Joins - Types and strategies

In this lecture

- Types of joins (inner, left, outer, cross).
- Strategies: how Teradata does this joins.
- Improving join performance.

Different types of joins

There are different types of Joins available.

- Inner Join
- Left Outer Join
- Right Outer Join
- Full Outer Join
- Self Join
- Cross Join
- Cartesian Production Join

Example

`Employee`

EmpNo	FName	LName	JDate	DeptNo	DoB
101	Mike	James	3/27/2015	1	1/5/1980
102	Robert	Williams	4/25/2017	2	3/5/1983
103	Peter	Paul	3/21/2017	2	4/1/1983
104	Alex	Stuart	2/1/2018	2	11/6/1984
105	Robert	James	1/4/2018	3	12/1/1984

Example (cont.)

`Salary`

EmpNo	Gross	Deduction	NetPay
101	40,000	4,000	36,000
102	80,000	6,000	74,000
103	90,000	7,000	83,000
104	75,000	5,000	70,000

INNER JOIN

```
SELECT A.EmpNo, A.DeptNo, B.NetPay
FROM
Employee A
INNER JOIN
Salary B
ON (A.EmpNo = B. EmpNo);
```

```
/* Output: */
EmpNo DeptNo NetPay
------------------
101     1       36000
102     2       74000
103     2       83000
104     2       70000
```

`OUTER JOIN`

- `LEFT OUTER JOIN` returns all the records from the left table and returns only the
  matching records from the right table.
- `RIGHT OUTER JOIN` returns all the records from the right table and returns only
  matching rows from the left table.
- `FULL OUTER JOIN` combines the results from both `LEFT OUTER` and `RIGHT OUTER`.

`OUTER JOIN` (cont.)

```
SELECT A.EmpNo, A.DeptNo, B.NetPay
FROM
Employee A
LEFT OUTER JOIN
Salary B
ON (A.EmpNo = B. EmpNo);
```

```
/* Output: */
EmpNo DeptNo NetPay
------------------
101     1       36000
102     2       74000
103     2       83000
104     2       70000
105     3       ?
```

CROSS JOIN

```
SELECT A.EmpNo, A.DeptNo, B.NetPay
FROM
Employee A
CROSS JOIN
Salary B
WHERE A.EmployeeNo=101
ON (A.EmpNo = B. EmpNo);
```

```
/* Output: */
EmpNo    DeptNo  EmpNo    NetPay
------------------------------
101      1       101      36000
101      1       104      70000
101      1       102      74000
101      1       103      83000
```

Processing joins

- The rows that are to be joined from each table **must** be in the physical AMP.
- To ensure this, the Optimizer creates a query execution plan that should be executed on each AMP. In particular, to specify:
  - **Table redistribution**.
  - **Table duplication**.
  - **Table local sorting**.
- Understanding this process helps us improve it.

`JOIN` strategies

Strategies for joining tables

- Join Strategies are used by the optimizer to choose the best plan to join tables based on the given join condition.
  - Merge (Exclusion)
  - Nested
  - Row Hash
  - Product (including Cartesian Product joins)

Merge (Exclusion)

- It is adopted when the join conditions are based on equality (=).
- There is a prerequisite though: the two tables must be sorted based on the join column in advance (actually sorted based on the join column row hash sequence).
- That brings a great advantage for this type of join: both tables only need to be scanned once, in an interleaved manner.

Requirements:

- The rows to be joined have to be located on a common AMP
- Both spools have to be sorted by the ROWID calculated over the join column(s)

Merge (cont.)

- **Process:**
  - The ROWHASH of each qualifying row in the left spool is used to look up matching rows with identical ROWHASH in the right spool (by means of a binary search as both spools are sorted by ROWID)
- Possible Join Preparations required:
  - Re-Distribution of one or both spools by ROWHASH or Duplication of the smaller spool to all AMPs
  - Sorting of one or both spools by the ROWID

Merge: data redistribution and duplication

- While joining two tables the data will be redistributed or duplicated across all AMPs to make sure joining rows are in the same AMPs.
- Relocation of rows to the common AMP can be done by redistribution of the rows by the join column(s) ROWHASH or by copying the smaller table as a whole to all AMPs.
- If one table PI is used and Other table PI not used, redistribution/duplication of the table will happen based on the table size. In these cases Secondary Indexes will be helpful.

Case 1 – PI = PI joins

- The Primary Indexes (or any other suitable index) of both tables equals the join columns:
  - There is no redistribution of data over AMP’s. Since AMP local joins happen as data are present in same AMP and need not be re-distributed.
  - These types of joins on unique primary index are very fast.
  - No join preparation is needed as the rows to be joined are already on the common AMP.

Case 2 – PI = non Index joins

- The rows of the second table have to be relocated to the common AMP data from second table will be re-distributed on all AMPs.
- Ideal scenario is when the small table gets redistributed.
    i. Duplicate all rows of one table onto every AMP (The duplication of all rows is done when the non-PI column is on a small table),
    ii. Redistribute the rows of one table by hashing the non-PI join column and sending them to the AMP containing the matching PI row,

Case 3 – non Index = non Index joins

- Neither the Primary Index of the first table (or any other suitable index) nor the Primary Index (or any other suitable index) of the second table matches the join columns:
- Data from both the tables are redistributed on all AMPs.
- This is one of the **longest processing queries**: you should collect stats in these columns.
- Redistribute both tables by hashed join column value

Nested Join

- Nested Join is the most efficient join method in Teradata.
- It is also the only join method that does not always use all the AMPs.

In order to make Nested Join picked, the following conditions must be satisfied:

1. The join condition is based on equality.
2. The join column is a unique index on one table.
3. The join column is any index on another table.

Nested Join: Example

```
SELECT emp.Ename , dep.Deptno, emp.salary
FROM
employee emp ,
department dep
WHERE emp.Enum = dep.Enum AND dep.Enum=2345;
```

Hash join

- Hash Join is also based on equality condition (=).
- This strategy gets its name from the fact that one smaller table is built as *hash-table*, and potential matching rows from the second table are searched by hashing against the smaller table.
- Usually optimizer will first identify a smaller table, and then sort it by the join column row hash sequence.

Hash join (cont.)

- If the smaller table is really small and can fit in the memory, the performance will be best. Otherwise, the sorted smaller table will be duplicated to all the AMPs.
- Then the larger table is processed one row at a time by doing a binary search of the smaller table for a match.
- Faster than merge joins since the large table does not need to be sorted.

Exclusion Join

- This join strategy is used to find non-matching rows.
- If the query contains `NOT IN` or `EXCEPT`, `MINUS` and/or set substraction operations, exclusion join will be picked.
- This kind of join can be done as either Merge Join or Product Join.

Exclusion Join: Example

```
SELECT
emp.Ename , DEP.Deptno, EMP.salary
FROM
EMPLOYEE EMP
WHERE
EMP.Enum
NOTIN
  ( Select Enum from
      DEPARTMENT DEP
      where Enum IS NOT NULL );
```

Product join

- to find a match between two tables with a join condition which is not based on equality (>, <, <>), or join conditions are ORed together.
- The reason why we call it “Product” join is that, the number of comparisons required is the “product” of the number of rows of both tables.
- For example, table t1 has 10 rows, and table t2 has 25 rows, then it would require 10×25=250 comparisons to find the matching rows.
- When the WHERE clause is missing, it will cause a special product join, called Cartesian Join or Cross Join,

What is the default join strategy in Teradata?

- There is no *default* join strategy.
- Optimizer decides the type of strategy based on the best retrieval path and other parameters to execute the query.
- These parameters include data demographics, statistics and indexes if any of them are available.

What is the default join strategy in Teradata? (cont.)

- Using `EXPLAIN` can help find out what join strategies are to be adopted.
- No matter which join strategy, it is always applied between two tables. The more tables, the more join steps.
- Rows must be on the same AMP to be joined, so row distribution or duplication is sometimes unavoidable.

Example

- There is no default, but you can influence the selection of join strategy with your query.

```
database tutorial;

/* Product join*/
select CustomerName, StoreId
from Customer, SalesTransaction;

/* Merge join*/
select c.CustomerName, s.StoreId
from Customer c
join SalesTransaction s
on c.CustomerId = s.customerid;
```

Improvement tricks

Identify skewness in joins

- Find out if there are tables with skewed data.
- Split the query in skew and non-skew parts

Example

```
/* UNION ALL To reduce skew*/
SELECT cust_id, name
(
sel cust_id, val
from skew_tableA A, nonskew_tableB B
on A.cust_id=B.cust_id
where a.cust_id=<skew_value>
UNION ALL
sel cust_id, val
from skew_tableA A, nonskew_tableB B
on A.cust_id=B.cust_id
where a.cust_id not in (<skew_value>)
)a
GROUP BY 1,2
```

Find the right columns to join

Nature of Join	How Join Occurs
PI to PI	No redistribution. Local AMP join.
PI to Non PI	Non PI table will be redistributed. Smaller table should be duplicated on AMPs.
Non PI to Non PI	Tables will be redistributed based on joining columns. Statistics play important role.
PPI to PPI	Row key based merge join. Only partitions are joined.

Nature of Join	How Join Occurs
PPI to NPPI	Sliding window merge join. It is like a product join for partitioned based tables.
Unmatched datatype	Conversion of data type; translation will occur.
Use of Function	**Statistics could be of no use**. Each value in a column will be computed, which will increase CPU. Table can go for full table scans.
Skewed values	Redistribution or duplication of table gets impacted. NULL values, if there, need to filtered.
Columns with no or stale stats	Columns of skew values and with millions of rows should have statistics. Can result in product join.

Eliminate Product joins

```sql
database tutorial;
/* Accidental product join*/
select CustomerName, StoreId
from Customer, SalesTransaction;

/*Merge join*/
select c.CustomerName, s.StoreId
from Customer c
join SalesTransaction s
on c.CustomerId = s.customerid;
```

Improve left join performance

- Filters for the INNER tables are placed on the ON clause; for the OUTER table are placed in the WHERE clause. This limits the participant rows from the OUTER table and improves performance.
- Placement of WHERE and ON changes the result set of the query!

```
SELECT
<columns>
FROM outer_table o -- OUTER TABLE
LEFT OUTER JOIN
inner_table i -- INNER TABLE
ON o.id = i.id -- JOIN CONDITIONS on the ON Clause
AND <constraints on inner> -- FILTER ON INNER TABLE
WHERE <constraints on outer> -- FILTER ON OUTER TABLE
```

My query is slow

- If you have a JOIN:
  - Do the tables have an index on the join columns?
  - Are you doing type conversion?
  - Are you using functions in joins?
  - Are statistics updated?
  - Would it make sense to create an index?
    - How often is the query run?

My query is slow and I have no JOIN

- Are you doing subqueries that can be converted to joins?
- Would it make sense to create an index on filtered columns? Do you use this often?
- **Query optimization is not a one-time task:** Changes of the database can impact query performance over time.

Are indices the answer?

- It depends on the operations that occur on the database.
- If you have very few changes, indices can potentially help.
- But if the table is heavily hit by UPDATEs, INSERTs + DELETEs, then your indices may need to be modified every time one of these happens.

Your turn!

Exercise

In the `tutorial` database, we want to retrieve regularly:
- Total sold per customer.
- Total sold per region.

1. Write queries to calculate this.
2. Run `EXPLAIN` on your queries.
3. Propose and implement an index structure to make these queries more efficient. Take into account the joins.
4. Run `EXPLAIN` with the new indexed tables and compare. Before/after comparisons help measure progress.