**Database Systems**

Theoretical Assignment

**Course** **No**. INFR10055

**NAME**: Zhen Yi

**UUN**: s1563190

**Problem 1:**

**Analysis**: This problem let us do translation from the relational calculus query to relational algebra. At first, we should know that some notations in relational calculus are not compatible. For example, the universal quantifier is not compatible in relational algebra as well as implication notation . We need to estimate them before translation. And then, we can do the translation and get the result.

**Solution**: For calculus query , through observation, we found that the implication and universal quantifier can both be transferred to the equivalent expression.

Now, let us do it:

Firstly, we can transform the expression like below:

Secondly, continuing transforming

Finally, we got that

That is, we got an expression which is compatible to relational algebra. Amazing, right? Now, let us do the main process.

There are several rules we should know to compute above expression.

1. Negation rule: is translated into
2. Existential quantitation[[1]](#footnote-1).
3. Atomic formula[[2]](#footnote-2).
4. For can be translated into R – S.

Using above rules, life can be easier:

Because of is not union compatible. Thus, we should compute first:

Then, we can get that:

Using rule (d). At last, we get:

**Problem 2:**

**Analysis**: This problem let us use the above rule to compute the specify instance.

R= {(a1, b1), (a1, c1), (a2, c1), (a1, b2), (a3, b1), (a2, b1), (a3, b2)}

S = {b1, b2}

We just need to compute the above instance by the rule from problem 1 step by step.

**Solution**:

Firstly, we need to compute the :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A’/B’ | a1 | a2 | a3 | c1 | **b1** | **b2** |
| a1 | T | T | T | **F** | **F** | **F** |
| a2 | T | T | T | **F** | **F** | T |
| a3 | T | T | T | T | **F** | **F** |
| c1 | T | T | T | T | T | T |
| b1 | T | T | T | T | T | T |
| b2 | T | T | T | T | T | T |

The above truth table means that for each pair (A’, B’) such that means the pair which appears in **.** And we compute which can be interpreted that for each pair appears in , delete it. Thus, the value of the cell means: 1) T, true, this cell still exists. 2)F, false, doesn’t exist. The number of remaining cells are 29.

Secondly, we should compute :

|  |  |  |
| --- | --- | --- |
| A | B | C |
| c1 | b1 | b1 |
| b1 | b1 | b1 |
| b2 | b2 | b1 |
| a2 | b2 | b2 |
| c1 | b2 | b2 |
| b1 | b2 | b2 |
| b2 | b2 | b2 |

And then, it is also easy to compute , we get:

{c1, b1, b2, a2}

Finally, is that:

{a1, a2, a3} – {c1, b1, b2, a2} = {a1, a3}

To sum up, the answer is {a1, a3}.

**Problem 3:**

**Analysis**: This problem is interesting. Actually, we have a simple to way to analyze whether it is conflict serializable or not. We can use the topological sort to help us understand the order of the transactions. The reason is that once we know the correct order, we will get the possible swap of the transaction, or we can assert that such schedule is not conflict serializable.

**Solution**:

The first schedule and the second one are both the read-write schedule. They are following the rules:

1. If Op1 and Op2 refer to different data items, they do not cause a conflict, and can be swapped.
2. If they are both operations on the same data item X, then:

if both are read(X), the order does not matter;

if Op1 =read(X), Op2 =write(X), the order matters.

if Op1 =write(X), Op2 =read(X), the order matters.

if Op1 =write(X), Op2 =write(X), the order matters.

1. First schedule can be drawn a precedence graph as below:

The Nodes are the transactions in the system.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| STEP | T1 | T2 | T3 | T4 |
| 1 | READ(A) |  |  |  |
| 2 |  |  | READ(B) |  |
| 3 |  | READ(A) |  |  |
| 4 |  |  |  | READ(B) |
| 5 | READ(C) |  |  |  |
| 6 |  |  | READ(A) |  |
| 7 |  |  |  | WRITE(B) |
| 8 |  | READ(B) |  |  |
| 9 |  | READ(C) |  |  |
| 10 |  |  | READ(C) |  |
| 11 | WRITE(C) |  |  |  |
| 12 |  |  | WRITE(C) |  |

because T2 reads C in step 9 before T1 writes it in step 11.

because T1 writes C in step 11 before T3 writes it in step 12.

because T3 reads C in step 10 before T1 writes it in step 11.

because T4 writes B in step 7 before T2 reads it in step 8.

because T3 reads B in step 2 before T4 writes it in step 7.

Because the G’= {V= {T1, T2}, E= {(T1, T2), (T2, T1)}} has a cycle, thus this schedule is NOT conflict serializable.

1. Second schedule can be drawn a precedence graph as below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| STEP | T1 | T2 | T3 | T4 | T5 |
| 1 |  | Read(X) |  |  |  |
| 2 | Write(X) |  |  |  |  |
| 3 |  |  | Read(X) |  |  |
| 4 |  | Read(Y) |  |  |  |
| 5 | Write(Y) |  |  |  |  |
| 6 |  | Read(Z) |  |  |  |
| 7 |  |  | Write(Z) |  |  |
| 8 |  |  |  |  | Read(Z) |
| 9 |  |  |  |  | Write(Z) |
| 10 |  |  |  | Write(Y) |  |

because T2 reads X in step 1 before T1 writes it in step 2(T2 also reads Y in step 4 and T1 writes it in step 5);

because T1 writes X in step 2 before T3 writes it in step 3.

because T2 reads Z in step 6 before T3 writes it in step 7.

because T3 writes Z in step 7 before T5 reads it in step 8.

because T1 writes Y in step 5 before T4 writes it in step 10.

This schedule is conflict serializable. Because there are no cycles in this schedule.

Therefore, we can find that the equivalent serial schedules as follows:

**Problem 4:**

**Analysis**: This problem is complex. It has 4 parts and each part should be careful processed. All the sub questions share the U and FDs. We can do them separately. Luckily, we have some decision procedure to solve the problems.

U = {A, B, C, D, E, F, G}

F = {}

Now, let us mark the order number of them, life can be easier:

**Solution**:

1. Find the candidate keys and the prime attributes of this schema.

To find the candidate keys, we should compute the keys first. Because the keys contain the candidate keys by definition. Thus, we should compute all the closure of each attribute and the attribute appears in LHS:

The procedure to compute the closure is showing in the problem 6. I will use a simple example to explain it:

To compute . Firstly, . And because , we have ; And then we can add C into closure because of and so on. Finally, we get .

And for any attribute X which we have in LHS.

By definition, the **candidate keys are B and DG**. B must be a candidate key and both D and G are not keys, thus DG is a candidate key.

The **prime attributes are B, D and G** which appear in the candidate keys.

1. Compute minimal cover of this schema

Let us just compute it. Fortunately, we have a method to get the minimal cover:

F = {}

1. For the original set F, we need to separate the RHS first. And produce the result which only have one attribute on the RHS for every functional dependency:

Following above instruction, we can get:

F’ = {}.

1. This step is important, because we need to estimate the redundant FD in F’.

The problem is how we do the estimation? Exactly, we can use the method to iterate step by step. Every iteration, we shall do the check to judge if the specify functional dependency can be deleted or not. The thought is easy. That is, if we can delete a FD in F’ such as . We can remove it from F’ and check whether in the new F’’. If the answer is YES, we just remove it, because it is redundant. Otherwise, we cannot delete it.

The procedure shows as follows:

F’’ = {}.

1. In the new F’’, suppose F’’ = F’’ – {}, we can get . Thus, we need remain it.
2. suppose F’’ = F’’ – {}, we can get . Thus, we need delete it. F’’ = {}.
3. suppose F’’ = F’’ – {}, we can get . Thus, we need delete it. F’’ = {}.
4. suppose F’’ = F’’ – {}, we can get . Thus, we need delete it. F’’ = {}.
5. suppose F’’ = F’’ – {}, we can get . Thus, we need remain it. F’’ = {}.
6. suppose F’’ = F’’ – {}, we can get . Thus, we need remain it. F’’ = {}.
7. suppose F’’ = F’’ – {}, we can get . Thus, we need remove it. F’’ = {}.
8. suppose F’’ = F’’ – {}, we can get . Thus, we need remain it. F’’ = {}.
9. suppose F’’ = F’’ – {}, we can get . Thus, we need remain it. F’’ = {}.
10. suppose F’’ = F’’ – {}, we can get . Thus, we need remove it. F’’ = {}.
11. suppose F’’ = F’’ – {}, we can get . Thus, we need remain it. F’’ = {}.
12. suppose F’’ = F’’ – {}, we can get . Thus, we need remain it. F’’ = {}.
13. suppose F’’ = F’’ – {}, we can get . Thus, we need remain it. F’’ = {}.
14. suppose F’’ = F’’ – {}, we can get . Thus, we need remove it.

F’’ = {}.

1. Finally, we need to process LHS now. For each FD, we need to check whether some attribute in the LHS is redundant or not.

F’’’ = {}.

1. For , . Thus, remove D. F’’ = {}.
2. For , . Thus, nothing to do. F’’ = {}.
3. For , . Thus, remove C and A. F’’ = {}.
4. For , nothing to do. F’’ = {}.
5. For ,
6. For , . Thus, nothing to do. F’’ = {}.
7. For , Thus, remove E. F’’ = {}.
8. For , Thus, remove E. F’’ = {}.

And then, we need to **iterate** above procedure **again** until no more change:

{}

We find that, we should remove

Finally, we get the minimal cover here:

{}

1. Produce a 3NF decomposition

For the minimal cover F’ from (b),

* There is no in F’ with XA = U, thus we should do something else.
* Select the key B, then the output is:
* Simplification:

One of attribute-sets produced is a key.

Thus, (B, ) is not needed. Hence the result is:

We can continue to simplify:

1. Produce a BCNF decomposition of this schema. Is it dependency-preserving?

Let us use the algorithm to produce a BCNF decomposition here.

**Part I:**

Input: U and F

S = {U, }

And then we do the while loop:

* Step 1: Select U; Check that it is not in BCNF because

but

Our sets are:

X = {C, A}, Y = {D}, Z = {B, E, F, G}

Thus, after first step we have two schemas:

* Step 2: is in BCNF, there is nothing to do.

is not in BCNF: BE is not a key.

Our sets are:

X = {B, E}, Y = {G, F, A}, Z = {C}

Thus, after first step we have two schemas:

After two steps, we have:

and are all in BCNF

**Part II:**

This BCNF decomposition is not dependency-preserving.

The definition of dependency-preserving is

However, it is easy to find that . Thus, it isn’t dependency-preserving.

**Problem 5:**

**Analysis**: This problem is about the understanding of expressive power of . Exactly, we can try to compute the transform of it. Because we all know that, we can express intersection by such notations.

**Solution**:

Let us have a look of the expression . The only thing we should to process is the notation . How could we get the same thing without ? Is it possible? The answer of it is positive. Because the expressive power of is strong.

That is, we get the final solution:

They are equivalent in semantic.

**Problem 6:**

**Analysis**: There are two parts in this problem. One of them is to prove the algorithm below returns the closure of w.r.t.. To prove the correctness of one algorithm, there are several methods. The must common one is proof by contradiction. Using this method, we can solve the problem directly and quickly. Another part is to find the running time of this algorithm. That is, we need to analyze the time complexity of this problem with big O notation.

**Solution**:

***Part I:***

We can compare this model of algorithm to decision procedure. First of all, we need to interpret this pseudo-code. Exactly, the problem of this question is that we need to prove this program always return the correct answer and always terminates as well.

1. The program always returns the correct answer:

Let us use proof by contradiction here. Suppose that, the program terminates and return the wrong answer, and assume that we also know the **acceptable** **answer** is , we will compare it with our hypothesis later. The possible wrong answers which program returns can be two situations.

1. Assume that the answer which we get is , and That means there exists at least one element A that . Now, let us interpret the program. Because the program shows that the repeat-until loop will terminate if no more changes of the loop. However, if the elements A satisfies

, some will satisfy , and the loop should not terminate. However, the program returns and . This is in contradict with the definition. Thus, it is impossible get the answer .

1. Assume that the answer which we get is , and . Same to above proof. There exists at least one element A’ that .

However the closure of program is , not . If some and it satisfy implies that . But . This is in contradict with the definition. Thus, it is impossible get the answer .

Above all, if the program terminates, we will always get the acceptable answer.

1. The program should always terminate.

Suppose the program loops. Each time the repeat-until loop of program will check the condition if it no more changes or not. Because the program loops imply that the program always changes. However, the total number of set unused is finite and if it changes, such number decrease 1 every time. That means the number of set unused can always approach 0. If it reaches 0, the program must be terminated. But it does not. This is in contradict with the definition.

In conclusion, the algorithm always returns the correct answer and terminates. Thus, this algorithm returns the closure of w.r.t..

***Part II:***

To analyze the time complexity of this algorithm, we need to know the definite upper bound of the input size. The problem shows that n is the size of number of FDs. Thus, we have a portal to find out the time complexity of it.

1. What is the worst situation of every pass to find a new elements Z and add it to closure? The answer is the size of each traversal. At the beginning, the size of F is n. Thus, we may be use n times to find the element which we need, if it locates at the end of the queue. The time complexity of check and and remove and add Z to closure can be O(c) where c is constant.

On the other hand, the size of F after each traversal decreases 1 until program terminates.

1. From i, we sum all the passes using algebra:

The time complexity of the operation remove and add depend on the implementation. That means if we use the binary tree data structure, the time complexity is O(). However, we can use index here, such as Hash table and other advanced data structure. Thus, we can get the constant time complexity of them. And then it can be omitted in big O notation. Same reason to belong() operation.

Above all, the running time is .

1. Please refer to lecture handout 4 [↑](#footnote-ref-1)
2. Please refer to lecture handout 4 [↑](#footnote-ref-2)