## Chapter 5.6 Selection of Connection Sequence

In this Chapter, we take key problem into consideration: Choose Sequence for Join which relates to more than three Relations.

### Chapter 5.6.1 The Meaning of Left and Right Connection Parameters

***Introduction:***

When choose the sequence for Join, then we need to remember that Join Operator is mostly not symmetrical, therefore, seen from this kind of meaning, two represented Relations are totally different, the Join cost depends on which Relation represents which meaning.

***Example:***

For one trip Join, it read the smaller Relation into Main Memory, and form one type of structure, it called Hash Table, therefore it can be used conveniently to match tuples from other Relation. Then read other Relation, one block for each time, and start Join Operator for tuples in the Block and in the Main Memory.

***Definition:***

*Prerequisite:*

When try to choose one Physical Plan, we decide to use *One-Trip Join*.

* Choose the Smaller Relation and save it into Main Memory, then this kind of Relation is called *Construction Relation*.
* For Right Join Parameter, it is called *Query Relation*, and we try to match tuple in the Block with the already Saved Tuple in Main Memory.

***Here other Join Algorithm that are divided by parameters, included:***

1. Inner Loop Join, in that kind of Join, the left Parameter is the outer Loop Relation.
2. Index Join, in that kind of Join, there should have index in the Right Parameter.

### Chapter 5.6.2 Connection Tree

***Situation:***

When we try to Join two Relations, then we need to sort all Parameters. According to the normal custom, we need to choose the smaller Relation as the left parameter. The size of each parameter is important and has different meaning. This is normal, since the Query that relates to Join needs to choose at least one Attribute, and the Selection Operator makes estimation decreasing.

***Example:***

*For Query:*

SELECT movieTitle

FROM StarsIn, MovieStar

WHERE starName = name AND

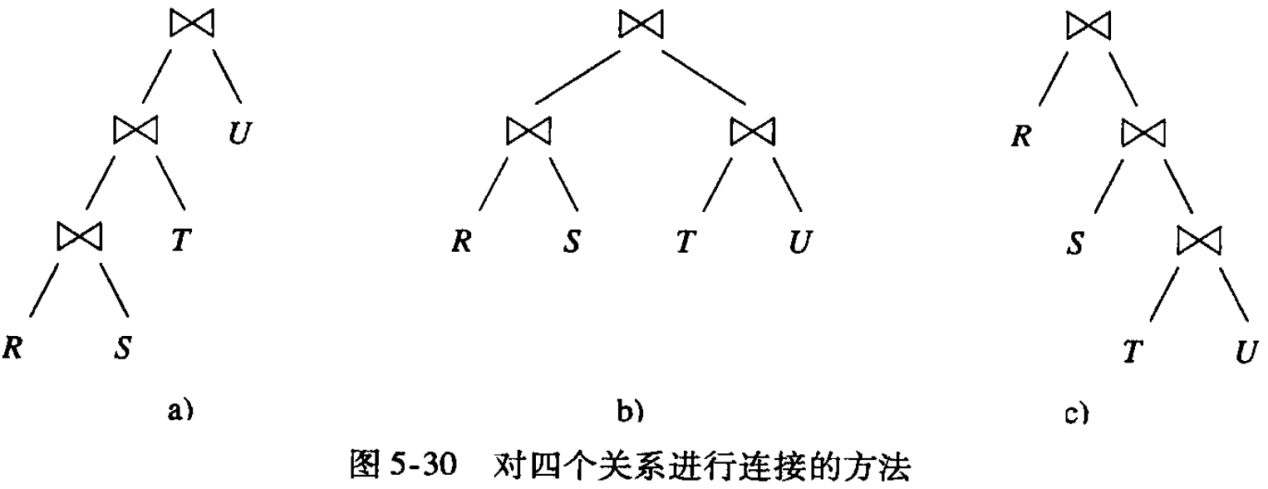
Birthdate LIKE ‘%1960’;

This is an example that we try to Join Relations include StatsIn and MovieStar, but there have not include any estimation size of Relations. We can think that one Selection will get almost 1/50 tuples among Relation MovieStar.

But since each Movie will include several Movie Stars, therefore we can assume that the size of StarsIn is far bigger than the size of MovieStar, and the second parameter of Join Operation *Selection ( birthdate LIKE ‘%1960’ ) MovieStar* is far less than the first Relation StarsIn.

As the result, we choose the MovieStar as the left Parameter which is different from the connection tree in 5 - 24.

When there have two Relations, and we can only choose one Relation as the left Parameter. When the choice relates to more than two Relations, and the number of Connection Tree will be increased quickly. For four Relations R, S, T and U, we try to connect these four trees and the possible tree structure as below. Normally, we will get 24 different trees.



### Chapter 5.6.3 Left Deep Connection Tree

### ***Definition:***

If Every Right Child in the Binary Tree, then it is Left Deep Connection Tree. Also, If Every Left Child in the Binary Tree, then it is Right Deep Connection Tree. There also has the situation that all trees are not the Left Deep Tree and not the Right Deep Tree.

***Advantage:***

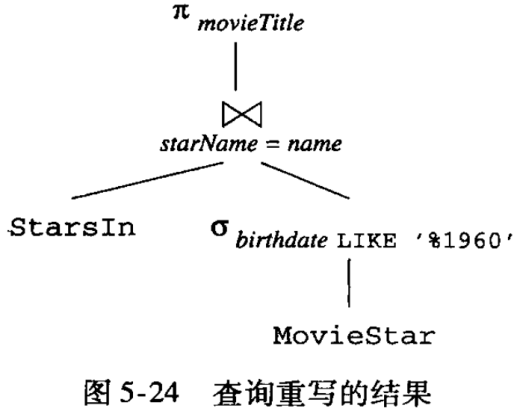
We will discuss advantages for Left Deep Connection Tree:

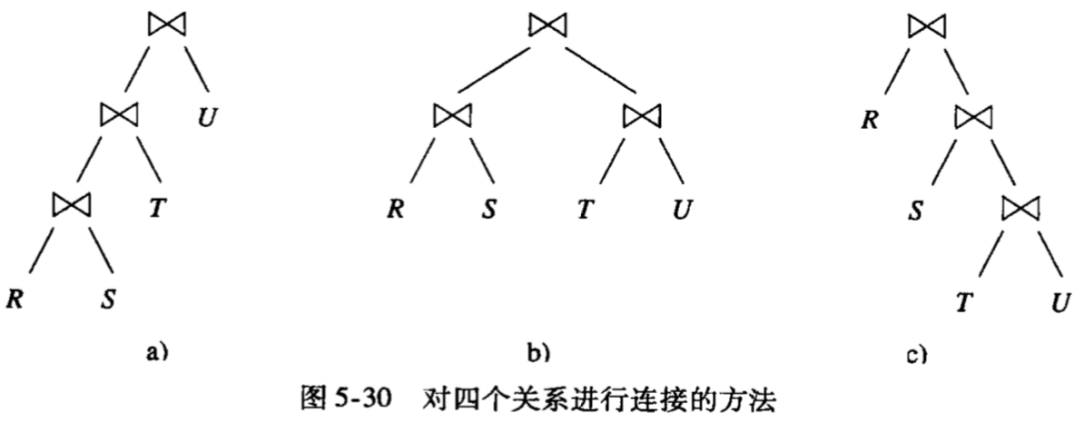
* For all possible Left Deep Connection Trees with given number of Tree Leaves, it maybe very big but it not such big like all other trees. So if we limit Query into Left Deep Connection Tree, then Query Plan can be used for bigger Query.
* Left Deep Connection Tree can be used well to interact with Normal Join Algorithm, especially Inner Loop Join and One - Trip Join. The Query Plan based on Left Deep Connection Tree would be more effective than non - Left Deep Connection Tree.

Actually, one tree node in Left Deep Connection Tree and Right Deep Connection Tree can be all Inner Tree Nodes with all other Operators except Join Operator.

***Example:***

Take the 5-24 as an example, this is actually Left Connection Tree, although the Selection Operator has been applied on the Right - Hand.





Here, for the Multi - Join with given numbers Relations, the number of Left Deep Connection Tree will not increase as fast as all other trees.

***Calculation:***

* For n Relations by using the Left Deep Connection Tree, we can have only one Type, and we can use n! to assign all Relations.
* For n Relations by using the Right Deep Connection Tree, the number will be the same as the Left Deep Connection Tree.
* For n Relations, the number of Tree Shape with n Relations can be decided by using Recursion below:

*T ( 1 ) = 1*

*T ( n ) = T ( i ) \* T ( n - i ) ( i ranges from 1 ~ n - 1. )*

***Explanation for Calculation Expression above:***

* For the second Equation, we can choose i from 1 to n - 1 randomly as the tree leave numbers and these leaves can be arranged by using the random arrangement method with i tree leaves, actually this can be selected from T ( i ) arrangement.
* Similarly, for the left n - i tree leaves, we can use one of T ( n - i ) method to decide it’s arrangement.

***Value of T ( n ):***

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| n | 1 | 2 | 3 | 4 | 5 | 6 |
| T ( n ) | 1 | 1 | 2 | 5 | 14 | 42 |

Here, we use T ( n ) \* n ! to decide all Tree numbers by using Tree Leaves. Therefore, 6 tree leaves and Tree Number with all tree leaves are signified is 42 \* n !, which is to say, 30 240, among which 6 ! is the number of Left Deep Tree, also 6 ! is the number of Right Deep Tree.

Now, let’s consider the second advantage of Left Deep Connection Tree: useful for the effective Plan.

* In One - Trip Join, and Construction Relation is in the left, then any time, the main memory that required will be smaller than using Right Deep Connection Tree and Bushy Tree.
* If using Inner Loop Cycle, and Outside Cycle Relation is on the left, then we can avoid constructing Random Intermediate Relation more than once.

***Example 1 - Calculate Utilized Size of Left Deep Connection Tree:***

Consider the Left Deep Connection Tree in the 5 - 30 (a) above, and assume for each of all three Relations, we simply Join for each Operator.

Like the most of situation, normally we use Left Parameter as Construction Relation, which means that Left Parameter would be stored in the main memory. We need to calculate R Join S, then keep Relation R into main memory, and during calculation, we need to keep result in the main memory.

To sum up, we need to keep B( R ) + B( R Join S ) size main memory. If we need to choose the smallest Relation as R, and choose an Operator to make Relation R smaller, then we can get all required buffer areas.

After Join Relation R and S, we need to continue Join with Relation T. However, the required Buffer area for Relation R is no longer needed any more, we can keep it to store the result of (R Join S) Join T.

Similar, Join result with Relation U, then use the Buffer Area for ( R Join S ) can be used to keep R Join S. Generally, the main memory required for One - Trip Join by using the Left Deep Connection Tree is at most the space for two temporary Relations.

***Example 2 - Calculate Utilized Size of Right Connection Tree:***

Let’s consider the realization of Right Deep Connection Tree. First load Relation R into main memory, since the Left Parameter is used as Construction Relation.

We construct S Join ( T Join U ) and use it as the Query Relation and put Relation S into the Buffer Memory, and we need to construct T Join U as the Query Relation, for T Join U, we need to read Relation T into main buffer. So, at that time, Relation R, S, T are all kept in the main memory.

Generally, if we tried to calculate one Right Deep Connection Tree with n tree leaves, we need read n - 1 Relations into main memory.

***Conclusion:***

However, the whole size of B ( R ) + B ( S ) + B ( T ) maybe smaller than the space that required when calculating random two Relations for Left Deep Connection Tree, and the space maybe B ( R ) + B ( R Join S ) and B( R Join S ) + B ( ( R Join S ) Join T ).

***Especially:***

When Relation R is small, then Relation R Join Relation S could be much smaller than S, and ( R Join S ) Join T could be smaller than T, which prove that Left Deep Connection Tree is pretty useful.

***Example 3 - Calculate Utilized Size of Inner Loop Join***

Now, let’s consider by using Utilized Size of Inner Loop Join to realize forth - trip - join, also for three Join would assign one iterator. Otherwise, in order to simplify the process, assume that Relation R, S, T and U are all stored Relation but not Expression.

By using Left Deep Connection Tree, the iterator will get the main memory size block for Left Parameter (R Join S) Join T, as long as the Relation U is the stored Relation, then it will try to connect this block with all Relation Block in U, the iterator tries to scan Relation U but not need to construct it. Once it get the next block for the left Parameter, then it puts it into main memory, and try to read U again. But we need to pay attention that if the parameters in two sides are both large, then Inner Loop Join can not avoid such repetition.

Similarly, when we try to get the block of ( R Join S ) Join T, we need to put R Join S into main memory and start scan Relation T. It is necessary to scan Relation T, and can not be avoided.

Now compare the cycle behavior of Left Deep Connection Tree with the cycle behavior of Right Deep Connection Tree. The cycle begins with read Relation R into main memory. Then we must construct all Relations S Join ( T Join U ) and compare it with Relation R. Then when we read the block of Relation R into main memory, then we must construct all Relations S Join ( T Join U ) again. Every block for Relation R needs to construct such Relation.

Of course, we need to construct S Join ( T Join U ), and store it in main memory or disk. If we store it into the disk, then we need to use extra disk I/O when compared with Left Deep Connection Tree; Otherwise, if we store it into the main memory, then it will definitely have some overuse main memory problem.

### Chapter 5.6.4 Choose Connection Sequence and Grouping by Dynamic Programming

### ***Prerequisite:***

### When we need to choose the connection sequence for multi Relations, we have three options:

### Considerate all

### Considerate one sub-class

### Select one Connection Sequence by using Heuristic Method.

### Here, we need to take the Dynamic Programming to enumerate and it can be used as or consider the sequence, or just considerate the specific sequence, such as limit the Left Deep Connection Tree Sequence. The thinking pattern of Dynamic Programming is that: we need to fill with an Cost Table, only to remember the least information that required for the conclusion.

***Definition:***

Assume that we want to Join Relations such as R1 Join R2 Join Rn. In the Dynamic Programming, we need to construct one table which include one or more Relations among n Relations. In this table, includes:

1. The size estimation for Relation Join.
2. The least cost when calculate Relations Join.
3. The least cost Expression. This Expression groups and joins already existed Relations.

The construction of this table is to generalize the current Sub - Collection. Two variables, which are depended on the whether we want to consider all possible tree structure or just consider the Left Deep Connection Tree.

***Basics:***

One single Relation includes the size of Relation R, value 0 cost and Relation R Expression. It’s also easy to calculate one pair Relation { Ri, Rj }. When the cost equals to 0, since there has not relates to intermediate Relations, and the size estimation is given by the public rule - it is just given by the Ri \* Rj / max\_num (common properties), if there has any common property. However, the Expression is Ri Join Rj, otherwise it is Rj Join Ri. Then we need to pick the smaller one as the our left parameter.

***Conclusion:***

Now, we can construct the table and calculation table for the size of 3, 4 or so, till we can get all sub - collection for the size of n. The table tells us the best calculation for Join all Relations and the estimation Cost for this method, this value can be used afterwards. We need to calculate the collection that includes k Relations in collection R.

***Calculation:***

First, we consider the Left Deep Connection Tree, for each Relation R in R Collection, we need first take Collection R - R into consideration and then rejoin Relation R again. Then the cost would be the addition of these two part costs.

Here we need to choose the least cost Relation R, for the Expression with the least cost in Collection R - R as final Join Left Parameter, and use Relation R as the Right Parameter.

If we want to take all trees into consideration, then the calculation of Relation Collection R would be far more complex. We need to consider all possible situations that divide Collection R into Sub - Collection R1 and Sub - Collection R2. For each sub - collection, we need to consider the sum of these two fields:

* The best cost for Collection R1 and Collection R2.
* The result of Collection R1 and Collection R2.

For the division of the best cost, we use this value to

***Example:***

### Chapter 5.6.5 Dynamic Programming with Specific Cost Function

### Chapter 5.6.6 Choose Connection Sequence by Greedy Algorithm