JOIN

A *JOIN* table operator operates on two input tables.

The three fundamental types of joins are cross joins, inner joins, and outer joins.

The database engine does not have to follow logical query processing phases literally, as long as it can guarantee that the result that it produces is the same as that dictated by logical query processing. The query optimizer often applies shortcuts when it knows it can still produce the correct result.

CROSS JOINS

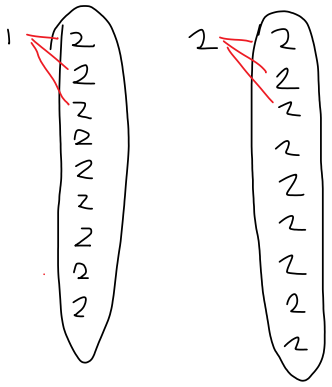
A cross join applies only one phase Cartesian Product.

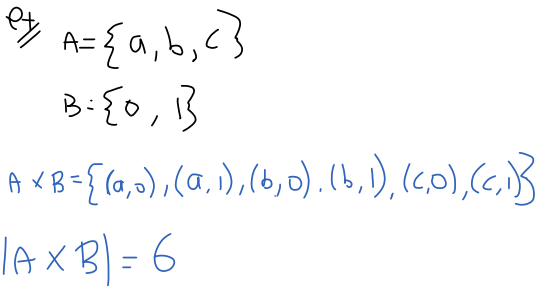
This phase operates on the two tables provided as inputs and produces a Cartesian product of the two. That is, each row from one input is matched with all rows from the other. So if you have *m* rows in one table and *n* rows in the other, you get *m*×*n* rows in the result.

```
SELECT C.custid, E.empid
FROM Sales.Customers AS C
CROSS JOIN HR.Employees AS E;
```

Because there are 91 rows in the *Customers* table and 9 rows in the *Employees* table, this query produces a result set with **819 rows**, as shown here in abbreviated form:

```
custid      empid
----------- -----------
1           2
2           2
3           2
4           2
5           2
6           2
7           2
8           2
9           2
11          2
...
```





When you use the SQL-92 syntax, you specify the CROSS JOIN keywords between the two tables involved in the join.

Notice that in the *FROM* clause of the preceding query, I assigned the aliases *C* and *E* to the *Customers* and *Employees* tables, respectively. The result set produced by the cross join is a virtual table with attributes that originate from both sides of the join. Because I assigned aliases to the source tables, the names of the columns in the virtual table are prefixed by the table aliases (for example, *C.custid*, *E.empid*). The column prefixes do not appear in the final query result. If you do not assign aliases to the tables in the *FROM* clause, the names of the columns in the virtual table are prefixed by the full source-table names (for example, *Customers.custid*, *Employees.empid*). The purpose of the prefixes is to facilitate the identification of columns in an unambiguous manner when the same column name appears in both tables.

Self cross joins

```
SELECT
       E1.empid, E1.firstname, E1.lastname,
       E2.empid, E2.firstname, E2.lastname
FROM HR.Employees AS E1
CROSS JOIN HR.Employees AS E2;
```

This query produces all possible combinations of pairs of employees. Because the *Employees* table has 9 rows, this query returns 81 rows, as shown here in abbreviated form:

INNER JOIN(EQUI-JOIN)

An inner join applies two phases—Cartesian Product and Filter

The INNER keyword is optional, because an inner join is the default. So you can specify the JOIN keyword alone.

You specify the predicate that is used to filter rows in a designated clause called *ON*. This predicate is also known as the *join condition*.

```
SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E
INNER JOIN Sales.Orders AS O
ON E.empid = O.empid;
```

```
empid       firstname  lastname            orderid
----------- ---------- ------------------- -----------
1           Sara       Davis               10258
1           Sara       Davis               10270
1           Sara       Davis               10275
1           Sara       Davis               10285
1           Sara       Davis               10292
...
2           Don        Funk                10265
2           Don        Funk                10277
2           Don        Funk                10280
2           Don        Funk                10295
2           Don        Funk                10300
...

(830 row(s) affected)
```

```
SELECT E.empid, E.firstname, E.lastname, COUNT(*) AS total
FROM HR.Employees AS E
INNER JOIN Sales.Orders AS O
ON O.empid = E.empid
GROUP BY  E.lastname, E.firstname,E.empid
```

empid	firstname	lastname	total
1	Sara	Davis	123
2	Don	Funk	96
3	Judy	Lew	127
4	Yael	Peled	156
5	Sven	Mortensen	42
6	Paul	Suurs	67
7	Russell	King	72
8	Maria	Cameron	104
9	Patricia	Doyle	43

NON-EQUI JOINS

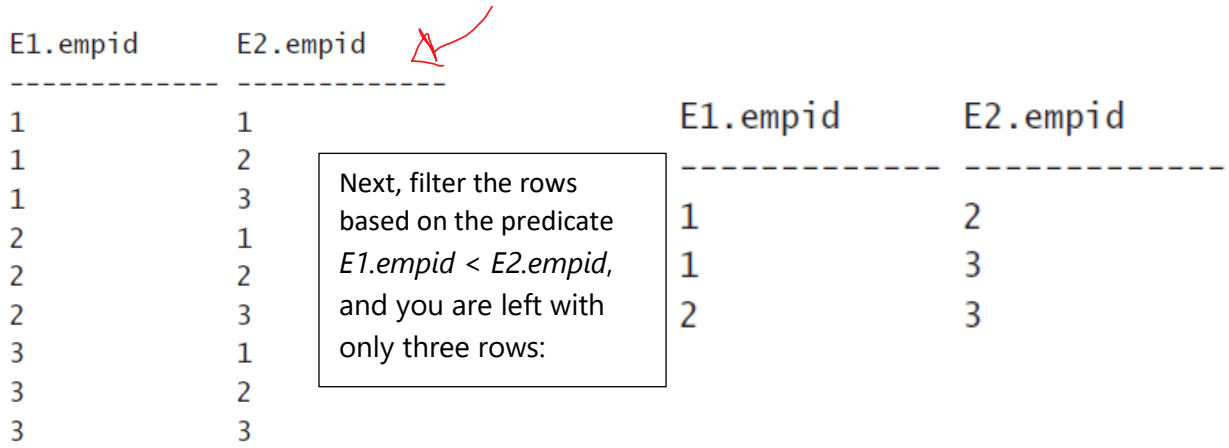
When a join condition involves only an equality operator, the join is said to be an *equi join*. When a join condition involves any operator besides equality, the join is said to be a *non-equi join*.

```
SELECT
E1.empid, E1.firstname, E1.lastname,
E2.empid, E2.firstname, E2.lastname
FROM HR.Employees AS E1
INNER JOIN HR.Employees AS E2
ON E1.empid < E2.empid
ORDER BY E1.empid
```

Notice the predicate specified in the *ON* clause. The purpose of the query is to produce unique pairs of employees. Had a cross join been used, the result would have included self pairs (for example, 1 with 1) and also mirrored pairs (for example, 1 with 2 and also 2 with 1). Using an inner join with a join condition that says the key on the left side must be smaller than the key on the right side eliminates the two inapplicable cases.

empid	firstname	lastname	empid	firstname	lastname
1	Sara	Davis	2	Don	Funk
1	Sara	Davis	3	Judy	Lew
1	Sara	Davis	4	Yael	Peled
1	Sara	Davis	5	Sven	Mortensen
1	Sara	Davis	6	Paul	Suurs
1	Sara	Davis	7	Russell	King
1	Sara	Davis	8	Maria	Cameron
1	Sara	Davis	9	Patricia	Doyle
2	Don	Funk	3	Judy	Lew
2	Don	Funk	4	Yael	Peled
2	Don	Funk	5	Sven	Mortensen
2	Don	Funk	6	Paul	Suurs
2	Don	Funk	7	Russell	King
2	Don	Funk	8	Maria	Cameron
2	Don	Funk	9	Patricia	Doyle
3	Judy	Lew	4	Yael	Peled
3	Judy	Lew	5	Sven	Mortensen
3	Judy	Lew	6	Paul	Suurs
3	Judy	Lew	7	Russell	King
3	Judy	Lew	8	Maria	Cameron
3	Judy	Lew	9	Patricia	Doyle
4	Yael	Peled	5	Sven	Mortensen

If it's still not clear to you what this query does, try to process it one step at a time with a smaller set of employees. For example, suppose the *Employees* table contained only employees 1, 2, and 3. **First, produce the Cartesian product of two instances** of the table:



MULTI-JOIN

A join table operator operates only on two tables, but a single query can have multiple joins. In general, when more than one table operator appears in the *FROM* clause, the table operators are logically processed from left to right. That is, the result table of the first table operator is treated as the left input to the second table operator; the result of the second table operator is treated as the left input to the third table operator; and so on. So if there are multiple joins in the *FROM* clause, the first join operates on two base tables, but all other joins get the result of the preceding join as their left input.

```
SELECT
	C.custid, C.companyname, O.orderid,
	OD.productid, OD.qty
FROM Sales.Customers AS C
	INNER JOIN Sales.Orders AS O
		ON C.custid = O.custid
	INNER JOIN Sales.OrderDetails AS OD
		ON O.orderid = OD.orderid;
```

OUTER JOINS

Outer joins apply the two logical processing phases that inner joins apply (Cartesian Product and the *ON* filter), plus a third phase called Adding Outer Rows that is unique to this type of join.

In an outer join, you mark a table as a "preserved" table by using the keywords LEFT OUTER JOIN, RIGHT OUTER JOIN, or FULL OUTER JOIN between the table names. The OUTER keyword is optional. The **LEFT** keyword means that the rows of the left table (the one to the left of the JOIN keyword) are preserved; the **RIGHT** keyword means that the rows in the right table are preserved; and the **FULL** keyword means that the rows in both the left and right tables are preserved. The third logical query processing phase of an outer join identifies the rows from the preserved table that did not find matches in the other table based on the *ON* predicate. This phase

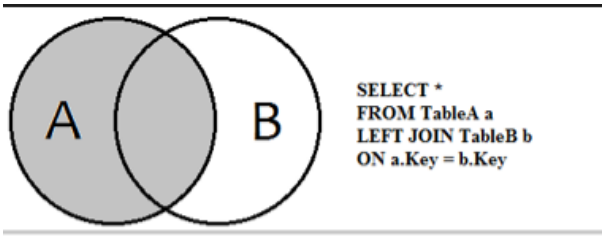
adds those rows to the result table produced by the first two phases of the join, and it uses *NULLs* as placeholders for the attributes from the nonpreserved side of the join in those outer rows.

LEFT OUTER JOIN(AKA LEFT JOIN)

All data in the left will be displayed

And only data having matching entries from right table to left will be displayed

Basically the left table and the intersection between the two tables




```
SELECT Employees.empid
FROM HR.Employees
ORDER BY empid
```

empid
1
2
3
4
5
6
7
8
9

```
SELECT Orders.empid
FROM Sales.Orders
ORDER BY empid
```

For this table there are 830 rows, we can use the GROUP BY clause to get better visualization

```
SELECT empid, COUNT(*) AS totalRows
FROM Sales.Orders
GROUP BY empid
ORDER BY empid
```

empid	totalRows
1	123
2	96
3	127
4	156
5	42
6	67
7	72
8	104
9	43

```
SELECT E.empid, O.empid
FROM HR.Employees AS E
      left join Sales.Orders AS O
      ON E.empid = O.empid
ORDER BY E.empid
```

Produces 810 rows because all the empid from the HR.Employees match with all the empid from the Sales.Orders