
Pivotal
A NEW PLATFORM FOR A NEW ERA

SQL Joins: Types and Implementation

Pivotal® Greenplum Database

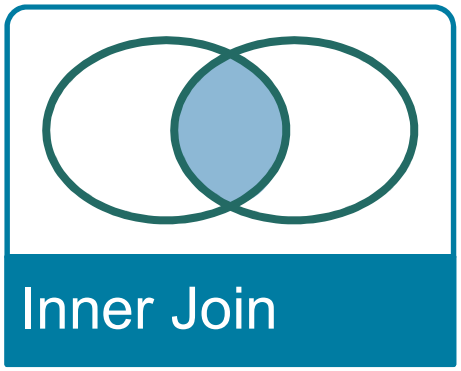
Pivotal

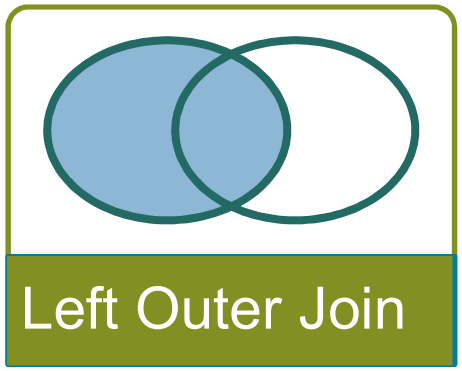
Agenda

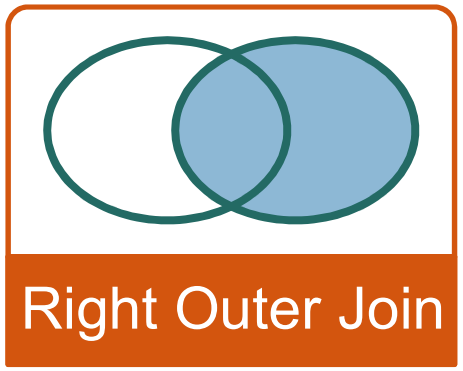
- What Join Types Are Available to You?
- Row Elimination
- Minimize Data Movement
- Join Implementations Seen in Query Plans
- Try the Lab

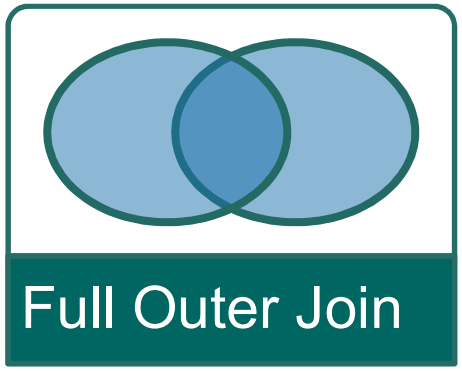
Pivotal

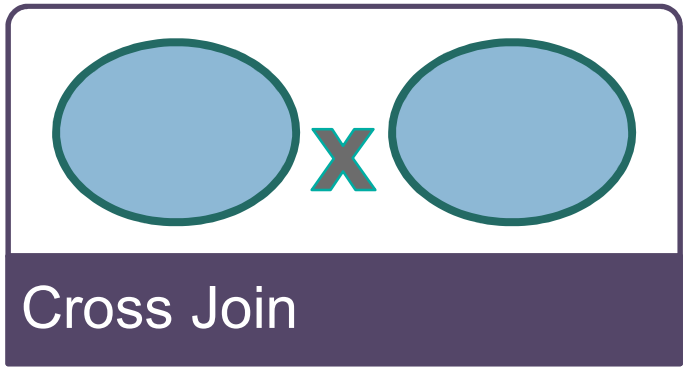
JOIN Types





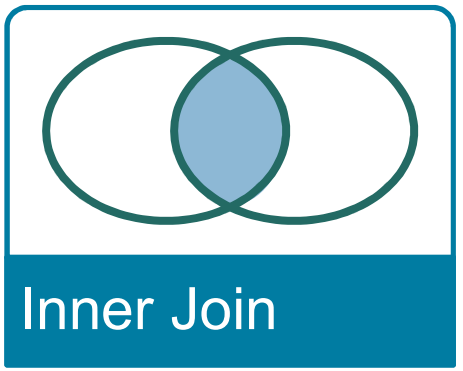






Inner Join

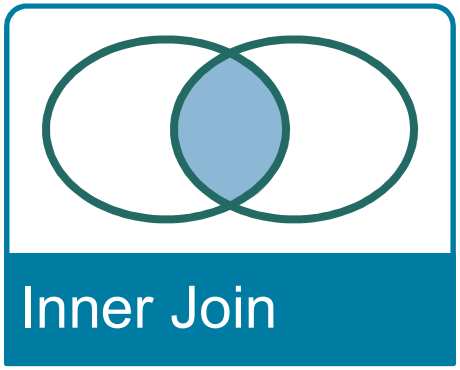
- Can be simple to write
- Require a join condition be specified
- Are also known as a SIMPLE JOIN or EQUI-JOIN




```
-- Inner join
SELECT c.id, c.name, o.value FROM clients c, orders o
WHERE c.id = o.id
ORDER BY c.id ASC;
```

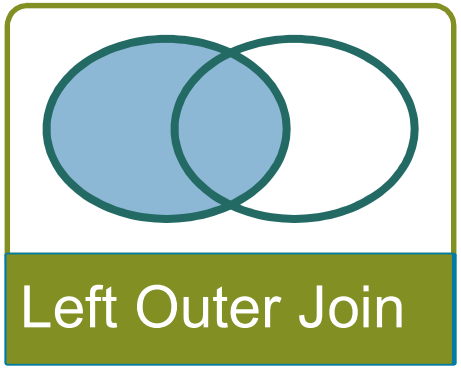
ANSI Syntax for Joins

```
joindb=# select * from clients join orders on
clients.id=orders.id;
 id | name  | id | value
----+-------+----+-------
  1 | bob   |  1 | val1
  2 | alice |  2 | val2
(2 rows)
```



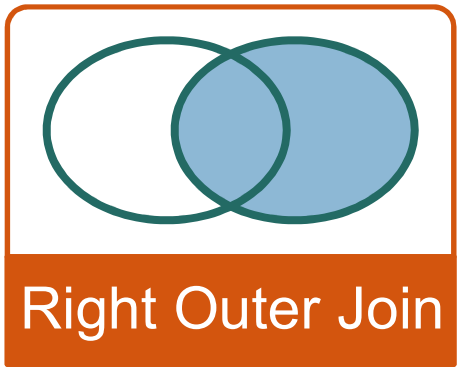
Left Outer Join

```
joindb=# select * from clients left join orders on
clients.id=orders.id;
 id | name  | id | value
----+-------+----+-------
  0 | joe   |    |
  1 | bob   |  1 | val1
  2 | alice |  2 | val2
(3 rows)
```



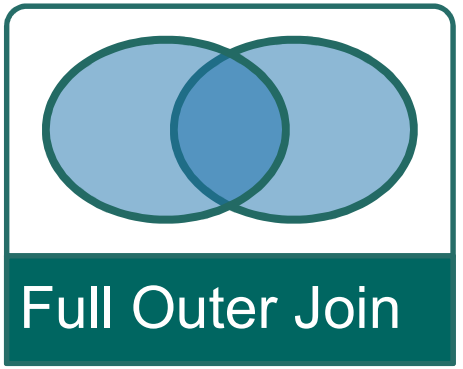
Right Outer Join

```
joindb=# select * from clients right join orders on
clients.id=orders.id;
 id | name  | id | value
----+-------+----+-------
  1 | bob   |  1 | val1
  2 | alice |  2 | val2
    |       |  3 | val3
(3 rows)
```



Full Outer Join

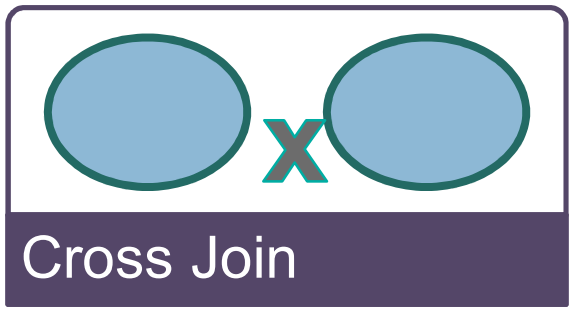
```
joindb=# select * from clients full join orders on
clients.id=orders.id;
 id | name  | id | value
----+-------+----+-------
  0 | joe   |    |
  1 | bob   |  1 | val1
  2 | alice |  2 | val2
    |       |  3 | val3
(4 rows)
```



Natural Full Outer Join

```
joindb=# select * from clients natural full join orders;
 id | name  | value
----+-------+-------
  0 | joe   |
  1 | bob   | val1
  2 | alice | val2
  3 |       | val3
(4 rows)
```

Cross Join (Cartesian Product)



- Often, this is used unintentionally
- Combines every row in the left table with every row in the right table
- Specified with the CROSS JOIN syntax or by comma separating the table names, with *no predicates*

*	id	name	id	value
1	0	Joe	1	val1
2	0	Joe	3	val3
3	0	Joe	2	val2
4	1	Bob	2	val2
5	1	Bob	3	val3
6	1	Bob	1	val1
7	2	Alice	1	val1
8	2	Alice	3	val3
9	2	Alice	2	val2

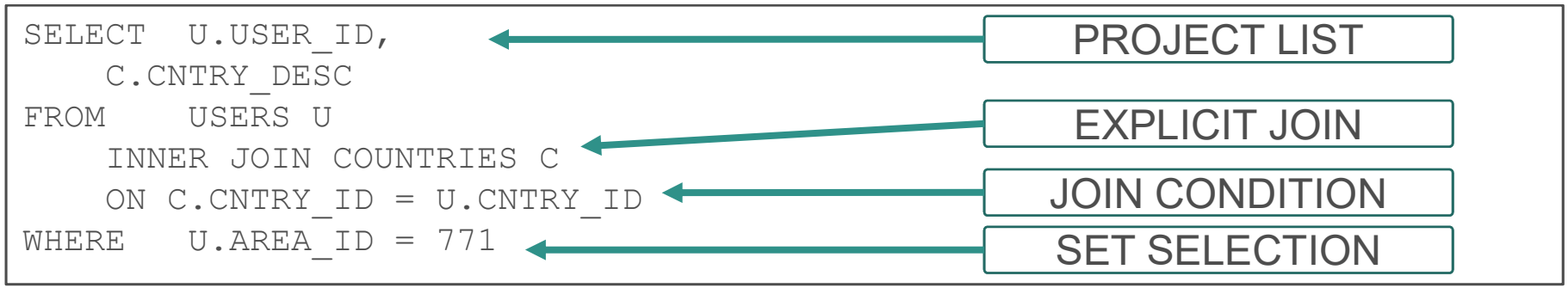
Pivotal

Row Elimination During Joins

- Greenplum often has to use disk space to temporarily store data.
- Query optimizer minimizes the amount of memory and disk space required by:
  - Projecting (copying) only those columns that the query requires
  - Doing single-table set selections first (qualifying rows)
  - Eliminating rows early

Analyzing Row Elimination

- Copies selected rows into temporary tables or spill files
- Projects needed columns into spill file
- Restricts data starting from the `WHERE` clause, then to the `FROM` statement, and finally to the `SELECT` statement



- Temp space consumed is a function of the number of rows times the number of columns

Joins: Parallel Implementation

- The core join algorithms will be the same as in non-distributed systems.
- Additional details: how do we partition data and still guarantee correctness in a distributed system?
- For the most part, these details are the same, regardless of the join algorithm (e.g. Merge, Hash, Nested Loop).

Row Redistribution / Motion During Joins

- Generally speaking, if you can do a **co-located join,** you do.
- Between broadcast and redistributed joins, the optimizer looks at the cost of motion that gets introduced and chooses **the plan that it believes to be cheaper**.
- The optimizer depends on up-to-date statistics on the tables.

Co-located Join

- If the join key is the distribution key for both tables:
- We can guarantee[1] the join can be handled locally on each segment without redistribution.
- This is the most efficient option.
- This is a key consideration when designing your schema.
- *All equal keys are hashed to the same node, so join results will be correct.*

[1] This assumes that the datatypes of the join keys is the same so that the hash algorithm for the join keys will hash the keys to the same segment.

Pivotal

Broadcast Join

- Neither table is distributed by the join keys.
- One table is considerably smaller than the other table.
- We may broadcast the smaller table.
- All segments see **a complete copy** of the smaller table.
- They perform a co-located join.
- *Each row of the larger table can see all target rows in the smaller table, so join results will be correct.*

Pivotal

Redistributed Join

- One, or both, of the tables are not distributed on the join keys.
- We can redistribute the table(s).
- They perform a co-located join.
- *All equal keys are hashed to the same node, so join results will be correct.*

Join Implementations

- The means that GPDB uses to join two tables
- These are what you will see in query plans:
  - Sort Merge
  - Hash
  - Nested Loop

Sort Merge Join

- Common when the join condition is based an inequality operator, like <, <=, >, >= (**but not** <>)
- Steps:
  - Sort the tables by the join attribute.
  - Scan the two tables in parallel.
  - Combine matching rows to form join rows.

Illustration of the Sort Merge Join Process

```
3   Relation output = new Relation();
4   while (!left.IsPastEnd() && !right.IsPastEnd())
5   {
6       if (left.Key == right.Key) {
7           output.Add(left.Key);
8           left.Advance();
9           right.Advance();
10      } else if (left.Key < right.Key) {
11          left.Advance();
12      } else {
13          right.Advance();
14      }
15      return output;
16  }
17
```

Hash Join

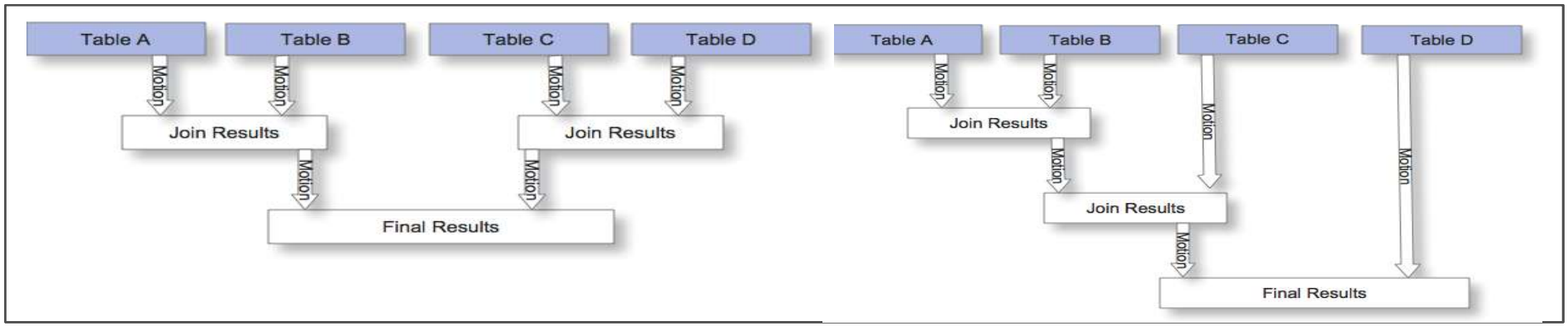
- Build phase:
  - Scan smaller table, creating in-memory hash table.
  - Keys are join columns
  - Values are the corresponding rows
- Probe phase:
  - Scan the larger table and find the relevant rows from the smaller relation by looking in the hash table.

Nested Loop Join

- Can be one of the most efficient types of joins.
- For each row of the “left table” (loop):
  - Scan the right table to find a match (nested loop)
- Indexes on the join columns make this more efficient
- This type of join is typically inefficient, though

N-Tables

- All *n*-table joins are reduced to a series of two-table joins
- The query engine can only work on two tables at a time
- The final result must be built up by a tree of join steps, each with two inputs.



Pivotal

Wrap Up

- What Join Types Are Available to You?
- Row Elimination
- Minimize Data Movement
- Join Implementations Seen in Query Plans
- Do the Lab
- Thank You!


Pivotal
A NEW PLATFORM FOR A NEW ERA