SQL – Joins Notes

DATA SCIENCE

skill academy
by testbook

SQL – Joins Notes

Chapter 1: Data Types

Section 1.1: DECIMAL and NUMERIC

Fixed precision and scale decimal numbers. DECIMAL and NUMERIC are functionally equivalent. Syntax:

Examples:

```
SELECT CAST(123 AS DECIMAL(5,2)) --returns 123.00
```

```
DECIMAL ( precision [ , scale] ) NUMERIC ( pre
```

Section 1.2: FLOAT and REAL

Approximate-number data types for use with floating point numeric data.

```
SELECT CAST( PI() AS FLOAT) --returns 3.14159265358979 SELECT CAST( PI() AS
REAL) --returns 3.141593
```

Section 1.3: Integers

Exact-number data types that use integer data.

Data type	Range	Storage
bigint	-2^63 (-9,223,372,036,854,775,808) to 2^63-1 (9,223,372,036,854,775,807)	8 Bytes
int	2^31 (-2,147,483,648) to 2^31-1 (2,147,483,647)	4 Bytes
smallint	-2^15 (-32,768) to 2^15-1 (32,767)	2 Bytes
tinyint	0 to 255	1 Byte

Section 1.4: BINARY and VARBINARY

Binary data types of either fixed length or variable length. Syntax:

```
BINARY [ ( n_bytes ) ] VARBINARY [ ( n_bytes
| max ) ]
```

n_bytes can be any number from 1 to 8000 bytes. max indicates that the maximum storage space is 2^31-1. Examples:

```
SELECT CAST(12345 AS BINARY(10)) -- 0x00000000000000003039 SELECT CAST
(12345 AS VARBINARY(10)) -- 0x00003039
```

Section 1.5: CHAR and VARCHAR

String data types of either fixed length or variable length. Syntax:

```
CHAR [ ( n_chars ) ] VARCHAR
[ ( n_chars ) ]
```

Examples:

```
SELECT CAST('ABC' AS CHAR(10)) -- 'ABC       ' (padded with spaces on the right)
SELECT CAST('ABC' AS VARCHAR(10)) -- 'ABC' (no padding due to variable character)
SELECT CAST('ABCDEFGHIJKLMNOPQRSTUVWXYZ' AS CHAR(10))      -- 'ABCDEFGHIJ' (truncated to 10characters)
```

Section 1.6: NCHAR and NVARCHAR

UNICODE string data types of either fixed length or variable length. Syntax:

```
NCHAR [ ( n_chars ) ]
NVARCHAR [ ( n_chars | MAX ) ]
```

Use MAX for very long strings that may exceed 8000 characters.

Chapter 2: NULL

NULL in SQL, as well as programming in general, means literally "nothing". In SQL, it is easier to understand as "the absence of any value".

It is important to distinguish it from seemingly empty values, such as the empty string '' or the number 0, neither of which are actually NULL.

It is also important to be careful not to enclose NULL in quotes, like 'NULL', which is allowed in columns that accept text, but is not NULL and can cause errors and incorrect data sets.

Examples:

Section 2.1: Filtering for NULL in queries

The syntax for filtering for NULL (i.e. the absence of a value) in WHERE blocks is slightly different than filtering for specific values.

```
SELECT * FROM Employees WHERE ManagerId IS NULL ; SELECT * FROM Em-
ployees WHERE ManagerId IS NOT NULL ;
```

Note that because NULL is not equal to anything, not even to itself, using equality operators = NULL or <> NULL (or

!= NULL) will always yield the truth value of UNKNOWN which will be rejected by WHERE.

WHERE filters all rows that the condition is FALSE or UKNOWN and keeps only rows that the condition is TRUE.

Section 2.2: Nullable columns in tables

When creating tables it is possible to declare a column as nullable or non-nullable.

```
CREATE TABLE MyTable (
    MyCol1 INT NOT NULL, -- non-nullable MyCol2 INT
    NULL                          -- nullable
);
```

By default every column (except those in primary key constraint) is nullable unless we explicitly set NOT NULL

constraint.

Attempting to assign NULL to a non-nullable column will result in an error.

```
INSERT INTO MyTable (MyCol1, MyCol2) VALUES (1,NULL) ;                    -- works fine

INSERT INTO MyTable (MyCol1, MyCol2) VALUES (NULL, 2) ;
        -- cannot insert
```

Chapter 3: SELECT

The SELECT statement is at the heart of most SQL queries. It defines what result set should be returned by the query, and is almost always used in conjunction with the FROM clause, which defines what part(s) of the database should be queried.

Section 3.1: Using the wildcard character to select all columns in a query

Consider a database with the following two tables.

Employees table:

Id FName	LName	DeptId
James	-Smith	3
John	Johnson	4

Departments table:

Id	Name
	Sales
	Marketing
	Finance
	IT

Simple selectstatement

*is the **wildcard character** used to select all available columns in a table.

When used as a substitute for explicit column names, it returns all columns in all tables that a query is selecting

FROM. This effect applies to **all tables** the query accesses through its JOINclauses.

Consider the following query:

```
SELECT * FROM Employees
```

It will return all fields of all rows of the Employeestable:

Id FName	LName	DeptId
James	-Smith	3
John	Johnson	4

Dot notation

To select all values from a specific table, the wildcard character can be applied to the table with *dot notation*. Consider the following query:

```
SELECT
    Employees.*, Departments.Name
FROM
    Employees
JOIN
    Departments
    ON Departments.Id = Employees.DeptId
```

This will return a data set with all fields on the Employeetable, followed by just the Namefield in the Departments

table:

Id FName	LName	DeptId	Name
James	-Smith	3	Finance
John	Johnson	4	IT

Warnings Against Use

It is generally advised that using *is avoided in production code where possible, as it can cause a number of potential problems including:

- Excess IO, network load, memory use, and so on, due to the database engine reading data that is not needed and transmitting it to the front-end code. This is particularly a concern where there might be large fields such as those used to store long notes or attached files.
- Further excess IO load if the database needs to spool internal results to disk as part of the processing for a query more complex than SELECT <columns> FROM <table>.
- Extra processing (and/or even more IO) if some of the unneeded columns are: computed columns in databases that support them

in the case of selecting from a view, columns from a table/view that the query optimiser could otherwise optimise out

The potential for unexpected errors if columns are added to tables and views later that results ambiguous column names. For example SELECT * FROM orders JOIN people ON people.id = orders.personid ORDER BY displayname- if a column column called displaynameis added to the orders table to allow users to give their orders meaningful names for future reference then the column name will appear twice in the output so the ORDER BY clause will be ambiguous which may cause errors ("ambiguous column name" in recent MS SQL Server versions), and if not in this example your application code might start displaying the order name where the person name is intended because the new column is the first of that name returned, and so on.

When Can You Use *, Bearing The Above Warning In Mind?

While best avoided in production code, using *is fine as a shorthand when performing manual queries against the database for investigation or prototype work.

Sometimes design decisions in your application make it unavoidable (in such circumstances, prefer tablealias.*

over just *where possible).

When using EXISTS, such as SELECT A.col1, A.Col2 FROM A WHERE EXISTS (SELECT * FROM B where A.ID = B.A_ID), we are not returning any data from B. Thus a join is unnecessary, and the engine knows no values from B are to be returned, thus no performance hit for using *. Similarly COUNT(*)is fine as it also doesn't actually return any of the columns, so only needs to read and process those that are used for filtering purposes.

Section 3.2: SELECT Using Column Aliases

Column aliases are used mainly to shorten code and make column names more readable.

Code becomes shorter as long table names and unnecessary identification of columns (e.g., there may be 2 IDs in the table, but only one is used in the statement) can be avoided. Along with table aliases this allows you to use longer descriptive names in your database structure while keeping queries upon that structure concise.

Furthermore they are sometimes required, for instance in views, in order to name computed outputs.

All versions of SQL

Aliases can be created in all versions of SQL using double quotes (").

```
SELECT
    FName AS "First Name", MName
    AS "Middle Name", LName AS
    "Last Name"
FROM Employees
```

Different Versions of SQL

You can use single quotes ('), double quotes (") and square brackets ([]) to create an alias in Microsoft SQL Server

```
SELECT
    FName AS "First Name", MName
    AS 'Middle Name', LName AS
    [Last Name]
FROM Employees
```

Both will result in:

First Name Middle Name Last Name

James	John	Smith
John	James	Johnson
Michael	Marcus	Williams

This statement will return FName and LName columns with a given name (an alias). This is achieved using the AS operator followed by the alias, or simply writing alias directly after the column name. This means that the following query has the same outcome as the above.

```
SELECT
    FName "First Name", MName
    "Middle Name", LName "Last
    Name"
FROM Employees
```

First Name Middle Name Last Name

James	John	Smith
John	James	Johnson
Michael	Marcus	Williams

However, the explicit version (i.e., using the ASoperator) is more readable.

If the alias has a single word that is not a reserved word, we can write it without single quotes, double quotes or brackets:

```
SELECT
    FName AS FirstName,
    LName AS LastName
FROM Employees
```

FName	LName
James	Smith
John	Johnson
Micheal	Williams

A further variation available in MS SQL Server amongst others is **<alias>=<column-or-calculation>**, for instance:

```
SELECT FullName = FirstName + ' ' + LastName,
        Addr1      = FullStreetAddress,
FROM CustomerDetails
```

which is equivalent to:

```
SELECT FirstName + ' ' + LastName As FullName FullStreetAddress
                                    As Addr1,
        TownName                    As Addr2
FROM CustomerDetails
```

Both will result in:

FName	Addr1	Addr2
James Smith	123	AnyStreet TownVille
John Johnson	668	MyRoad Anytown
Micheal Williams	999	High End Dr Williamsburg

Some find using =instead of Aseasier to read, though many recommend against this format, mainly because it is not standard so not widely supported by all databases. It may cause confusion with other uses of the =character.

All Versions of SQL

Also, if you *need* to use reserved words, you can use brackets or quotes to escape:

```
SELECT
    FName AS FirstName,
    LName AS LastName
FROM
    Employees
ORDER BY
    LastName DESC
```

However, you may not use

```
SELECT
    FName AS SELECT,
    LName AS FROM
 FROM
    Employ-
ees ORDER BY
    LastName DESC
```

To create an alias from these reserved words (SELECT and FROM). This will cause numerous errors on execution.

Section 3.3: Select Individual Columns

```
SELECT
    PhoneNumber, Email,
    PreferredContact
FROM Customers
```

This statement will return the columns PhoneNumber, Email, and PreferredContact from all rows of the Customers

table. Also the columns will be returned in the sequence in which they appear in the SELECT clause.

The result will be:

PhoneNumber Email PreferredContact

3347927472 william.jones@example.com PHONE

2137921892 dmiller@example.net EMAIL NULL richard0123@example.com EMAIL

If multiple tables are joined together, you can select columns from specific tables by specifying the table name before the column name: [table_name].[column_name]

```
SELECT
    Customers.PhoneNumber, Custom-
    ers.Email, Custom-
    ers.PreferredContact, Orders.Id AS
    OrderId
FROM
    Custom-
ers LEFT JOIN
    Orders ON Orders.CustomerId = Customers.Id
```

*AS OrderId means that the Id field of Orders table will be returned as a column named OrderId. See selecting with column alias for further information.

To avoid using long table names, you can use table aliases. This mitigates the pain of writing long table names for each field that you select in the joins. If you are performing a self join (a join between two instances of the same table), then you must use table aliases to distinguish your tables. We can write a table alias like Customers c or Customers AS c. Here c works as an alias for Customers and we can select let's say Email like this: c.Email.

```
SELECT
    c.PhoneNumber,
    c.Email,
    c.PreferredContact, o.Id
    AS OrderId
FROM
    Customers c
LEFT JOIN
    Orders o ON o.CustomerId = c.Id
```

Section 3.4: Selecting specified number of records

The SQL 2008 standard defines the FETCH FIRST clause to limit the number of records returned.

```
SELECT Id, ProductName, UnitPrice, Package FROM Prod-
uct
ORDER BY UnitPrice DESC
FETCH FIRST 10 ROWS ONLY
```

This standard is only supported in recent versions of some RDMSs. Vendor-specific non-standard syntax is provided in other systems. Progress OpenEdge 11.x also supports the FETCH FIRST <n> ROWS ONLY syntax.

Additionally, OFFSET <m> ROWS before FETCH FIRST <n> ROWS ONLY allows skipping rows before fetching rows.

```
SELECT Id, ProductName, UnitPrice, Package FROM Prod-
uct
ORDER BY UnitPrice DESC
OFFSET 5 ROWS
FETCH FIRST 10 ROWS ONLY
```

The following query is supported in SQL Server and MS Access:

```
SELECT TOP 10 Id, ProductName, UnitPrice, Package FROM Product
ORDER BY UnitPrice DESC
```

To do the same in MySQL or PostgreSQL the LIMIT keyword must be used:

```
SELECT Id, ProductName, UnitPrice, Package FROM Prod-
uct
ORDER BY UnitPrice DESC
LIMIT 10
```

In Oracle the same can be done with ROWNUM:

```
SELECT Id, ProductName, UnitPrice, Package FROM Prod-
uct
WHERE ROWNUM <= 10
ORDER BY UnitPrice DESC
```

Results: 10 records

Id	Product Name	Unit Price	Package
38	Côte de Blaye	263.50	12 - 75 cl bottles
29	Thüringer Rostbratwurst	123.79	50 bags x 30 sausgs.
9	Mishi Kobe Niku	97.00	18 - 500 g pkgs.
20	Sir Rodney's Marmalade	81.00	30 gift boxes
18	Carnarvon Tigers	62.50	16 kg pkg.
59	Raclette Courdavault	55.00	5 kg pkg.
51	Manjimup Dried Apples	53.00	50 - 300 gpkgs.
62	Tarte au sucre	49.30	48 pies
43	Ipoh Coffee	46.00	16 - 500 g tins
28	Rössle Sauerkraut	45.60	25 - 825 g cans

Vendor Nuances:

It is important to note that the TOP in Microsoft SQL operates after the WHERE clause and will return the specified number of results if they exist anywhere in the table, while ROWNUM works as part of the WHERE clause so if other conditions do not exist in the specified number of rows at the beginning of the table, you will get zero results when there could be others to be found.

Section 3.5: Selecting with Condition

The basic syntax of SELECT with WHERE clause is:

```
SELECT column1, column2, columnN FROM table_name
WHERE [condition]
```

The [condition] can be any SQL expression, specified using comparison or logical operators like >, <, =, <>, >=, <=, LIKE, NOT, IN, BETWEEN etc.

The following statement returns all columns from the table 'Cars' where the status column is 'READY':

```
SELECT * FROM Cars WHERE status = 'READY'
```

See WHERE and HAVING for more examples.

Section 3.6: Selecting with CASE

When results need to have some logic applied 'on the fly' one can use CASE statement to implement it.

```
SELECT CASE WHEN Col1 < 50 THEN 'under' ELSE 'over' END threshold FROM TableName
```

also can be chained

```
SELECT
    CASE WHEN Col1 < 50 THEN 'under'
        WHEN Col1 > 50 AND Col1 <100 THEN 'between' ELSE 'over'
    END threshold FROM TableName
```

one also can have CASE inside another CASE statement

```
SELECT
    CASE WHEN Col1 < 50 THEN 'under' ELSE
        CASE WHEN Col1 > 50 AND Col1 <100 THEN Col1
        ELSE 'over' END
    END threshold
FROM TableName
```

Section 3.7: Select columns which are named after reserved keywords

When a column name matches a reserved keyword, standard SQL requires that you enclose it in double quotation marks:

```
SELECT
    "ORDER"
    , ID
FROM ORDERS
```

Note that it makes the column name case-sensitive.

Some DBMSes have proprietary ways of quoting names. For example, SQL Server uses square brackets for this purpose:

```
SELECT
    [Order], ID
FROM ORDERS
```

while MySQL (and MariaDB) by default use backticks:

```
SELECT
	`Order`,
	id
FROM orders
```

Section 3.8: Selecting with table alias

```
SELECT e.Fname, e.LName FROM Employees e
```

The Employees table is given the alias 'e' directly after the table name. This helps remove ambiguity in scenarios where multiple tables have the same field name and you need to be specific as to which table you want to return data from.

```
SELECT e.Fname, e.LName, m.Fname AS ManagerFirstName FROM Employees e
	JOIN Managers m ON e.ManagerId = m.Id
```

Note that once you define an alias, you can't use the canonical table name anymore. i.e.,

```
SELECT e.Fname, Employees.LName, m.Fname AS ManagerFirstName FROM Employees e
JOIN Managers m ON e.ManagerId = m.Id
```

would throw an error.

Note that although an alias/range variable must be declared for the dervied table (otherwise SQL will throw an error), it never makes sense to actually use it in the query.

Section 3.9: Selecting with more than 1 condition

The AND keyword is used to add more conditions to the query.

Name Age Gender

Sam 18 M

John 21 M

Bob 22 M

Mary 23 F

```
SELECT name FROM persons WHERE gender = 'M' AND age > 20;
```

This will return:

Name John Bob

using OR keyword

```
SELECT name FROM persons WHERE gender = 'M' OR age < 20;
```

This will return:

name Sam John Bob

These keywords can be combined to allow for more complex criteria combinations:

```
SELECT name
FROM pesons
WHERE (gender = 'M' AND age < 20) OR
      (gender = 'F' AND age > 20);
```

This will return:

name Sam Mary

Section 3.11: Selecting with Aggregate functions

Average

The AVG() aggregate function will return the average of values selected.

```
SELECT AVG(Salary) FROM Employees
```

Aggregate functions can also be combined with the where clause.

```
SELECT AVG(Salary) FROM Employees where DepartmentId = 1
```

Aggregate functions can also be combined with group by clause.

If employee is categorized with multiple department and we want to find avg salary for every department then we can use following query

```
SELECT AVG(Salary) FROM Employees GROUP BY DepartmentId
```

Minimum

The MIN() aggregate function will return the minimum of values selected.

```
SELECT MIN(Salary) FROM Employees
```

Maximum

The MAX() aggregate function will return the maximum of values selected.

```
SELECT MAX(Salary) FROM Employees
```

Count

The COUNT() aggregate function will return the count of values selected.

```
SELECT Count(*) FROM Employees
SELECT Count(*) FROM Employees where ManagerId IS NOT NULL
```

Specific columns can also be specified to get the number of values in the column. Note that NULL values are not counted.

```
Select Count(ManagerId) from Employees
```

Count can also be combined with the distinct keyword for a distinct count.

Sum

The SUM() aggregate function returns the sum of the values selected for all rows.

Section 3.12: Select with condition of multiple values from column

```
SELECT * FROM Cars          WHERE status IN ( 'Waiting', 'Working' )
```

This is semantically equivalent to

```
SELECT * FROM Cars          WHERE ( status = 'Waiting' OR status = 'Working' )
```

i.e. value IN (<value list>) is a shorthand for disjunction (logical OR).

Section 3.14: Selection with sorted Resultsz

Counting rows based on a specific column value:

```
SELECT category, COUNT(*) AS item_count FROM item
GROUP BY category;
```

Getting average income by department:

```
SELECT department, AVG(income) FROM employees
GROUP BY department;
```

The important thing is to select only columns specified in the GROUP BY clause or used with aggregate functions. There WHERE clause can also be used with GROUP BY, but WHERE filters out records before any grouping is done:

```
SELECT department, AVG(income) FROM
employees
WHERE department <> 'ACCOUNTING'
GROUP BY department;
```

If you need to filter the results after the grouping has been done, e.g, to see only departments whose average income is larger than 1000, you need to use the HAVINGclause:

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
GROUP BY department
HAVING avg(income) > 1000;
```

Section 3.14: Selection with sorted Results

```
SELECT * FROM Employees ORDER BY LName
```

This statement will return all the columns from the table Employees.

Id FName LName PhoneNumber

John Johnson 2468101214 1

James Smith 1234567890

Michael Williams 1357911131

```
SELECT * FROM Employees ORDER BY LName DESC
```

Or

```
SELECT * FROM Employees ORDER BY LName ASC
```

This statement changes the sorting direction.

One may also specify multiple sorting columns. For example:

```
SELECT * FROM Employees ORDER BY LName ASC, FName ASC
```

This example will sort the results first by LNameand then, for records that have the same LName, sort by FName. This will give you a result similar to what you would find in a telephone book.

In order to save retyping the column name in the ORDER BYclause, it is possible to use instead the column's number. Note that column numbers start from 1.

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY 3
```

You may also embed a CASE statement in the ORDER BY clause.

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY CASE WHEN LName='Jones' THEN 0 ELSE 1 END ASC
```

This will sort your results to have all records with the LName of "Jones" at the top.

Section 3.15: Selecting with null

```
SELECT Name FROM Customers WHERE PhoneNumber IS NULL
```

Selection with nulls take a different syntax. Don't use =, use IS NULL or IS NOT NULL instead.

Section 3.16: Select distinct (unique values only)

```
SELECT DISTINCT ContinentCode FROM Countries;
```

This query will return all DISTINCT (unique, different) values from ContinentCode column from Countries table

ContinentCode

OC EU AS NA AF

SQLFiddle Demo

Section 3.17: Select rows from multiple tables

```
SELECT
* FROM
    table1
    ,
    table2

SELECT
    table1.column1,
    table1.column2,
    table2.column1
FROM
    table1
    ,
    table2
```

This is called cross product in SQL it is same as cross product in sets

These statements return the selected columns from multiple tables in one query. There is no specific relationship between the columns returned from each table.

Chapter 4: GROUP BY

Results of a SELECT query can be grouped by one or more columns using the GROUP BY statement: all results with the same value in the grouped columns are aggregated together. This generates a table of partial results, instead of one result. GROUP BY can be used in conjunction with aggregation functions using the HAVINGstatement to define how non-grouped columns are aggregated.

Section 4.1: Basic GROUP BY example

It might be easier if you think of GROUP BY as "for each" for the sake of explanation. The query below:

```
SELECT EmpID, SUM (MonthlySalary) FROM Employee
GROUP BY EmpID
```

is saying:

"Give me the sum of MonthlySalary's for each EmpID" So if your table looked like this:

```
+-----+-------------+
|EmpID|MonthlySalary|
+-----+-------------+
|1    |200          |
+-----+-------------+
|2    |300          |
+-----+-------------+
```

Result:

```
+-+---+
|1|200|
+-+---+
|2|300|
+-+---+
```

Sum wouldn't appear to do anything because the sum of one number is that number. On the other hand if it looked like this:

```
+-----+-------------+
|EmpID|MonthlySalary|
+-----+-------------+
|1    |200          |
+-----+-------------+
|1    |300          |
+-----+-------------+
|2    |300          |
+-----+-------------+
```

Result:

```
+-+---+
|1|500|
+-+---+
|2|300|
+-+---+
```

Then it would because there are two EmpID 1's to sum together.

Section 4.2: Filter GROUP BY results using a HAVING clause

A HAVING clause filters the results of a GROUP BY expression. Note: The following examples are using the Library example database.

Examples:

Return all authors that wrote more than one book (live example).SELECT

```
    a.Id,
    a.Name,
    COUNT(*) BooksWritten
FROM BooksAuthors ba
    INNER JOIN Authors a ON a.id = ba.authorid GROUP BY
    a.Id,
    a.Nam
    e
HAVING COUNT(*) > 1              -- equals to HAVING BooksWritten > 1
;
```

Return all books that have more than three authors (live example).

```
 SELECT
    b.Id,
    b.Title,
    COUNT(*) NumberOfAuthors
 FROM BooksAuthors ba
    INNER JOIN Books b ON b.id = ba.bookid GROUP BY
    b.Id,
    b.Title
 HAVING COUNT(*) > 3              -- equals to HAVING NumberOfAuthors > 3
 ;
```

Section 4.3: USE GROUP BY to COUNT the number of rows for each unique entry in a given column

Let's say you want to generate counts or subtotals for a given value in a column. Given this table, "Westerosians":

Name GreatHouseAllegience

Arya Stark Cercei Lannister Myrcella Lannister Yara Greyjoy Catelyn Stark Sansa Stark

Without GROUP BY, COUNT will simply return a total number of rows:

```
SELECT Count(*) Number_of_Westerosians FROM
Westerosians
```

returns...

Number_of_Westerosians

6

But by adding GROUP BY, we can COUNT the users for each value in a given column, to return the number of people in a given Great House, say:

```
SELECT GreatHouseAllegience House, Count(*)
Number_of_Westerosians FROM Westerosians
GROUP BY GreatHouseAllegience
```

returns...

House Number_of_Westerosians

Stark 3

Greyjoy 1

Lannister 2

It's common to combine GROUP BY with ORDER BY to sort results by largest or smallest category:

```
SELECT GreatHouseAllegience House, Count(*) Num-
ber_of_Westerosians FROM Westerosians
GROUP BY GreatHouseAllegience
ORDER BY Number_of_Westerosians Desc
```

returns...

House Number_of_Westerosians

Stark 3

Lannister 2

Greyjoy 1

Chapter 5: ORDER BY

Section 5.1: Sorting by column number (instead of name)

You can use a column's number (where the leftmost column is '1') to indicate which column to base the sort on, instead of describing the column by its name.

Pro: If you think it's likely you might change column names later, doing so won't break this code.

Con: This will generally reduce readability of the query (It's instantly clear what 'ORDER BY Reputation' means, while 'ORDER BY 14' requires some counting, probably with a finger on the screen.)

This query sorts result by the info in relative column position 3 from select statement instead of column name

Reputation.

SELECT DisplayName, JoinDate, Reputation FROM Users ORDER BY 3

DisplayName JoinDate Reputation

Community 2008-09-15 1

Jarrod Dixon 2008-10-03 11739

Geoff Dalgas 2008-10-03 12567

Joel Spolsky 2008-09-16 25784

Jeff Atwood 2008-09-16 37628

Section 5.2: Use ORDER BY with TOP to return the top x rows based on a column's value

In this example, we can use GROUP BY not only determined the sort of the rows returned, but also what rows are

returned, since we're using TOP to limit the result set.

Let's say we want to return the top 5 highest reputation users from an unnamed popular Q&A site.

Without ORDER BY

This query returns the Top 5 rows ordered by the default, which in this case is "Id", the first column in the table (even though it's not a column shown in the results).

```
SELECT TOP 5 DisplayName, Reputation FROM Users
```

returns...

DisplayName Reputation

Community 1

Geoff Dalgas 12567

Jarrod Dixon 11739

Jeff Atwood 37628

Joel Spolsky 25784

With ORDER BY

```
SELECT TOP 5 DisplayName,
Reputation FROM Users
ORDER BY Reputation desc
```

returns...

DisplayName Reputation

JonSkeet **865023**

Darin Dimitrov **661741**

BalusC **650237**

Hans Passant **625870**

Marc Gravell **601636**

Remarks

Some versions of SQL (such as MySQL) use a LIMITclause at the end of a SELECT, instead of TOPat the beginning, for example:

```
SELECT DisplayName, Reputa-
tion FROM Users
ORDER BY Reputation
DESC LIMIT 5
```

Section 5.3: Customized sorting order

To sort this table Employeeby department, you would use ORDER BY Department. However, if you want a different sort order that is not alphabetical, you have to map the Departmentvalues into different values that sort correctly; this can be done with a CASE expression:

Name Department

Hasan IT

Yusuf HR

Hillary HR

Joe IT

Merry HR

Ken Accountant

```
SELECT *
FROM Employee
ORDER BY CASE Department
        WHEN 'HR'         THEN 1
        WHEN 'Accountant' THEN 2 ELSE 3
        END;
```

Name Department

Yusuf **HR** Hillary **HR**
Merry **HR**

Ken **Accountant**

Hasan **IT**

Joe **IT**

Section 5.4: Order by Alias

Due to logical query processing order, alias can be used in order by.

```
SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY jd, rep
```

And can use relative order of the columns in the select statement .Consider the same example as above and instead of using alias use the relative order like for display name it is 1 , for Jd it is 2 and so on

```
SELECT DisplayName, JoinDate as jd, Reputation as rep FROM Users
ORDER BY 2, 3
```

Section 5.5: Sorting by multiple columns

SELECT DisplayName, JoinDate, Reputation FROM Users ORDER BY JoinDate, Reputation

DisplayName JoinDate Reputation

Community	**2008-09-15 1**
Jeff Atwood	**2008-09-16 25784**
Joel Spolsky	2008-09-16**37628**
Jarrod Dixon	**2008-10-03 11739**
Geoff Dalgas	2008-10-03**12567**

Chapter 6: AND & OR Operators

Section 6.1: AND OR Example

Have a table

Name Age City

Bob 10 Paris

Mat 20 Berlin

Mary 24 Prague

```
select Name from table where Age>10 AND City='Prague'
```

Gives

Name

Mary

```
select Name from table where Age=10 OR City='Prague'
```

Gives

Name

Bob

Mary

Chapter 6: LIKE operator

Section 6.1: Match open-ended pattern

The %wildcard appended to the beginning or end (or both) of a string will allow 0 or more of any character before the beginning or after the end of the pattern to match.

Using '%' in the middle will allow 0 or more characters between the two parts of the pattern to match. We are going to use this Employees Table:

Id	FName	LName	Phone Number	Manager ID	DepartmentID	Salary	Hire Date
1	John	Johnson	2468101214	1	1	400	23-03-2005
2	Sophie	Amudsen	2479100211	1	1	400	11-01-2010
3	Ronny	SMith	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005
5	Hilde	Knag	2468021911	2	1	800	01-01-2000

Following statement matches for all records having FName **containing**string 'on' from Employees Table.

```
SELECT * FROM Employees WHERE FName LIKE '%on%';
```

Id	FName	LName	Phone Number	Manager ID	DepartmentID	Salary	Hire Date
3	Ronny	SMith	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005

Following statement matches all records having PhoneNumber **starting with** string '246' from Employees

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '246%';
```

Id FName LName PhoneNumber ManagerId DepartmentId Salary Hire_date

2 Sophie Amudsen 247910011	1	1	400	11-01-2010
5 Hilde Knag 246802191	2	1	800	01-01-2000

All records where Fname **3rd character** is 'n' from Employees.

```
SELECT * FROM Employees WHERE FName LIKE '_n%';
```

(two underscores are used before 'n' to skip first 2 characters)

Id FName LName PhoneNumber ManagerId DepartmentId Salary Hire_date

3 Ronny Smith	2462544026	2	1	600	06-08-2015
4 Jon Sanchez	2454124602	1	1	400	23-03-2005

Section 6.2: Single character match

To broaden the selections of a structured query language (SQL-SELECT) statement, wildcard characters, the percent

sign (%) and the underscore (_), can be used.

The _ (underscore) character can be used as a wildcard for any single character in a pattern match. Find all employees whose Fname start with 'j' and end with 'n' and has exactly 3 characters in

```
SELECT * FROM Employees WHERE FName LIKE 'j_n'
```

Fname.

_ (underscore) character can also be used more than once as a wild card to match patterns. For example, this pattern would match "jon", "jan", "jen", etc.

These names will not be shown "jn","john","jordan", "justin", "jason", "julian", "jillian", "joann" because in our query one underscore is used and it can skip exactly one character, so result must be of 3 character Fname.

For example, this pattern would match "LaSt", "LoSt", "HaLt", etc.

```
SELECT * FROM Employees WHERE FName LIKE '_A_T'
```

Section 6.3: Match by range or set

Match any single character within the specified range (e.g.: [a-f]) or set (e.g.: [abcdef]). This range pattern would match "gary" but not "mary":

```
SELECT * FROM Employees WHERE FName LIKE '[a-g]ary'
```

This set pattern would match "mary" but not "gary":

```
SELECT * FROM Employees WHERE Fname LIKE '[lmnop]ary'
```

The range or set can also be negated by appending the ^ caret before the range or set: This range pattern would not match "gary" but will match "mary":

```
SELECT * FROM Employees WHERE FName LIKE '[^a-g]ary'
```

This set pattern would not match "mary" but will match"gary":

```
SELECT * FROM Employees WHERE Fname LIKE '[^lmnop]ary'
```

Section 6.4: Wildcard characters

wildcard characters are used with the SQL LIKE operator. SQL wildcards are used to search for data within a table. Wildcards in SQL are:%, _, [charlist], [^charlist]

% - A substitute for zero or more characters

```
Eg:    //selects all customers with a City starting with "Lo" SELECT *
       FROM Customers
       WHERE City LIKE 'Lo%';
      //selects all customers with a City containing the pattern "es" SELECT *
    FROM Customers
     WHERE City LIKE '%es%';
```

- A substitute for a single character

```
Eg://selects all customers with a City starting with any character, followed by "erlin" SELECT *
FROM Customers
WHERE City LIKE '_erlin';
```

charlist] - Sets and ranges of characters to match

Eg://selects all customers with a City starting with "a", "d", or "l"

```
SELECT * FROM Custom-
ers WHERE City LIKE
'[adl]%';
```

```
//selects all customers with a City starting with "a", "d", or "l" SELECT
* FROM Customers
WHERE City LIKE '[a-c]%';
```

[^charlist] - Matches only a character NOT specified within the brackets

Eg://selects all customers with a City starting with a character that is not "a", "p", or "l" SELECT * FROM Customers

```
WHERE City LIKE

'[^apl]%'; or

SELECT * FROM Customers
WHERE City NOT LIKE '[apl]%' and city like '_%';
```

Chapter 7: IN clause

Section 7.1: Simple IN clause

To get records having any of the given ids

```
select  *
from
products
where id in (1,8,3)
```

The query above is equal to

```
select  *
from
```

```
products
where id = 1
    or id = 8
    or id = 3
```

Section 7.2: Using IN clause with a subquery

```
SELECT *
FROM
customers
WHERE id IN
(
    SELECT DISTINCT
    customer_id FROM orders
);
```

The above will give you all the customers that have orders in the system.

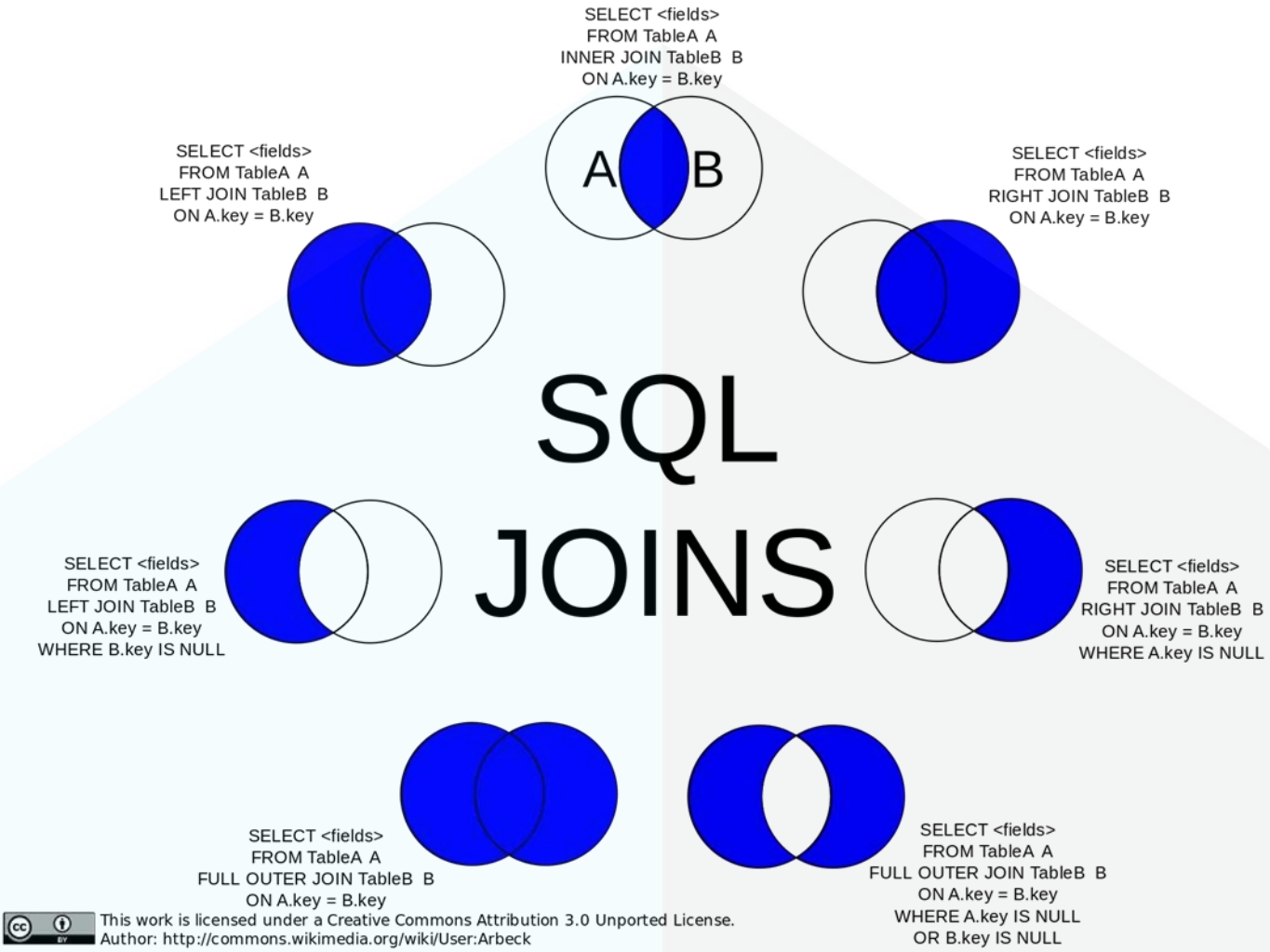
Chapter 8: JOIN

JOIN is a method of combining (joining) information from two tables. The result is a stitched set of columns from both tables, defined by the join type (INNER/OUTER and LEFT/RIGHT/FULL, explained below) and join criteria (how rows from both tables relate).

A table may be joined to itself or to any other table. If information from more than two tables needs to be accessed, multiple joins can be specified in a FROM clause.

Section 8.1: Differences between inner/outer joins

SQL has various join types to specify whether (non-)matching rows are included in the result: INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN (the INNER and OUTER keywords are optional). The figure below underlines the differences between these types of joins: the blue area represents the results returned by the join, and the white area represents the results that the join will not return




```
A     B
-     -
1     3
2     4
```

Note that (1,2) are unique to A, (3,4) are common, and (5,6) are unique to B.

Inner Join

An inner join using either of the equivalent queries gives the intersection of the two tables, i.e. the two rows they have in common:

```
select * from a INNER JOIN b on a.a = b.b;

a | b
--+--
3 | 3
4 | 4
```

Left outer join

A left outer join will give all rows in A, plus any common rows in B:

```
select * from a LEFT OUTER JOIN b on a.a = b.b;
```

```
a |      b
--+-----
| null
| null 3 |   3
```

Right outer join

Similarly, a right outer join will give all rows in B, plus any common rows in A:

```
select * from a RIGHT OUTER JOIN b on a.a = b.b;
```

```
a       |    b
-----+----
        3
3       |
        4
null |
```

Full outer join

A full outer join will give you the union of A and B, i.e., all the rows in A and all the rows in B. If something in A doesn't have a corresponding datum in B, then the B portion is null, and vice versa.

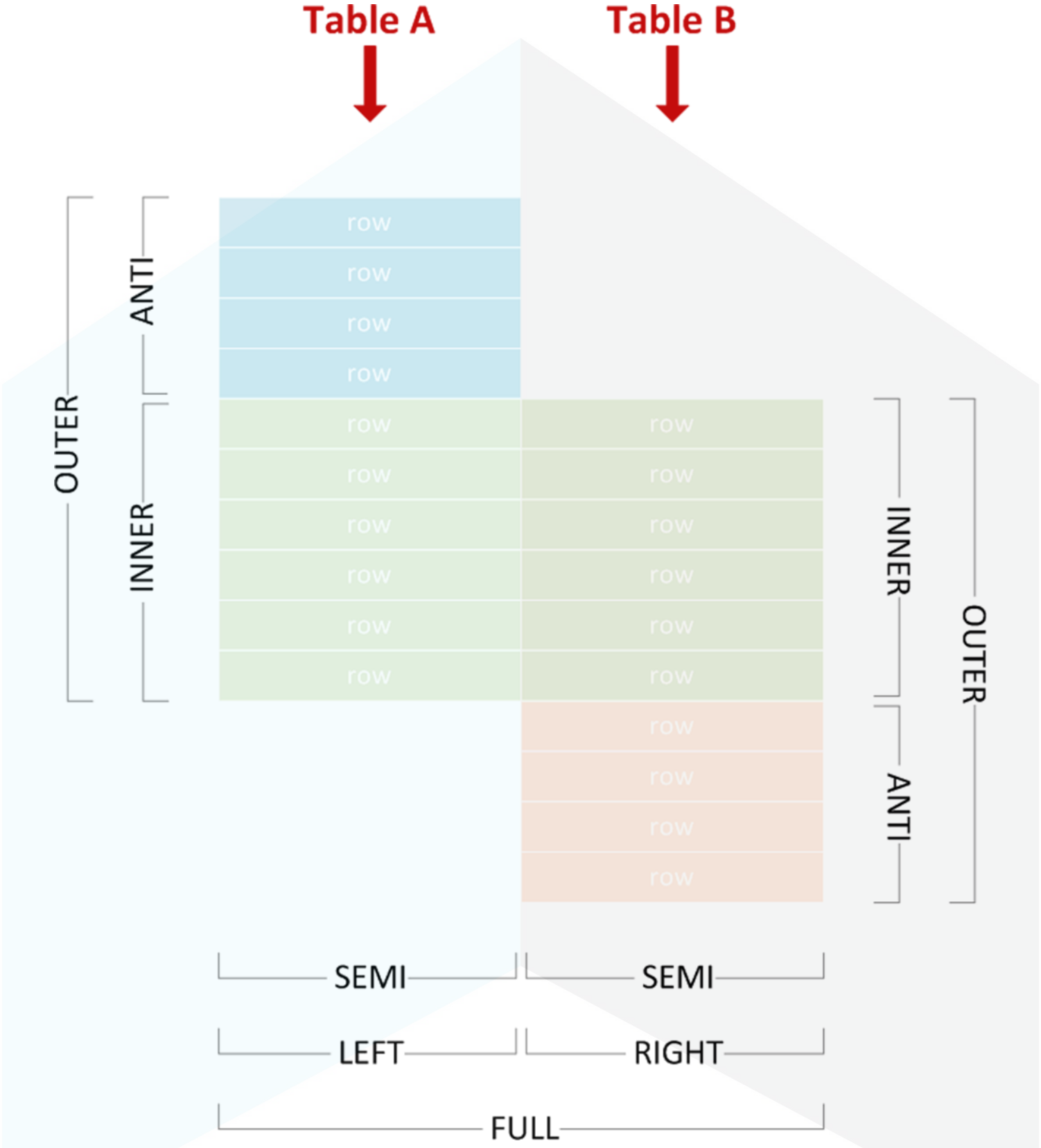
```
select * from a FULL OUTER JOIN b on a.a = b.b;
```

```
a      |    b
```

1 | null		
2 | null		
3 |	3	
4 |	4	
null	|	6
null	|	5

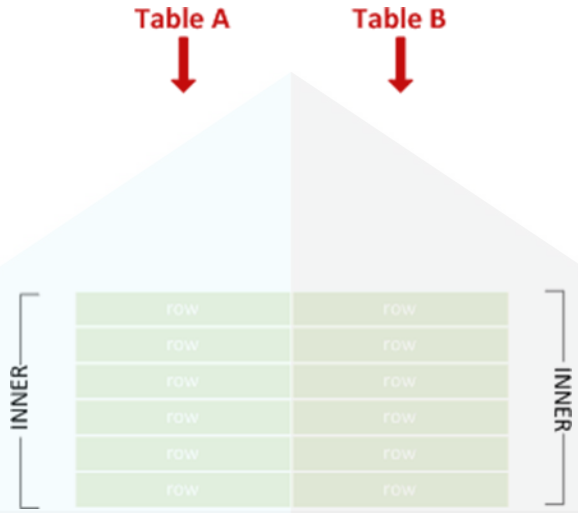
Section 8.2: JOIN Terminology: Inner, Outer

Let's say we have two tables (A and B) and some of their rows match (relative to the given JOIN condition, whatever it may be in the particular case):



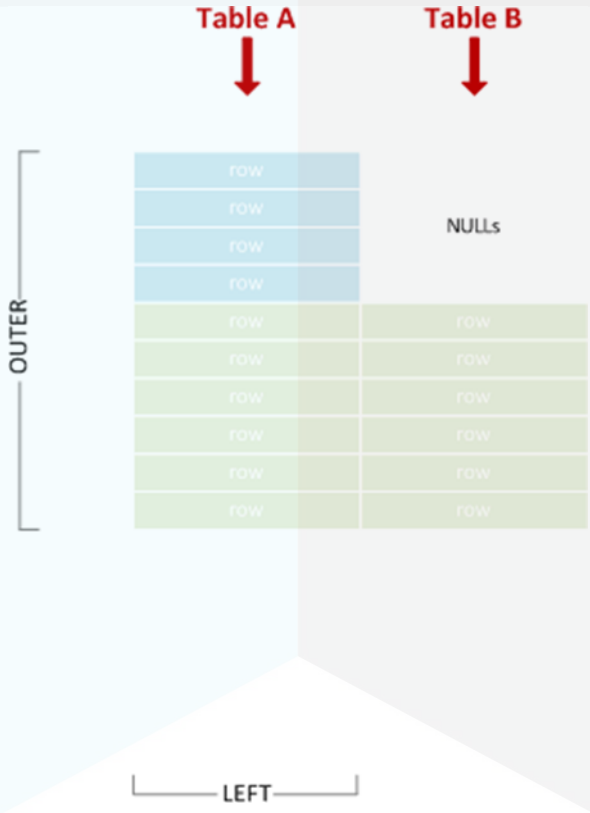
Inner Join

Combines left and right rows that match.



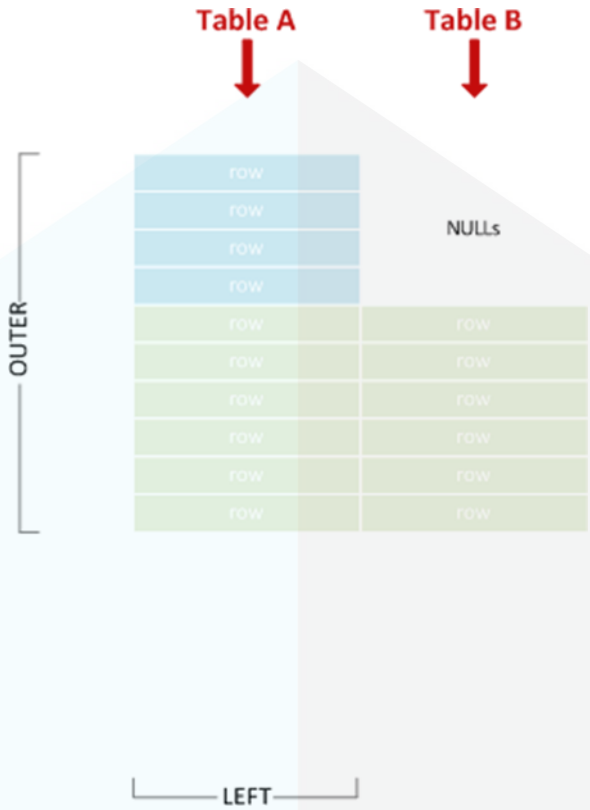

```
SELECT * FROM A JOIN B ON X = Y;
```

X	Y
Lisa	Lisa
Marco	Marco



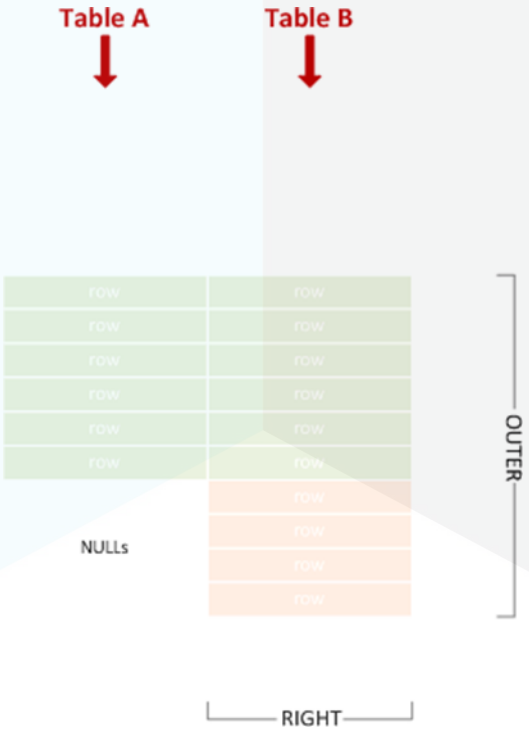

```
SELECT * FROM A LEFT JOIN B ON X = Y;
```

X	Y
Amy	NULL
John	NULL
Lisa	Lisa



Left Outer Join

Sometimes abbreviated to "left join". Combines left and right rows that match, and includes non-matching left rows.

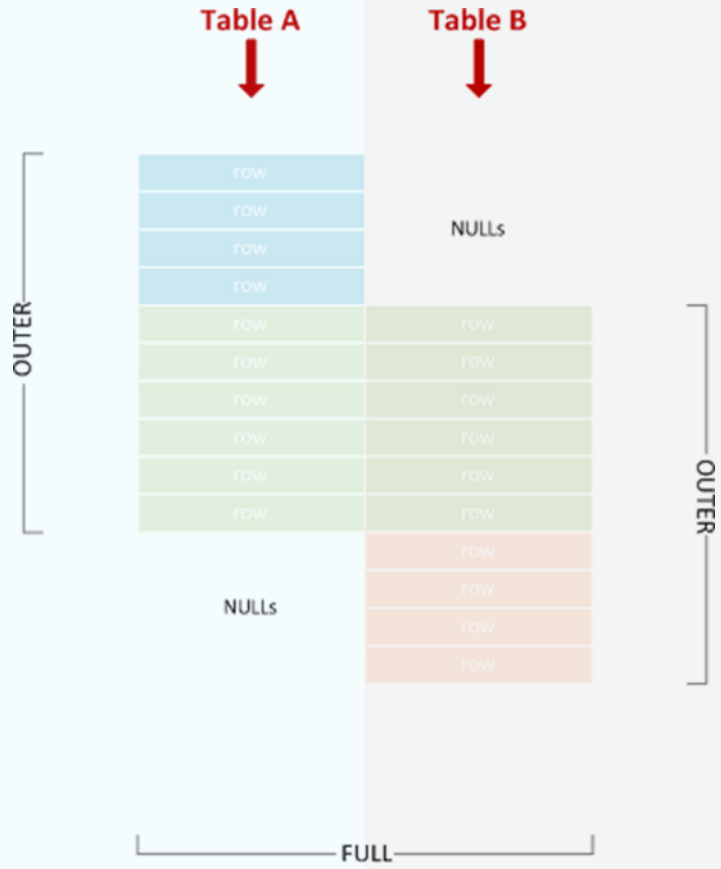


Right Outer Join

Sometimes abbreviated to "right join". Combines left and right rows that match, and includes non-matching right rows

```
SELECT * FROM A RIGHT JOIN B ON X = Y;
```

X	Y
Lisa	Lisa
Marco	Marco
Phil	Phil
NULL	Tim



Full OuterJoin

```
SELECT * FROM A FULL JOIN B ON X = Y;
```

X	Y
Amy	NULL
John	NULL
Lisa	Lisa
Marco	Marco
Phil	Phil

Sometimes abbreviated to "full join". Union of left and right outer join