, D, E) with respect to F={A→C, B→A, C→DE, D→AC, B→E}, because neither of the conditions (R1 ∩ R2) -> R1 or (R1 ∩ R2) -> R2 holds.

1. Compute the canonical cover Fc. (2 points)

A canonical cover has two important properties:

No functional dependency in the canonical cover has an extraneous attribute. An extraneous attribute is an attribute that can be removed from a functional dependency without changing the closure of F.

No two functional dependencies in the canonical cover have the same left-hand side. That is, there are no two dependencies X -> Y and X -> Z in the canonical cover.

A canonical cover can be computed by using the following algorithm:

Start with Fc = F

Repeat until no more changes can be made to Fc:

Apply the union rule to combine any dependencies in Fc that have the same left-hand side. For example, replace X -> Y and X -> Z with X -> YZ.

Eliminate any extraneous attributes from the left-hand sides of the dependencies in Fc. An attribute A is extraneous in X -> Y if A is a proper subset of X and (X - A)+ contains Y, where (X - A)+ is the closure of (X - A) with respect to Fc. For example, if A -> BC and B -> C are in Fc, then B is extraneous in A -> BC, because (A - B)+ = AC contains C.

Eliminate any extraneous attributes from the right-hand sides of the dependencies in Fc. An attribute A is extraneous in X -> Y if X -> A is in Fc+ (the closure of Fc) and (X -> Y - A) is in Fc+. For example, if A -> BC and A -> B are in Fc, then B is extraneous in A -> BC, because A -> B is in Fc+ and (A -> C) is in Fc+.

Return Fc as the final result.

For your question, you need to compute the canonical cover of F={A→C, B→A, C→DE, D→AC, B→E}.

One possible computation of the canonical cover of F is:

Start with Fc = {A→C, B→A, C→DE, D→AC, B→E}

Repeat until no more changes can be made to Fc:

Apply the union rule to combine any dependencies in Fc that have the same left-hand side. In this case, there are no such dependencies, so no change.

Eliminate any extraneous attributes from the left-hand sides of the dependencies in Fc. In this case, there are no such attributes, so no change.

Eliminate any extraneous attributes from the right-hand sides of the dependencies in Fc. In this case, we can see that C is extraneous in D → AC, because D → C is in Fc+ (since D → AC and C → DE imply D → C) and (D → A) is in Fc+. Similarly, E is extraneous in C → DE, because C → E is in Fc+ (since C → DE implies C → E) and (C → D) is in Fc+. Therefore, we can remove C from D → AC and E from C → DE. Now Fc = {A→C, B→A, C→D, D→A, B→E}.

No more changes can be made to Fc, so stop the loop.

Return Fc = {A→C, B→A, C→D, D→A, B→E} as the final result.

The answer is that the canonical cover of F={A→C, B→A, C→DE, D→AC, B→E} is {A→C, B→A, C→D, D→A, B→E}.

1. Give a lossless-join and dependency-preserving decomposition of R into 3NF. (4points)

A decomposition of R into 3NF can be obtained by using the following algorithm:

Compute the canonical cover Fc of F

For each functional dependency X -> Y in Fc, create a relation schema R(X, Y) with X -> Y as its only functional dependency

Eliminate any redundant relation schemas that are subsets of another relation schema

If none of the relation schemas contains a candidate key of R, add another relation schema that contains a candidate key of R

For your question, you need to decompose R = (A, B, C, D, E) into 3NF with respect to F={A→C, B→A, C→DE, D→AC, B→E}.

One possible decomposition of R into 3NF is:

Compute the canonical cover Fc of F. In the previous question, we have computed Fc as {A→C, B→A, C→D, D→A, B→E}.

For each functional dependency X -> Y in Fc, create a relation schema R(X, Y) with X -> Y as its only functional dependency. We get:

R1(A, C) with A -> C

R2(B, A) with B -> A

R3(C, D) with C -> D

R4(D, A) with D -> A

R5(B, E) with B -> E

Eliminate any redundant relation schemas that are subsets of another relation schema. In this case, there are no such schemas, so no change.

If none of the relation schemas contains a candidate key of R, add another relation schema that contains a candidate key of R. In this case, we know that {A}, {B}, and {C} are candidate keys of R. We can see that none of the relation schemas contains any of these keys. Therefore, we need to add another relation schema that contains one of these keys. We can choose any one of them arbitrarily. For example, we can add:

R6(A) with no functional dependencies

The final decomposition of R into 3NF is {R1(A, C), R2(B, A), R3(C, D), R4(D, A), R5(B, E), R6(A)}.

1. (13 points) Considering the **Banking** database in Figure 1, answer the following questions.
2. The attribute *account\_number* in the table *account* is defined as the primary key and a clustering index is created on it. Some tuples/records are then inserted into the table *account*, the records in data file storing *account* are organized as a heap file, sequential file, or multitable clustering file, and why? (3 points)

A heap file organization is a method of storing records in no particular order, wherever there is free space in the file. A heap file organization is simple and flexible, but it does not support efficient searching or sorting of records based on any attribute value.

A sequential file organization is a method of storing records in sorted order based on a search key attribute. A sequential file organization supports efficient range queries and sequential access of records based on the search key, but it requires extra space for overflow records and extra time for insertion and deletion of records.

A hashing file organization is a method of storing records in buckets based on a hash function applied to a hash key attribute. A hashing file organization supports efficient equality queries and random access of records based on the hash key, but it does not support range queries or sorting of records based on any attribute value.

A multitable clustering file organization is a method of storing records of several different relations in the same file based on a common attribute value. A multitable clustering file organization supports efficient join queries and sequential access of related records, but it requires extra space for duplicate attribute values and extra time for updating multiple relations.

The file organization of the table account is a sequential file organization based on account\_number as the search key. This is because a clustering index is a type of index that stores records in sorted order based on the indexed attribute. Since account\_number is defined as the primary key, it can be used as a search key to uniquely identify each record in the table account. Therefore, by creating a clustering index on account\_number, the records in the table account are organized as a sequential file based on account\_number.

1. For the table *account*, after *account\_number* has been defined as the primary key, another index BNIdx is defined on the attribute *branch\_name* and organized as a B+-tree. Then, the index BNIdx is a primary/clustering index or secondary/non-clustering index, and why? (3 points)

A primary index is an index on a set of attributes that can uniquely identify each record in a file. A primary index is usually created on the primary key of a relation. A primary index supports efficient equality and range queries on the indexed attributes, but it requires extra space and time for maintaining the index structure.

A secondary index is an index on a set of attributes that are not sufficient to uniquely identify each record in a file. A secondary index is usually created on some non-key attributes of a relation. A secondary index supports efficient equality and range queries on the indexed attributes, but it requires extra space and time for maintaining the index structure and accessing the records through the index.

A clustering index is an index that determines the physical order of records in a file. A clustering index is usually created on a set of attributes that have high correlation or frequency of occurrence in queries. A clustering index supports efficient sequential access and join queries on the indexed attributes, but it requires extra time for reorganizing the records when inserting or deleting records.

A non-clustering index is an index that does not affect the physical order of records in a file. A non-clustering index is usually created on a set of attributes that have low correlation or frequency of occurrence in queries. A non-clustering index supports efficient random access queries on the indexed attributes, but it requires extra space for storing pointers to the records.

The index BNIdx is a secondary/non-clustering index. This is because branch\_name is not a primary key attribute of account, so BNIdx cannot be a primary index. Moreover, branch\_name does not determine the physical order of records in account, since account\_number does as a clustering index, so BNIdx cannot be a clustering index either. Therefore, BNIdx is a secondary/non-clustering index.

1. For the following SQL query, in addition to the existing primary indices on the primary keys of the tables, on which attributes the indices can be further defined to speed up the query? (3 points)

select *branch\_city*, sum(*amount*)

from *branch* inner join *loan* on *branch\_name*

where *assets*>1000 group by *branch\_city*

The attributes that can be further indexed to speed up the query are assets and branch\_name. This is because:

An index on assets can help to filter out the records that do not satisfy the where clause condition (assets>1000) without scanning the whole branch table. This can reduce the number of records that need to be joined with the loan table.

An index on branch\_name can help to perform the join operation between branch and loan tables more efficiently by using a nested loop join or a merge join algorithm. This can reduce the number of data pages that need to be accessed or sorted during the join operation.

1. Give a SQL statement to define a composite index on combined search key (*branch\_name*, *amount*) on the table *loan*.

Can this index be efficiently used for the following query, and why? (4 points) select \*

from *loan*

where *amount*>100

However, this index may not be efficiently used for the following query:

select \* from loan where amount>100

This is because:

The query does not use branch\_name as a filter or join condition, so the index cannot help to reduce the number of records that need to be examined based on branch\_name.

The query uses amount as a range condition (>100), so the index cannot help to locate the exact records that satisfy the condition based on amount. The index can only help to find the first record that has amount>100, but then it still has to scan all the subsequent records until it reaches the end of the table or finds a record that has branch\_name different from the first record.

The query selects all columns (\*), so the index cannot help to avoid accessing the data pages of the table loan. The index can only help to access the data pages more efficiently by using a bookmark lookup or a RID lookup algorithm.

Therefore, this index may not improve the query performance significantly and may incur extra overhead for maintaining and accessing the index structure.

1. (12 points) Consider the **Banking** database given in Figure 1.
2. Give a SQL statement to find the customer meeting the following requirements, and list

his name: (i) the customer lives in Beijing; and (ii) he has an account that is with the balance more than 100 and belongs to a branch located in Tianjing. (3 points)

SELECT customer\_name FROM customer INNER JOIN account ON customer.customer\_id = account.customer\_id INNER JOIN branch ON account.branch\_name = branch.branch\_name WHERE customer.city = ‘Beijing’ AND account.balance > 100 AND branch.city = ‘Tianjing’;

1. For the SQL statement in (1), use heuristic optimization scheme to give an optimized query tree. (9 points)

Break up the WHERE clause into three SELECT operations: σcustomer.city=‘Beijing’, σaccount.balance>100, and σbranch.city=‘Tianjing’

Move each SELECT operation as far down the query tree as possible: σcustomer.city=‘Beijing’ is moved to customer, σaccount.balance>100 is moved to account, and σbranch.city=‘Tianjing’ is moved to branch

Rearrange the leaf nodes of the tree: customer and account are swapped to match their join condition on customer\_id

Combine each CARTESIAN PRODUCT operation with a subsequent SELECT operation into a JOIN operation: × and σcustomer.customer\_id=account.customer\_id are combined into ⋈customer.customer\_id=account.customer\_id, and × and σaccount.branch\_name=branch.branch\_name are combined into ⋈account.branch\_name=branch.branch\_name

Break down and move the list of projection attributes down the tree as far as possible: Πcustomer\_name is broken down into Πcustomer\_name,customer\_id on customer, Πcustomer\_id on account, and Πbranch\_name on branch

Identify sub-trees that can be executed by a single algorithm: each JOIN operation can be executed by a hash join algorithm

1. (18 points) Consider the concurrent transactions T1, T2 ,T3 and T4 under the schedule S
2. Construct the precedence graph for S. Is *S* a serializable schedule? If not, give the reason. If it is, give all serial schedules that are equivalent to S. (5 points)
3. Is S a recoverable schedule? and why? (3 points)

1. Is S a cascading or cascadeless schedule? and why? (3 points)
2. Does S obey the two-phase locking protocol? and why? (3 points)
3. Does S obey the strict two-phase locking protocol? and why? Does S obey the rigorous

two-phase locking protocol? and why? (4 points)

|  |  |  |  |
| --- | --- | --- | --- |
| T1 | T2 | T3 | T4 |
|  |  |  | Lock\_X(A) Lock\_X(C) Read(A) A:=A-100  Write(A)  Unlock(A) |
| Lock\_S(B) Read(B)  Lock\_S(Q) |  |  |  |
|  | Lock\_S(A) Lock\_X(B) Read(A) Read(B) B:=B-A  Write(B) Lock\_S(Q)  Unlock(B) |  |  |
|  |  | Lock\_S(B) Read(B)  Lock\_S(A) |  |
|  |  |  | Read(C) C:=C+100  Write(C) |
|  | Read(Q) Unlock(A)  Unlock(Q) |  |  |
|  |  | Read(A) Lock\_X(C) Read(C) C:=A+B  Write(C) Unlock(B)  Unlock(C) |  |
| Read(Q) Lock\_X(C) Read(C) C:=C+Q  Unlock(Q) |  |  |  |
|  |  |  | Unlock(C) commit |
|  | commit |  |  |
| Write(C) Unlock(C) Unlock(B)  Commit |  |  |  |
|  |  | Unlock(A)  commit |  |