6. Relational Algebra (Part II)

6.1. Introduction

In the previous chapter, we introduced relational algebra as a fundamental model of relational database manipulation. In particular, we defined and discussed three important operations it provides: Select, Project and Natural Join. These constitute what is called the basic set of operators and all relational DBMS, without exception, support them.

We have presented examples of the power of these operations to construct solutions (derived relations) to various queries. However, there are classes of practical queries for which the basic set is insufficient. This is best illustrated with an example. Using again the same example domain of customers and products they purchase, let us consider the following requirement:

“Get the names of customers who had purchased both product number 1 and product number 2”

All the required pieces of data are in the relations shown above. It is quite easy to see what the answer is—from the Transaction relation, customers number 1 and number 2 are the ones we are interested in, and cross-referencing the Customer relation (to retrieve their names) the customers are Codd and Martin respectively. Now, how can we construct this solution using the basic operation set?

Working backwards, the final relation we wish to construct is a single-column relation with the attribute ‘Cname’. Thus, the last operation needed will be a projection of some relation over that attribute. Such a relation must first be the result of joining Customer and Transaction (over ‘C#’), since Customer alone does not have data on products purchased. Second, it must contain only tuples of customers who had purchased products 1 and 2, i.e. some form of selection must be applied. This analysis suggests that the required sequence of operations is a Join, followed by a Select, and finally a Project.

The following then may be a possible solution:

<table>
<thead>
<tr>
<th>Customer</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>Cname</td>
</tr>
<tr>
<td>1</td>
<td>Codd</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
join Customer AND Transaction over C# giving A
select A where P# = 1 AND P# = 2 giving B
project B over Cname giving Result

The join results in:

<table>
<thead>
<tr>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
<td>2</td>
<td>23.01</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
<td>2</td>
<td>29.01</td>
<td>20</td>
</tr>
</tbody>
</table>

At this point, however, we discover a problem: the selection on A results in an empty relation!

The problem is the selection condition: no tuple can possibly satisfy a condition that requires a single attribute to have two different values (“P# = 1 AND P# = 2”). This is obvious once it is pointed out, although it might not have been so at first glance. Thus while the selection statement is syntactically correct, its logic is erroneous. What is needed, effectively, is to select tuples of a particular customer only if there exists one with P# = 1 and another with P# = 2, ie. the form of selection needed is dependent across tuples. But the basic Select operator cannot express this because it operates on each tuple in turn and independently of one another.¹

Thus the proposed solution above is not a solution at all. In fact, no combination of the basic operations can handle the query or other queries of this sort, for example:

“Get the names of customers who bought the product CPU but not the product VDU”, or
“Get the names of customers who bought every product type that the company sells”, etc

These examples suggest that additional operations are needed. In the following, we shall present them and show how they are used.

We will round up this chapter and our discussion of relational algebra with a discussion of two other important topics: how operations handle “null” values, and how sequences of operations can be optimised for performance. A null value is

¹ Some readers may have noted that if OR was used instead of AND in the selection operation, the desired result would be constructed. However, this is coincidental. The use of OR is logically erroneous—it means one or the other, but not necessarily both. To see this, change the example slightly by deleting the last tuple in Transaction and recompute the result (using OR). Your answer would still be Codd and Martin, but the correct answer should be Codd alone!
inserted into a tuple field to denote an (as yet) unknown value. Clearly, this affects the evaluation of conditions involving attribute values. Exactly how will be explained in Section 6.4. Finally, we will see that there may be several different sequences of operations that derive the same result. In such cases, we may well ask which sequence is more efficient, i.e. least costly or better in performance, in some sense. A more precise notion of ‘efficiency’ of operators and how a given operator sequence can be made more efficient will be discussed in section 6.5.

6.2. Division

As the name of this operation implies, it involves dividing one relation by another. Division is in principle a partitioning operation. Thus, \( 6 \div 2 \) can be paraphrased as partitioning a single group of 6 into a number of groups of 2—in this case, 3 groups of 2. The basic terminology used in arithmetic will be used here as well. Thus in an expression like \( x \div y \), \( x \) is the dividend and \( y \) the divisor. Division does not always yield whole groups of the divisor, e.g. \( 7 \div 2 \) gives 3 groups of 2 and a remainder group of 1. Relational division too can leave remainders but, much like integer division, we ignore remainders and focus only on constructing whole groups of the divisor.

The manner in which a relational dividend is partitioned is a little more complex. First though, we should ask what aspect of a relation is being partitioned? The answer simply is the set of tuples in the relation. Next, we ask how we decide to group some tuples together and not others? Not surprisingly, the basis for such decisions has to do with the attribute values in the tuples. Let’s take a look at an example first before we describe the process more precisely.

The illustration above shows how we may divide a relation \( R \), which is a simple binary relation in this case with two attributes \( A_1 \) and \( A_2 \). For clarity, the values of attribute \( A_1 \) have been sorted so that a given value appears in contiguous rows (where there’s more than one). The question we’re interested in is which of these values have in common an arbitrary subset of values of attribute \( A_2 \).

For example,

“which values of \( A_1 \) share the subset \( \{a,b\} \) of \( A_2 \)?”
By inspecting R, the reader can verify that the answer are the values 1 and 2, because only tuples with these A1 values have corresponding A2 entries of both ‘a’ and ‘b’. Put another way, the tuples of R are grouped by the common denominator or divisor {a,b}. This is shown in the relation R’ where we emphasise the groups formed using double-line borders. Other tuples (the remainder of the division) are ignored. Note that R’ is not the final result of division—it is only an intermediate working result. The desired result are the values of attribute A1 in it, or put another way, the projection of R’ over A1.

From this example, we can see that a division of a relation R is performed over some attribute of R. The divisor is a subset of values from that attribute domain and the result is a relation comprising the remaining attributes of R. In relational algebra expressions, the divisor is in fact specified by another relation D. For this to be meaningful at all, D must have at least one attribute in common with the R. The division is over the common attribute(s) and the set of values used as the actual divisor are the values found in D. The general operation is depicted in the figure below.

![Figure 6-1. The Division Operation](image)

Figure 6-2 shows a simple example of dividing a binary relation R1 by a unary relation R2. The division is over the shared attribute I2. The divisor is the set {1,2,3}, these being the values found in the shared attribute in R2. Inspecting the tuples of R1, the value ‘a’ occur in tuples such that their I2 values match the divisor. So ‘a’ is included in the result. ‘b’ is not, however, as there is no tuple <b,2>.
We can now specify the form of the operation:

```
divide <dividend-relation-name> by <divisor-relation-name>
giving <result-relation-name>
```

<dividend-relation-name> and <divisor-relation-name> must be names of defined relations or results of previous operations. <result-relation-name> must be a unique name used to denote the result relation. As mentioned above, the divisor must share attributes with the dividend. In fact, we shall insist (on a stronger condition) that the intension of the divisor must be a subset of the dividend’s. This is not really a restriction as any relation that shares attributes with the dividend can be turned into the required form simply by projecting over them.

We can now show how division can be used for the type of queries mentioned in the introduction. Take the query:

“Get the names of customers who bought every product type that the company sells”

The Transaction relation records customers who have ever bought anything. For this query, however, we are not interested in the dates or purchase quantities but only in the product types a customer purchased. So we project Transaction over C# and P# to give us a working relation A. This is shown on the left side of the following illustration. Next, we need all the product types the company sells, and these may be obtained by projecting the relation Product over P# to give us a working relation B. This is shown on the right side of the illustration.
Now as we are interested in only those customers that purchased all products (ie. all the values in B), B is thus used to divide A to result in the working relation C. In this case, there is only one such customer. Finally, the details of the customer are obtained by joining C with the Customer relation over C#.

**Formal Definition**

To formally define the Divide operation, we will use the notation introduced and used in Chapter 5. However, for convenience, we repeat here principal definitions to be used.

If $\sigma$ denotes a relation, then let

- $S(\sigma)$ denote the finite set of attribute names of $\sigma$ (ie. its intension)
- $T(\sigma)$ denote the finite set of tuples of $\sigma$ (ie. its extension)
\[ \tau \alpha, \text{ where } \tau \in T(\sigma) \text{ and } \alpha \in S(\sigma), \text{ denote the value of attribute } \alpha \text{ in tuple } \tau \]

\[ S_{\text{tuple}}(x) \text{ denote the set of elements in tuple } x \]

Furthermore, if \( \tau \in T(\sigma) \), \( \tau' \) denotes a tuple, and \( S_{\text{tuple}}(\delta) \subseteq S(\sigma) \), we define:

\[ R(\tau, \delta, \tau') \equiv \forall \alpha \bullet \alpha \in S_{\text{tuple}}(\delta) \iff \tau \alpha \in S_{\text{tuple}}(\tau') \]

The Divide operation takes the form

\[ \text{divide } \sigma \text{ by } \nu \text{ giving } \rho \]

As with other operations, the input sources \( \sigma \) and \( \nu \) must denote valid relations that are either defined in the schema or are results of previous operations, and \( \rho \) must be a unique identifier to denote the result of the division. The intensions of \( \sigma \) and \( \nu \) must be such that

\[ S(\nu) \subseteq S(\sigma) \]

The Divide operation can then be characterised by the following:

- \( S(\rho) \equiv S(\sigma) - S(\nu) \)
- \( T(\rho) \equiv \{ \tau \mid \tau_1 \in T(\sigma) \land R(\tau_1, \Delta, \tau) \land T(\nu) \subseteq IM(\tau) \} \)

where

\[ S_{\text{tuple}}(\Delta) = S(\rho), \]

\[ S_{\text{tuple}}(\delta) = S(\nu), \text{ and} \]

\[ IM(\tau) = \{ t' \mid t \in T(\sigma) \land R(t, \delta, t') \land R(t, \Delta, \tau) \} \]

### 6.3. Set Operations

Relations are basically sets. We should, therefore, be able to apply standard set operations on them. To do this, however, we must observe a basic rule: a set operation on two or more sets is meaningful if the sets comprise values of the same type. This is so that comparison of values from different sets is meaningful. It is quite pointless, for example, to attempt an intersection of a set of integers and a set of names. We can still perform the operation, of course, but we can already tell at the outset that the result will be a null set because any value from one will never be equal to any value from the other.

To ensure this rule is observed for relations, we need to state what it means for two relations to comprise values of the same type. As a relation is a set of tuples, the values we are interested in are the tuples themselves. So when is it meaningful to compare two tuples for equality? Clearly, the structure of the tuples must be identical, i.e. the tuples must be of equal length and their corresponding elements must be of the same type. Only then can two tuples be equal, i.e. when their corresponding element values are equal. The structure of a tuple, put another way, is in fact the intension or schema of the relation it occurs in. Thus, meaningful set operations on relations
require that the source relations have identical intensions/schemas. Such relations are said to be *union-compatible*.

The set operations included in relational algebra are Union, Intersection, and Difference. Keeping in mind that they are applied to whole tuples, these operations behave in exactly the standard way. It goes without saying that their results are also relations with intensions identical to the source relations.

The Union operation takes the form

\[
\text{<source-relation-1> union <source-relation-2> giving <result-relation>}
\]

where \(<source-relation-i>\) are valid relations or results of previous operations and are union-compatible, and \(<result-relation>\) is a unique identifier denoting the resulting relation.

Figure 6-3 illustrates this operation.

```
\begin{array}{c|c|c}
R1 & \text{Union} & R2 \\
\hline
1 & \text{giving} & 2 \\
a & R1 & b \\
b & R2 & c \\
c & \text{R3} & d \\
\end{array}
```

**Figure 6-3** Relational Union Operation

The Intersection operation takes the form

\[
\text{<source-relation-1> intersect <source-relation-2> giving <result-relation>}
\]

where \(<source-relation-i>\) are valid relations or results of previous operations and are union-compatible, and \(<result-relation>\) is a unique identifier denoting the resulting relation.

Figure 6-4 illustrate this operation.

```
\begin{array}{c|c|c}
R1 & \text{Intersect} & R2 \\
\hline
1 & \text{giving} & 2 \\
a & R1 & b \\
b & R2 & c \\
c & \text{R3} & d \\
\end{array}
```

**Figure 6-4** Relational Intersection Operation
The Difference operation takes the form

\[ <\text{source-relation-1}> \text{ minus } <\text{source-relation-2}> \text{ giving } <\text{result-relation}> \]

where \(<\text{source-relation-}\text{i}>\) are valid relations or results of previous operations and are union-compatible, and \(<\text{result-relation}>\) is a unique identifier denoting the resulting relation.

Figure 6-5 illustrate this operation.

As an example of the need for set operations, consider the query: “which customers purchased the product CPU but not the product VDU?”

The sequence of operations to answer this question is quite lengthy, but not difficult. Probably the best way to construct a solution is to work backwards and observe that if we had a set of customers who purchased CPU (say W1) and another set of customers who purchased VDU (say W2), then the solution is obvious: we only want customers that appear in W1 but not in W2, or in other words, the operation “W1 minus W2”.

The problem now has been reduced to constructing the sets W1 and W2. Their constructions are similar, the difference being that one focuses on the product CPU while the other the product VDU. We show the construction for W1 below.
The above Join operation is needed to bring in the product name into the resulting relation. This is then used as the basis of a selection, as shown on the right.

Y1 now has only customer numbers that purchased the product CPU. As we are interested only in the customers and not other details, we perform the projection on the right.

Finally, details of such customers are obtained by joining Z1 and Customer, giving the desired relation W1.

The construction for W2 is practically identical to that above except that the selection operation specifies the condition “Pname = VDU”. The reader may like to perform these steps as an exercise and verify that the following relation is obtained:
Now we need only perform the difference operation “W1 minus W2 giving Result” to construct a solution to the query:

<table>
<thead>
<tr>
<th></th>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
</tr>
</tbody>
</table>

**Formal Definition**

If \( \sigma \) denotes a relation, then let

- \( S(\sigma) \) denote the finite set of attribute names of \( \sigma \) (ie. its intension)
- \( T(\sigma) \) denote the finite set of tuples of \( \sigma \) (ie. its extension)

The form of set operations is

\[
\sigma <\text{set operator}> \lambda \quad \text{giving} \quad \rho
\]

where \(<\text{set operator}>\) is one of ‘union’, ‘intersect’ or ‘minus’; \( \sigma, \lambda \) are source relations and \( \rho \) the result relation. The source relations must be union-compatible, ie. \( S(\sigma) = S(\lambda) \).

The set operations are characterised by the following:

- \( S(\rho) = S(\sigma) = S(\lambda) \) for all \(<\text{set operator}>\)s
- for ‘union’
  \[
  T(\rho) \equiv \{ t \mid t \in T(\sigma) \lor t \in T(\lambda) \}
  \]
- for ‘intersect’
  \[
  T(\rho) \equiv \{ t \mid t \in T(\sigma) \land t \in T(\lambda) \}
  \]
- for ‘minus’
  \[
  T(\rho) \equiv \{ t \mid t \in T(\sigma) \land t \notin T(\lambda) \}
  \]

**6.4. Null values**

In populating a database with data objects, it is not uncommon that some of these objects may not be completely known. For example, in capturing new customer information through forms that customers are requested to fill, some fields may have been left blank (some customers may take exception to revealing their age or phone numbers!). In these cases, rather than not have any information at all, we can still record those that we know about. But what value do we insert into the unknown fields
of data objects? Leaving a field blank is not good enough as it can be interpreted as an empty string which may be a valid value for some domains. We need a value that denotes ‘unknown’ and that cannot be confused with valid domain values.

It is here that the Null value is used. We can think of it as a special value different from any other value from any attribute domain. At the same time, we may think of it as belonging to every attribute domain in the database, i.e. it may appear as a value for any attribute and not violate any type constraints. Syntactically, different DBMSs may use different symbols to denote null values. For our purposes, we will use the symbol ‘?’.

How do null values affect relational operations? All relational operations involve comparing values in tuples, including Projection (which involves comparison of result tuples for duplicates). The key to answering this question is in how we evaluate boolean operations involving null values. Thus, for example, what does “? > 5” evaluate to? The unknown value could be greater than 5. But then again, it may not be. That is, the value of the boolean expression cannot be determined on the basis of available information. So perhaps we should consider the result of the comparison as unknown as well?

Unfortunately, if we did this, the relational operations we’ve discussed cease to be well-defined! They all rely on comparisons evaluating categorically to one of two values: TRUE or FALSE. For example, if the above comparison (“? > 5”) was generated in the process of selection, we would not know whether to include or exclude the associated tuple in the result if we were to admit a third value (UNKNOWN). If we wanted to do that, we must go back and redefine all these operations based on some form of three-valued logic.

To avoid this problem, most systems that allow null values simply interpret any comparison involving them as FALSE. The rationale is that even though they could be true, they are not demonstrably true on the basis of what is known. That is, the result of any relational operation conservatively includes only tuples that demonstrably satisfy conditions of the operation. Adopting this convention, all the operations defined previously still hold without any amendment. Some implications on the outcome of each operation are considered below.

For the Select operation, an unknown value cannot identify a tuple. This is illustrated in Figure 6-6 which shows two Select operations applied to the relation R. Note that between the two operations, the selection criteria ranges over the entire domain of the attribute I2. One would expect therefore, that any tuple in R1 would either be in the result of the first or the second. This is not the case, however, as the second tuple in R1 (<b,?>) is not selected in either operation—the unknown value in it falsifies the selection criteria of both operations!
For Projection, tuples containing null values that are otherwise identical are not considered to be duplicates. This is because the comparison “? = ?”, by the above convention, evaluates to FALSE. This leads to the situation as illustrated in Figure 6-7 below. The reader should note from this example that the symbol ‘?’, while it denotes some value much like a mathematical variable, is quite unlike the latter in that it’s occurrences do not always denote the same value. Thus “? = ?” is not demonstrably true and therefore considered FALSE.

In a Join operation, tuples having null values under the common attributes are not concatenated. This is illustrated in Figure 6-8 (“?=1”, “1=?” and “?=?” are all FALSE).
In Division, the occurrence of even one null value in the divisor means that the result will be an empty relation, as any value in the dividend’s common attribute(s) will fail when matched with it. This is illustrated in Figure 6-9 below. Note, however, that this is not necessarily the case if only the dividend contains null values under the common attribute(s)—division may still be successful on tuples not containing null values.

In set operations, because tuples are treated as a single unit in comparisons, a single rule applies: tuples otherwise identical but containing null values are considered to be different (as was the case for Projection above). Figure 6-10 illustrates this for each set operation. Note that because of the occurrence of null values, the tuples in R2 are not considered duplicates of R1’s tuples. Thus their union simply collects tuples from both relations; subtracting R2 from R1 simply results in R1; and their intersection is empty.

Figure 6-8 Joining over null values

Figure 6-9 Division with null divisors

Figure 6-10 Set operations involving null values
6.5. Optimisation

Each relational operation entails a certain amount of work: retrieving a tuple, examining a tuple’s attribute values, comparing attribute values, creating new tuples, repeating a process on each tuple in a relation, etc. For a given operation, the amount of work clearly varies with the cardinality of source relation(s). For example, a selection performed on a relation twice the cardinality of another (of the same degree) would involve twice as much work.

We can also compare the relative amount of work needed between different operations based on the number of tuples processed. An operation with two source inputs, for example, need to repeat its logic on every possible tuple-pair formed by taking a tuple from each input relation. Thus if we had two relations of cardinalities M and N respectively, a total of M×N tuple-pairs must be processed, ie. M (or N) times more than, say, a selection operation on each individual relation. Of course, this is not an exact relative measure of work, as there are also differences in the amount of work expended by different operations at the tuple level. By and large, however, we are interested in the order of magnitude of work (rather than the exact amount of work) and this is fairly well approximated by the number of tuples processed.

We will call such a measure the efficiency of an operation. Thus, the efficiency of selection and projection is the cardinality of its single input relation, while the efficiency of join, divide and set operations is the product of the respective cardinalities of their two input relations.

Why should the efficiency of operations interest us? Consider the following sequence of operations:

```
join Customer AND Transaction over C# giving X;
select X where CCity = “London” giving Result
```

Suppose the cardinality of Customer was 100 and that of Transaction was 1000. Then the efficiency of the join operation is 100×1000 = 100000. The cardinality of X is 1000 (as it is certainly intended that the C# in every Transaction tuple matches a C# in one of the Customer tuples). Therefore, the efficiency of the selection is 1000. As these two operations are performed one after another, the efficiency of the entire sequence of operations is naturally the sum of their individual efficiencies, ie. 100000+1000 = 101000.

Now consider the following sequence:

```
select Customer where CCity = “London” giving X;
join X AND Transaction over C# giving Result
```

The reader can verify that this sequence is relationally equivalent to the first, ie. they produce identical results. But how does its efficiency compare with that of the first? Let us calculate using the same assumptions about the cardinalities. The efficiency of the selection is 100. To estimate the efficiency of the join, we need to make an assumption on the cardinality of X. Let’s say that 10 customers live in London. Then
the efficiency of the join is $10 \times 1000 = 10000$, and the efficiency of the sequence as a whole is $100 + 10000 = 10100$—ten times more efficient than the first!

Of course, the reader may think that the assumption about X’s cardinality was contrived to give this dramatic performance improvement. The point, however, is that the second sequence can do no worse than the first, i.e. if all customers in the Customer relation live in London, then it performs as poorly as the first. More likely, however, we expect a performance improvement.

The above example illustrates a very important point about relational algebra: there can be more than one (sequence of) expression that describe a desired result. The main aim of optimisation, therefore, is to translate a given (sequence of) expression into its most efficient equivalent form. Such optimisation may be done manually by a human user or automatically by the database management system. Automatic optimisation may in fact do better because the automatic optimiser has access to information that is not readily available to a human optimiser, e.g. current cardinalities of source relations, current data values, etc. But the overwhelming majority of relational DBMS’s available today merely execute operations requested by users as is. Thus, it is important that users know how to perform optimisations manually.

For manual optimisation, it is perhaps less important to derive the most efficient form of a query than to follow certain guidelines, heuristics or rules-of-thumb that lead to more efficient expressions. Frequently the latter will lead to acceptable performance and expending more effort to find the optimal expression may not significantly improve that performance if good heuristics are used. There is, in fact, a simple and effective rule to remember when writing queries: delay as long as possible the use of expensive operations! In particular, we should wherever possible put selection ahead of other operations because it reduces the cardinality of relations. Figure 6-11 illustrate the application of this principle. The reader should be able to verify that the two sequences of operations are logically equivalent and that intuitively the selection operations before the joins can significantly improve the efficiency of the query.

<table>
<thead>
<tr>
<th>Query:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Get names of customers who bought product CPU earlier than 24.01.</td>
<td></td>
</tr>
</tbody>
</table>

| Join Customer And Transaction Over C# Giving X; | Select Product Where |
| Join X And Product Over P# Giving Y; | Pname=CPU Giving P1; |
| Select Y Over Pname=CPU And Date<24.01 Giving W; | Select Transaction Where |
| Project W Over Cname Giving Result; | Date<24.01 Giving T1; |
| ✅ Good solution! | Join Customer And T1 Over C# Giving X; |
| | Join X AND P1 Over P# Giving W; |
| | Project W Over Cname Giving Result; |
| | ✅ Good solution! |
Figure 6-11 Delay expensive operations