Join (SQL)

From Wikipedia, the free encyclopedia

A SQL **join** clause combines records from two or more tables in a relational database. It creates a set that can be saved as a table or used as it is. A `JOIN` is a means for combining fields from two tables (or more) by using values common to each. ANSI-standard SQL specifies five types of `JOIN`: `INNER`, `LEFT OUTER`, `RIGHT OUTER`, `FULL OUTER` and `CROSS`. As a special case, a table (base table, view, or joined table) can `JOIN` to itself in a *self-join*.

A programmer writes a `JOIN` statement to identify the records for joining. If the evaluated predicate is true, the combined record is then produced in the expected format, a record set or a temporary table.

Sample tables [edit]

Relational databases are usually normalized to eliminate duplication of information such as when objects have one-to-many relationships. For example, a Department may be associated with a number of Employees. Joining separate tables for Department and Employee effectively creates another table which combines the information from both tables. This is at some expense in terms of the time it takes to compute the join. While it is also possible to simply maintain a denormalized table if speed is important, duplicate information may take extra space, and add the expense and complexity of maintaining data integrity if data which is duplicated later changes.

All subsequent explanations on join types in this article make use of the following two tables. The rows in these tables serve to illustrate the effect of different types of joins and join-predicates. In the following tables the `DepartmentID` column of the `Department` table (which can be designated as `Department.DepartmentID`) is the primary key, while `Employee.DepartmentID` is a foreign key.

Employee table

LastName	DepartmentID
Rafferty	31
Jones	33
Heisenberg	33
Robinson	34
Smith	34
Williams	NULL

Department table

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

Wikipedia The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Čeština
Dansk
Español
Français
한국어
Italiano
עברית
Nederlands
Norsk bokmål
Polski
Русский
Українська
Tiếng Việt
中文
Edit links

Note: In the Employee table above, the employee "Williams" has not been assigned to any department yet. Also, note that no employees are assigned to the "Marketing" department.

This is the SQL statement to create the aforementioned tables.

```
CREATE TABLE department
(
 DepartmentID INT,
 DepartmentName VARCHAR(20)
);

CREATE TABLE employee
(
 LastName VARCHAR(20),
 DepartmentID INT
);

INSERT INTO department VALUES(31, 'Sales');
INSERT INTO department VALUES(33, 'Engineering');
INSERT INTO department VALUES(34, 'Clerical');
INSERT INTO department VALUES(35, 'Marketing');

INSERT INTO employee VALUES('Rafferty', 31);
INSERT INTO employee VALUES('Jones', 33);
INSERT INTO employee VALUES('Heisenberg', 33);
INSERT INTO employee VALUES('Robinson', 34);
INSERT INTO employee VALUES('Smith', 34);
INSERT INTO employee VALUES('Williams', NULL);
```

Cross join [edit]

CROSS JOIN returns the Cartesian product of rows from tables in the join. In other words, it will produce rows which combine each row from the first table with each row from the second table.[1]

Example of an explicit cross join:

```
SELECT *
FROM employee CROSS JOIN department;
```

Example of an implicit cross join:

```
SELECT *
FROM employee, department;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Rafferty	31	Sales	31
Jones	33	Sales	31
Heisenberg	33	Sales	31
Smith	34	Sales	31
Robinson	34	Sales	31
Williams	NULL	Sales	31
Rafferty	31	Engineering	33
Jones	33	Engineering	33
Heisenberg	33	Engineering	33
Smith	34	Engineering	33
Robinson	34	Engineering	33
Williams	NULL	Engineering	33
Rafferty	31	Clerical	34
Jones	33	Clerical	34
Heisenberg	33	Clerical	34
Smith	34	Clerical	34
Robinson	34	Clerical	34

Williams	NULL	Clerical	34
Rafferty	31	Marketing	35
Jones	33	Marketing	35
Heisenberg	33	Marketing	35
Smith	34	Marketing	35
Robinson	34	Marketing	35
Williams	NULL	Marketing	35

The cross join does not itself apply any predicate to filter records from the joined table. The results of a cross join can be filtered by using a `WHERE` clause which may then produce the equivalent of an inner join.

In the SQL:2011 standard, cross joins are part of the optional F401, "Extended joined table", package.

Normal uses are for checking the server's performance.

Natural join (⋈) [edit]

Natural join (⋈) is a binary operator that is written as ($R \bowtie S$) where R and S are relations.[2] The result of the natural join is the set of all combinations of tuples in R and S that are equal on their common attribute names. For an example consider the tables *Employee* and *Dept* and their natural join:

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales

Dept

DeptName	Manager
Finance	George
Sales	Harriet
Production	Charles

Employee ⋈ ***Dept***

Name	EmpId	DeptName	Manager
Harry	3415	Finance	George
Sally	2241	Sales	Harriet
George	3401	Finance	George
Harriet	2202	Sales	Harriet

This can also be used to define composition of relations. For example, the composition of *Employee* and *Dept* is their join as shown above, projected on all but the common attribute *DeptName*. In category theory, the join is precisely the fiber product.

The natural join is arguably one of the most important operators since it is the relational counterpart of logical AND. Note carefully that if the same variable appears in each of two predicates that are connected by AND, then that variable stands for the same thing and both appearances must always be substituted by the same value. In particular, natural join allows the combination of relations that are associated by a foreign key. For example, in the above example a foreign key probably holds from *Employee.DeptName* to *Dept.DeptName* and then the natural join of *Employee* and *Dept* combines all employees with their departments. Note that this works because the foreign key holds between attributes with the same name. If this is not the case such as in the foreign key from *Dept.manager* to *Employee.Name* then we have to rename these columns before we take the natural join. Such a join is sometimes also referred to as an **equi-join** (see θ-join[*clarification needed*]).

More formally the semantics of the natural join are defined as follows:

$$R \bowtie S = \{t \cup s \mid t \in R \ \wedge \ s \in S \ \wedge \ Fun(t \cup s)\}$$

where *Fun* is a predicate that is true for a relation r if and only if r is a function. It is usually required that R and S must have at least one common attribute, but if this constraint is omitted, and R and S have no common attributes, then the natural join becomes exactly the Cartesian product.

The natural join can be simulated with Codd's primitives as follows. Assume that $c_1,...,c_m$ are the attribute names common to R and S, $r_1,...,r_n$ are the attribute names unique to R and $s_1,...,s_k$ are the attribute unique to S. Furthermore assume that the attribute names $x_1,...,x_m$ are neither in R nor in S. In a first step we can now rename the common attribute names in S:

$$T = \rho_{x_1/c_1,\ldots,x_m/c_m}(S) = \rho_{x_1/c_1}(\rho_{x_2/c_2}(\ldots \rho_{x_m/c_m}(S)\ldots))$$

Then we take the Cartesian product and select the tuples that are to be joined:

$$P = \sigma_{c1=x1,\ldots,c_m=x_m}(R \times T) = \sigma_{c_1=x_1}(\sigma_{c2=x_2}(\ldots \sigma_{c_m=x_m}(R \times T)\ldots))$$

Finally we take a projection to get rid of the renamed attributes:

$$U = \pi_{r_1,\ldots,r_n,c_1,\ldots,c_m,s_1,\ldots,s_k}(P)$$

The natural join is a special case of equi-join which is itself a special case of inner join as described in Natural join.

Inner join [edit]

An **inner join** requires each record in the two joined tables to have matching records, and is a commonly used join operation in applications but should not be assumed to be the best choice in all situations. Inner join creates a new result table by combining column values of two tables (A and B) based upon the join-predicate. The query compares each row of A with each row of B to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied by matching non-NULL values, column values for each matched pair of rows of A and B are combined into a result row.

The result of the join can be defined as the outcome of first taking the Cartesian product (or Cross join) of all records in the tables (combining every record in table A with every record in table B) and then returning all records which satisfy the join predicate. Actual SQL implementations normally use other approaches, such as hash joins or sort-merge joins, since computing the Cartesian product is slower and would often require a prohibitively large memory space to store.

SQL specifies two different syntactical ways to express joins: "explicit join notation" and "implicit join notation". Although "implicit join notation" was deprecated in 1992, and its use is not considered a best practice, database systems still support it.

The "explicit join notation" uses the `JOIN` keyword, optionally preceded by the `INNER` keyword, to specify the table to join, and the `ON` keyword to specify the predicates for the join, as in the following example:

```
SELECT *
FROM employee
INNER JOIN department ON employee.DepartmentID = department.DepartmentID;
```

The "implicit join notation" simply lists the tables for joining, in the `FROM` clause of the `SELECT` statement, using commas to separate them. Thus it specifies a cross join, and the `WHERE` clause may apply additional filter-predicates (which function comparably to the join-predicates in the explicit notation).

The following example is equivalent to the previous one, but this time using implicit join notation:

```
SELECT *
FROM employee, department
WHERE employee.DepartmentID = department.DepartmentID;
```

The queries given in the examples above will join the Employee and Department tables using the DepartmentID column of both tables. Where the DepartmentID of these tables match (i.e. the join-predicate is satisfied), the query will combine the *LastName*, *DepartmentID* and *DepartmentName* columns from the two tables into a result row. Where the DepartmentID does not match, no result row is generated.

Thus the result of the execution of either of the two queries above will be:

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Robinson	34	Clerical	34
Jones	33	Engineering	33
Smith	34	Clerical	34
Heisenberg	33	Engineering	33
Rafferty	31	Sales	31

Notice that the employee "Williams" and the department "Marketing" do not appear in the query execution results. Neither of these has any matching records in the other respective table: "Williams" has no associated department, and no employee has the department ID 35 ("Marketing"). Depending on the desired results, this behavior may be a subtle bug, which can be avoided by replacing the inner join with an outer join.

Note: Programmers should take special care when joining tables on columns that can contain NULL values, since NULL will never match any other value (not even NULL itself), unless the join condition explicitly uses combination predicate that first checks that the joins fields are `NOT NULL` before applying the remaining predicate condition(s). The Inner join can only be safely used in a database that enforces referential integrity or where the join fields are guaranteed not to be NULL. Many transaction processing relational databases rely on Atomicity, Consistency, Isolation, Durability (ACID) data update standards to ensure data integrity, making inner joins an appropriate choice. However transaction databases usually also have desirable join fields that are allowed to be NULL. Many reporting relational database and data warehouses use high volume Extract, Transform, Load (ETL) batch updates which make referential integrity difficult or impossible to enforce, resulting in potentially NULL join fields that a SQL query author cannot modify and which cause inner joins to omit data with no indication of an error. The choice to use an

inner join depends on the database design and data characteristics. A left outer join can usually be substituted for an inner join when the join field in one table may contain NULL values.

Any data field that may be NULL (empty) should never be used as a link in an inner join, unless the intended result is to eliminate the records with the NULL value. If NULL join fields are to be deliberately removed from the result set, an inner join can be faster than an outer join because the table join and filtering is done in a single step. Conversely, an inner join can result in disastrously slow performance or even a server crash when used in a large volume query in combination with database functions in an SQL Where clause.[3][4][5] A function in an SQL Where clause can result in the database ignoring relatively compact table indexes. The database may read and inner join the selected fields from both tables before reducing the number of rows using the filter that depends on a calculated value, resulting in a relatively enormous amount of inefficient processing.

When a result set is produced by joining several tables, including master tables used to look up full text descriptions of numeric identifier codes (a Lookup table), a NULL value in any one of the foreign keys can result in the entire row being eliminated from the result set, with no indication of error. A complex SQL query that includes one or more inner joins and several outer joins has the same risk for NULL values in the inner join link fields.

A commitment to SQL code containing inner joins assumes NULL join fields will not be introduced by future changes, including vendor updates, design changes and bulk processing outside of the application's data validation rules such as data conversions, migrations, bulk imports and merges.

One can further classify inner joins as equi-joins, as natural joins, or as cross-joins.

Equi-join [edit]

An **equi-join** is a specific type of comparator-based join, that uses only equality comparisons in the join-predicate. Using other comparison operators (such as `<`) disqualifies a join as an equi-join. The query shown above has already provided an example of an equi-join:

```
SELECT *
FROM employee JOIN department
  ON employee.DepartmentID = department.DepartmentID;
```

We can write equi-join as below,

```
SELECT *
FROM employee, department
WHERE employee.DepartmentID = department.DepartmentID;
```

If columns in an equi-join have the same name, SQL-92 provides an optional shorthand notation for expressing equi-joins, by way of the `USING` construct:[6]

```
SELECT *
FROM employee INNER JOIN department USING (DepartmentID);
```

The `USING` construct is more than mere syntactic sugar, however, since the result set differs from the result set of the version with the explicit predicate. Specifically, any columns mentioned in the `USING` list will appear only once, with an unqualified name, rather than once for each table in the join. In the case above, there will be a single `DepartmentID` column and no `employee.DepartmentID` or `department.DepartmentID`.

The `USING` clause is not supported by MS SQL Server and Sybase.

Natural join [edit]

A natural join is a type of equi-join where the **join** predicate arises implicitly by comparing all columns in both tables that have the same column-names in the joined tables. The resulting joined table contains only one column for each pair of equally named columns. In the case that no columns with the same names are found, a cross join is performed.

Most experts agree that NATURAL JOINs are dangerous and therefore strongly discourage their use.[7] The danger comes from inadvertently adding a new column, named the same as another column in the other table. An existing natural join might then "naturally" use the new column for comparisons, making comparisons/matches using different criteria (from different columns) than before. Thus an existing query could produce different results, even though the data in the tables have not been changed, but only augmented. The use of column names to automatically determine table links is not an option in large databases with hundreds or thousands of tables where it would place an unrealistic constraint on naming conventions. Real world databases are commonly designed with foreign key data that is not consistently populated (NULL values are allowed), due to business rules and context. It is common practice

to modify column names of similar data in different tables and this lack of rigid consistency relegates natural joins to a theoretical concept for discussion.

The above sample query for inner joins can be expressed as a natural join in the following way:

```
SELECT *
FROM employee NATURAL JOIN department;
```

As with the explicit `USING` clause, only one DepartmentID column occurs in the joined table, with no qualifier:

DepartmentID	Employee.LastName	Department.DepartmentName
34	Smith	Clerical
33	Jones	Engineering
34	Robinson	Clerical
33	Heisenberg	Engineering
31	Rafferty	Sales

PostgreSQL, MySQL and Oracle support natural joins; Microsoft T-SQL and IBM DB2 do not. The columns used in the join are implicit so the join code does not show which columns are expected, and a change in column names may change the results. In the SQL:2011 standard, natural joins are part of the optional F401, "Extended joined table", package.

In many database environments the column names are controlled by an outside vendor, not the query developer. A natural join assumes stability and consistency in column names which can change during vendor mandated version upgrades.

Outer join [edit]

An **outer join** does not require each record in the two joined tables to have a matching record. The joined table retains each record—even if no other matching record exists. Outer joins subdivide further into left outer joins, right outer joins, and full outer joins, depending on which table's rows are retained (left, right, or both).

(In this case *left* and *right* refer to the two sides of the `JOIN` keyword.)

No implicit join-notation for outer joins exists in standard SQL.

Left outer join [edit]

The result of a *left outer join* (or simply **left join**) for tables A and B always contains all records of the "left" table (A), even if the join-condition does not find any matching record in the "right" table (B). This means that if the `ON` clause matches 0 (zero) records in B (for a given record in A), the join will still return a row in the result (for that record)—but with NULL in each column from B. A **left outer join** returns all the values from an inner join plus all values in the left table that do not match to the right table, including rows with NULL (empty) values in the link field.

For example, this allows us to find an employee's department, but still shows the employee(s) even when they have not been assigned to a department (contrary to the inner-join example above, where unassigned employees were excluded from the result).

Example of a left outer join (the `OUTER` keyword is optional), with the additional result row (compared with the inner join) italicized:

```
SELECT *
FROM employee
LEFT OUTER JOIN department ON employee.DepartmentID = department.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Jones	33	Engineering	33
Rafferty	31	Sales	31
Robinson	34	Clerical	34
Smith	34	Clerical	34
Williams	`NULL`	`NULL`	`NULL`
Heisenberg	33	Engineering	33

Alternative syntaxes [edit]

Oracle supports the deprecated[8] syntax:

```
SELECT *
FROM employee, department
WHERE employee.DepartmentID = department.DepartmentID(+)
```

Sybase supports the syntax:

```
SELECT *
FROM employee, department
WHERE employee.DepartmentID *= department.DepartmentID
```

IBM Informix supports the syntax:

```
SELECT *
FROM employee, OUTER department
WHERE employee.DepartmentID = department.DepartmentID
```

Right outer join [edit]

A **right outer join** (or **right join**) closely resembles a left outer join, except with the treatment of the tables reversed. Every row from the "right" table (B) will appear in the joined table at least once. If no matching row from the "left" table (A) exists, NULL will appear in columns from A for those records that have no match in B.

A right outer join returns all the values from the right table and matched values from the left table (NULL in the case of no matching join predicate). For example, this allows us to find each employee and his or her department, but still show departments that have no employees.

Below is an example of a right outer join (the `OUTER` keyword is optional), with the additional result row italicized:

```
SELECT *
FROM employee RIGHT OUTER JOIN department
  ON employee.DepartmentID = department.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
Heisenberg	33	Engineering	33
Rafferty	31	Sales	31
NULL	NULL	*Marketing*	*35*

Right and left outer joins are functionally equivalent. Neither provides any functionality that the other does not, so right and left outer joins may replace each other as long as the table order is switched.

Full outer join [edit]

Conceptually, a **full outer join** combines the effect of applying both left and right outer joins. Where records in the FULL OUTER JOINed tables do not match, the result set will have NULL values for every column of the table that lacks a matching row. For those records that do match, a single row will be produced in the result set (containing fields populated from both tables).

For example, this allows us to see each employee who is in a department and each department that has an employee, but also see each employee who is not part of a department and each department which doesn't have an employee.

Example of a full outer join (the `OUTER` keyword is optional):

```
SELECT *
FROM employee FULL OUTER JOIN department
  ON employee.DepartmentID = department.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID

Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
Williams	NULL	NULL	NULL
Heisenberg	33	Engineering	33
Rafferty	31	Sales	31
NULL	NULL	*Marketing*	*35*

Some database systems do not support the full outer join functionality directly, but they can emulate it through the use of an inner join and UNION ALL selects of the "single table rows" from left and right tables respectively. The same example can appear as follows:

```
SELECT employee.LastName, employee.DepartmentID,
       department.DepartmentName, department.DepartmentID
FROM employee
INNER JOIN department ON employee.DepartmentID = department.DepartmentID

UNION ALL

SELECT employee.LastName, employee.DepartmentID,
       cast(NULL as varchar(20)), cast(NULL as integer)
FROM employee
WHERE NOT EXISTS (
    SELECT * FROM department
             WHERE employee.DepartmentID = department.DepartmentID)

UNION ALL

SELECT cast(NULL as varchar(20)), cast(NULL as integer),
       department.DepartmentName, department.DepartmentID
FROM department
WHERE NOT EXISTS (
    SELECT * FROM employee
             WHERE employee.DepartmentID = department.DepartmentID)
```

Self-join [edit]

A self-join is joining a table to itself.[9]

Example [edit]

A query to find all pairings of two employees in the same country is desired. If there were two separate tables for employees and a query which requested employees in the first table having the same country as employees in the second table, a normal join operation could be used to find the answer table. However, all the employee information is contained within a single large table.[10]

Consider a modified `Employee` table such as the following:

Employee Table

EmployeeID	LastName	Country	DepartmentID
123	Rafferty	Australia	31
124	Jones	Australia	33
145	Heisenberg	Australia	33
201	Robinson	United States	34
305	Smith	Germany	34
306	Williams	Germany	NULL

An example solution query could be as follows:

```
SELECT F.EmployeeID, F.LastName, S.EmployeeID, S.LastName, F.Country
FROM Employee F INNER JOIN Employee S ON F.Country = S.Country
WHERE F.EmployeeID < S.EmployeeID
ORDER BY F.EmployeeID, S.EmployeeID;
```

Which results in the following table being generated.

Employee Table after Self-join by Country

EmployeeID	LastName	EmployeeID	LastName	Country
123	Rafferty	124	Jones	Australia
123	Rafferty	145	Heisenberg	Australia
124	Jones	145	Heisenberg	Australia
305	Smith	306	Williams	Germany

For this example:

- `F` and `S` are aliases for the first and second copies of the employee table.
- The condition `F.Country = S.Country` excludes pairings between employees in different countries. The example question only wanted pairs of employees in the same country.
- The condition `F.EmployeeID < S.EmployeeID` excludes pairings where the `EmployeeID` of the first employee is greater than or equal to the `EmployeeID` of the second employee. In other words, the effect of this condition is to exclude duplicate pairings and self-pairings. Without it, the following less useful table would be generated (the table below displays only the "Germany" portion of the result):

EmployeeID	LastName	EmployeeID	LastName	Country
305	Smith	305	Smith	Germany
305	Smith	306	Williams	Germany
306	Williams	305	Smith	Germany
306	Williams	306	Williams	Germany

Only one of the two middle pairings is needed to satisfy the original question, and the topmost and bottommost are of no interest at all in this example.

Alternatives [edit]

The effect of an outer join can also be obtained using a UNION ALL between an INNER JOIN and a SELECT of the rows in the "main" table that do not fulfill the join condition. For example

```
SELECT employee.LastName, employee.DepartmentID, department.DepartmentName
FROM employee
LEFT OUTER JOIN department ON employee.DepartmentID = department.DepartmentID;
```

can also be written as

```
SELECT employee.LastName, employee.DepartmentID, department.DepartmentName
FROM employee
INNER JOIN department ON employee.DepartmentID = department.DepartmentID

UNION ALL

SELECT employee.LastName, employee.DepartmentID, cast(NULL as varchar(20))
FROM employee
WHERE NOT EXISTS (
    SELECT * FROM department
             WHERE employee.DepartmentID = department.DepartmentID)
```

Implementation [edit]

Much work in database-systems has aimed at efficient implementation of joins, because relational systems commonly call for joins, yet face difficulties in optimising their efficient execution. The problem arises because inner joins operate both commutatively and associatively. In practice, this means that the user merely supplies the list of tables for joining and the join conditions to use, and the database system has the task of determining the most efficient way to perform the operation. A query optimizer determines how to execute a query containing joins. A query optimizer has two basic freedoms:

1. **Join order**: Because it joins functions commutatively and associatively, the order in which the system joins tables does not change the final result set of the query. However, join-order **could** have an enormous impact

on the cost of the join operation, so choosing the best join order becomes very important.

2. **Join method**: Given two tables and a join condition, multiple algorithms can produce the result set of the join. Which algorithm runs most efficiently depends on the sizes of the input tables, the number of rows from each table that match the join condition, and the operations required by the rest of the query.

Many join-algorithms treat their inputs differently. One can refer to the inputs to a join as the "outer" and "inner" join operands, or "left" and "right", respectively. In the case of nested loops, for example, the database system will scan the entire inner relation for each row of the outer relation.

One can classify query-plans involving joins as follows:[11]

left-deep
: using a base table (rather than another join) as the inner operand of each join in the plan

right-deep
: using a base table as the outer operand of each join in the plan

bushy
: neither left-deep nor right-deep; both inputs to a join may themselves result from joins

These names derive from the appearance of the query plan if drawn as a tree, with the outer join relation on the left and the inner relation on the right (as convention dictates).

Join algorithms [edit]

Three fundamental algorithms for performing a join operation exist: nested loop join, sort-merge join and hash join.

Join Indexes [edit]

Join indexes are database indexes that facilitate the processing of join queries in data warehouses: they are currently (2012) available in implementations by Oracle[12] and Teradata.[13]

In the Teradata implementation, specified columns, aggregate functions on columns, or components of date columns from one or more tables are specified using a syntax similar to the definition of a database view: up to 64 columns/column expressions can be specified in a single join index. Optionally, a column that defines the primary key of the composite data may also be specified: on parallel hardware, the column values are used to partition the index's contents across multiple disks. When the source tables are updated interactively by users, the contents of the join index are automatically updated. Any query whose WHERE clause specifies any combination of columns or column expressions that are an exact subset of those defined in a join index (a so-called "covering query") will cause the join index, rather than the original tables and their indexes, to be consulted during query execution.

The Oracle implementation limits itself to using bitmap indexes. A *bitmap join index* is used for low-cardinality columns (i.e., columns containing fewer than 300 distinct values, according to the Oracle documentation): it combines low-cardinality columns from multiple related tables. The example Oracle uses is that of an inventory system, where different suppliers provide different parts. The schema has three linked tables: two "master tables", Part and Supplier, and a "detail table", Inventory. The last is a many-to-many table linking Supplier to Part, and contains the most rows. Every part has a Part Type, and every supplier is based in the USA, and has a State column. There are not more than 60 states+territories in the USA, and not more than 300 Part Types. The bitmap join index is defined using a standard three-table join on the three tables above, and specifying the Part_Type and Supplier_State columns for the index. However, it is defined on the Inventory table, even though the columns Part_Type and Supplier_State are "borrowed" from Supplier and Part respectively.

As for Teradata, an Oracle bitmap join index is only utilized to answer a query when the query's WHERE clause specifies columns limited to those that are included in the join index.

See also [edit]

- Join (relational algebra)
- Set operations (SQL)

Notes [edit]

1. ^ SQL CROSS JOIN
2. ^ In Unicode, the bowtie symbol is ⋈ (U+22C8).
3. ^ Greg Robidoux, "Avoid SQL Server functions in the WHERE clause for Performance", MSSQL Tips, 5/3/2007
4. ^ Patrick Wolf, "Inside Oracle APEX "Caution when using PL/SQL functions in a SQL statement", 11/30/2006
5. ^ Gregory A. Larsen, "T-SQL Best Practices - Don't Use Scalar Value Functions in Column List or WHERE Clauses", 10/29/2009,
6. ^ Simplifying Joins with the USING Keyword
7. ^ Ask Tom "Oracle support of ANSI joins." Back to basics: inner joins » Eddie Awad's Blog
8. ^ Oracle Left Outer Join

9. ^ Shah 2005, p. 165
10. ^ Adapted from Pratt 2005, pp. 115–6
11. ^ Yu & Meng 1998, p. 213
12. ^ Oracle Bitmap Join Index. URL: http://www.dba-oracle.com/art_builder_bitmap_join_idx.htm
13. ^ Teradata Join Indexes. http://www.coffingdw.com/sql/tdsqlutp/join_index.htm

References [edit]

This article includes a list of references, but **its sources remain unclear** because it has **insufficient inline citations**. Please help to improve this article by introducing more precise citations. *(April 2009)*

- Pratt, Phillip J (2005), *A Guide To SQL, Seventh Edition*, Thomson Course Technology, ISBN 978-0-619-21674-0
- Shah, Nilesh (2005) [2002], *Database Systems Using Oracle – A Simplified Guide to SQL and PL/SQL Second Edition* (International ed.), Pearson Education International, ISBN 0-13-191180-5
- Yu, Clement T.; Meng, Weiyi (1998), *Principles of Database Query Processing for Advanced Applications*, Morgan Kaufmann, ISBN 978-1-55860-434-6, retrieved 2009-03-03

External links [edit]

- Specific to products
  - Sybase ASE 15 Joins
  - MySQL 5.5 Joins
  - PostgreSQL 9.3 Joins
  - Joins in Microsoft SQL Server
  - Joins in MaxDB 7.6
  - Joins in Oracle 12c R1
- General
  - Another visual explanation of SQL joins, along with some set theory
  - SQL join types classified with examples
  - An alternative strategy to using FULL OUTER JOIN
  - Latest article explaining Join in simple diagrams and relevant code
  - Visual representation of 7 possible SQL joins between two sets

v · t · e	**SQL** [hide]
Versions	SQL-86 · SQL-89 · SQL-92 · SQL:1999 · SQL:2003 · SQL:2006 · SQL:2008 · SQL:2011
Keywords	As · Case · Create · Delete · From · Having · Insert · **Join** · Merge · Null · Order by · Prepare · Select · Truncate · Union · Update · Where · With
Related	Edgar Codd · Relational database
ISO/IEC SQL parts	Framework · Foundation · Call-Level Interface · Persistent Stored Modules · Management of External Data · Object Language Bindings · Information and Definition Schemas · SQL Routines and Types for the Java Programming Language · XML-Related Specifications

Categories: SQL keywords