

SQL

Introduction to SQL

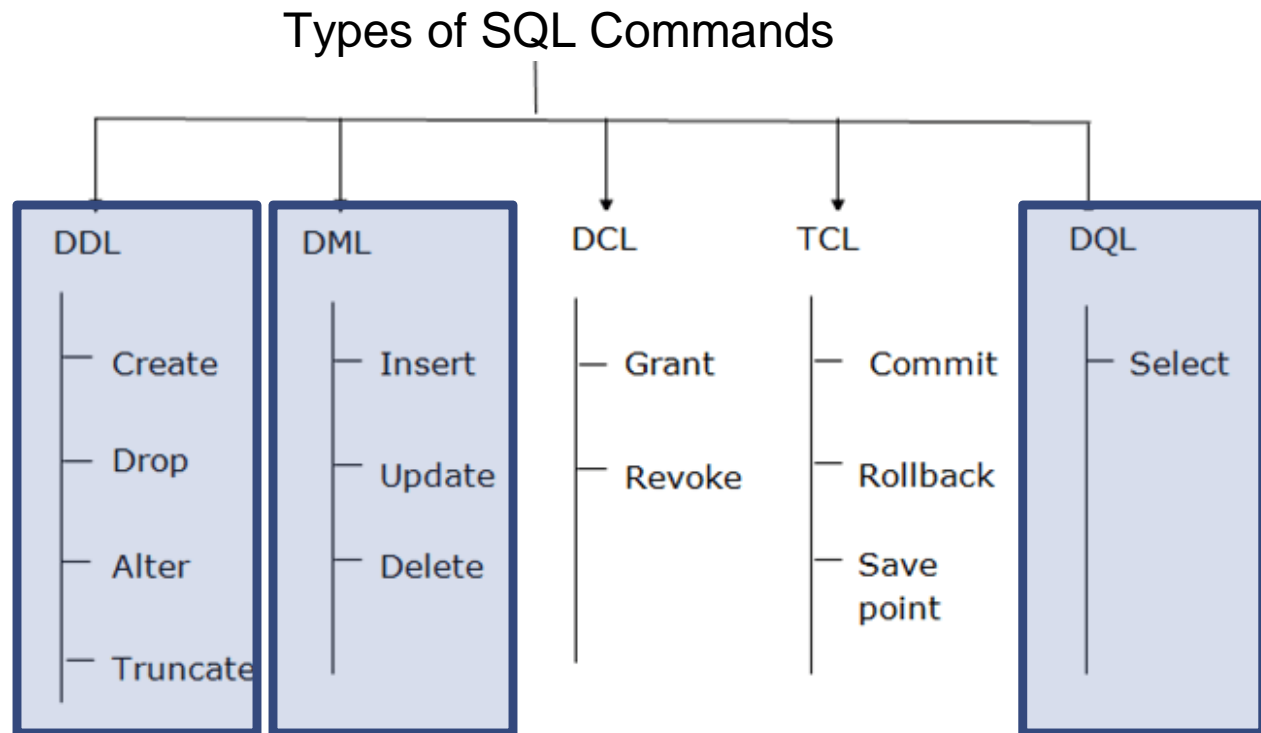
SQL OVERVIEW

THE IMPORTANCE OF SQL

- Relational Databases are everywhere
- SQL is used with all Relational Databases and many NoSQL
 - to access (and create, edit, delete, etc.) data stored in a database you need.... SQL
- SQL is easy to learn
 - SQL is relatively simple to learn because SQL query syntax relies on common English words
 - Learning SQL helps with learning how to think data manipulation
- Knowledge of SQL is a highly marketable skill (and important in interviews)
- SQL code can be integrated in most programming languages, including Python and R (and SAS, Stata, MATLAB, etc.)
 - let the database server process the SQL statement and return a smaller result set

SQL DEFINED

- SQL (pronounced S-Q-L or sequel) is short for Structured Query Language and is used for:
 - creating, maintaining, and dropping databases and database objects
 - inserting, deleting, and updating data in databases
 - retrieving and transforming data in databases



SQL DDL STATEMENTS

DDL – CREATE - EXAMPLE

- The CREATE clause is used to create databases, tables, indexes, views, etc.

For example:

```
CREATE TABLE IF NOT EXISTS Customer (  
    CustomerID SERIAL,  
    CustomerName TEXT NOT NULL,  
    CreditLimit NUMERIC CHECK(CreditLimit>0),  
    PaymentDays INTEGER CHECK(PaymentDays>0) NOT NULL DEFAULT 30,  
    PRIMARY KEY (CustomerID));
```

- IF NOT EXISTS can be excluded but you will then get an error message if you run the code twice without dropping the table, i.e., DROP TABLE Customer;
- Many flavors of SQL (but not postgres) have a command that combines DROP and CREATE:
CREATE OR REPLACE TABLE Customer (...

DDL – CREATE – NAMING CONVENTIONS AND QUOTES

```
CREATE TABLE IF NOT EXISTS Customer (  
    CustomerID SERIAL,  
    CustomerName TEXT NOT NULL,  
    CreditLimit NUMERIC CHECK(CreditLimit>0),  
    PaymentDays INTEGER CHECK(PaymentDays>0) NOT NULL DEFAULT 30,  
    PRIMARY KEY (CustomerID));
```

- Convention is to write key words in upper case and object names (identifiers) in lower case (often using PascalCase or under scores).
 - CREATE, TABLE, INTEGER, SELECT, etc.
 - Customer, CustomerID, CustomerName, etc.
- Key words and identifiers must begin with a letter (a-z) or an underscore (_). Subsequent characters can additionally contain digits (in postgres also dollar signs).

DDL – CREATE – NAMING CONVENTIONS AND QUOTES

```
CREATE TABLE IF NOT EXISTS Customer (  
    CustomerID SERIAL,  
    CustomerName TEXT NOT NULL,  
    CreditLimit NUMERIC CHECK(CreditLimit>0),  
    PaymentDays INTEGER CHECK(PaymentDays>0) NOT NULL DEFAULT 30,  
    PRIMARY KEY (CustomerID));
```

- Key words and (unquoted) identifiers are, however, case insensitive.
 - Unquoted identifiers are always folded to lower case
 - in the CREATE TABLE code we set the name to Customer, but the table is stored as customer.
 - The following two create statements are the same:

```
CREATE TABLE Customer  
CREATE TABLE customer
```
- String comparisons are case-sensitive when using common comparison operators, e.g., =, IN, and LIKE.
 - String literals are indicated using single quotes.

DDL – CREATE – NAMING CONVENTIONS AND QUOTES

- Double quotes are used for quoted identifiers. Postgres does not fold quoted identifiers, i.e., they are case-sensitive.

```
CREATE TABLE Customers (  
    id SERIAL NOT NULL,  
    email TEXT NOT NULL);
```

```
CREATE TABLE "Suppliers"(  
    id SERIAL NOT NULL,  
    email TEXT NOT NULL);
```

Lower case identifier	SELECT * FROM customers;	SELECT * FROM suppliers;
PascalCase identifier	SELECT * FROM Customers;	SELECT * FROM Suppliers;
PascalCase Quoted Identifier	SELECT * FROM "Customers";	SELECT * FROM "Suppliers";

DDL – CREATE – DATA TYPES

Data Types:

```
CREATE TABLE IF NOT EXISTS Customer (  
    CustomerID SERIAL,  
    CustomerName TEXT NOT NULL,  
    CreditLimit NUMERIC CHECK(CreditLimit>0),  
    PaymentDays INTEGER CHECK(PaymentDays>0) NOT NULL DEFAULT 30,  
    PRIMARY KEY (CustomerID));
```

DDL – CREATE – DATA TYPES

Strings

CHAR(n)	fixed-length, blank padded
VARCHAR(n)	variable-length with length limit
TEXT, VARCHAR	variable unlimited length

There is no performance difference, but CHAR and VARCHAR can specify length limit, which functions as a constraint.

Numbers

SMALLINT	integers between -32,768 to 32,767.
INT	integers between -2,147,483,648 to 2,147,483,647.
BIGINT	integers between -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
SERIAL	autoincrementing integer 1 to 2,147,483,647.
DECIMAL, NUMERIC	user-specified precision, exact up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
DOUBLE, REAL	inexact floating-point with 15 and 6, respectively, decimal digits precision

Because floating-point numbers are stored as an integer multiplied by another integer power of 2 they are often unable to exactly represent the number that should be stored. Therefore, DECIMAL and NUMERIC are recommended for storing monetary amounts and other quantities where exactness is required. However, these data types are very slow compared to DOUBLE and REAL.

Date/Time

DATE	is used to store the dates only using 'YYYY-MM-DD'
TIME	is used to stores the time of day values using 'HH:MM:SS'
TIMESTAMP	is used to stores both date and time values, e.g., '2022-08-19 17:23:54'
TIMESTAMPTZ	is used to store a timezone-aware timestamp data type, e.g., '2022-08-19 17:23:54 EST'
INTERVAL	is used to store periods of time, e.g., '100 days' or '22 hours'

Arrays

In PostgreSQL, an array column can be used to store an array of strings, integers, dates, etc.

DDL – CREATE – CONSTRAINTS

```
CREATE TABLE IF NOT EXISTS Customer (  
    CustomerID SERIAL,  
    CustomerName TEXT NOT NULL,  
    CreditLimit NUMERIC CHECK(CreditLimit>0),  
    PaymentDays INTEGER CHECK(PaymentDays>0) NOT NULL DEFAULT 30,  
    PRIMARY KEY (CustomerID));
```

We will take a look at:

- CHECK
- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY

DDL – CREATE – CONSTRAINTS – **CHECK**

- A **CHECK** constraint allows you to specify that values in a certain column must satisfy a Boolean (truth-value) expression. The constraint definition comes after the data type.

```
CreditLimit NUMERIC CHECK(CreditLimit>0)
```

- You can also give the constraint a separate name. This clarifies error messages and makes it easier to refer to the constraint when you need to change it. The syntax is:

```
CreditLimit NUMERIC CONSTRAINT positive_credit_limit CHECK(CreditLimit>0)
```

- Check constraints can also be written as table constraints (which also allows the constraint to reference multiple columns). Table level check constraints can also be named:

```
CREATE TABLE IF NOT EXISTS Customer (  
    .  
    .  
    CHECK (CreditLimit>0),  
    CONSTRAINT valid_payment_days CHECK (PaymentDays>0));
```

DDL – CREATE – CONSTRAINTS – **NOT NULL AND UNIQUE**

```
CREATE TABLE IF NOT EXISTS Customer (  
    CustomerID SERIAL,  
    CustomerName TEXT NOT NULL,  
    CreditLimit NUMERIC CHECK(CreditLimit>0),  
    PaymentDays INTEGER CHECK(PaymentDays>0) NOT NULL DEFAULT 30,  
    PRIMARY KEY (CustomerID));
```

- The **NOT NULL** constraint prevents the insertion of NULL values in a column.
- NOT NULL constraints are always written as column constraints.
- A **UNIQUE** constraint ensure that values contained in a column or a group of columns (then needs to be added as a table constraint) are unique among all the rows in the table (NULL is allowed in multiple rows). The syntax is:
 CustomerID INTEGER **UNIQUE**,

DDL – CREATE – CONSTRAINTS – **PRIMARY KEY**

```
CREATE TABLE IF NOT EXISTS Customer (  
    CustomerID SERIAL,  
    CustomerName TEXT NOT NULL,  
    CreditLimit NUMERIC CHECK(CreditLimit>0),  
    PaymentDays INTEGER CHECK(PaymentDays>0) NOT NULL DEFAULT 30,  
    PRIMARY KEY (CustomerID));
```

- A **PRIMARY KEY** constraint indicates that a column, or group of columns, uniquely identifies each row in the table. This requires that the values be both unique and not null.
 - If a single column then it can be either a column or table constraint,
 - If a composite primary key, i.e., a key that spans multiple columns, then it has to be a table constraint: **PRIMARY KEY** (col_1, col_2).
- **UNIQUE** and **PRIMARY KEY** constraints automatically create b-tree indexes.

DDL – CREATE – CONSTRAINTS – FOREIGN KEY

```
CREATE TABLE OrderHeader (  
    OrderID SERIAL,  
    BackOrderID INTEGER,  
    OrderDate DATE NOT NULL,  
    CustomerID INTEGER NOT NULL,  
    SalesPerson TEXT,  
    PRIMARY KEY (OrderID),  
    FOREIGN KEY (BackOrderID) REFERENCES OrderHeader(OrderID) ON  
        UPDATE CASCADE ON DELETE SET NULL,  
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID) ON  
        UPDATE CASCADE ON DELETE RESTRICT);
```

- A **FOREIGN KEY** constraint specifies that values in a column (or a group of columns) must match values in a column in another table. This maintains the referential integrity between two related tables.
- In this example, the two constraints prevent the addition of rows to the OrderHeader table with
 - a (non-NULL) BackOrderID that does not match an OrderID in the OrderHeader table, and
 - a CustomerID that does not appear in the Customer table.

DDL – CREATE – CONSTRAINTS

```
CREATE TABLE OrderHeader (  
    .  
    .  
    FOREIGN KEY (BackOrderID) REFERENCES OrderHeader(OrderID) ON  
    UPDATE CASCADE ON DELETE SET NULL,  
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID) ON  
    UPDATE CASCADE ON DELETE RESTRICT);
```

- ON UPDATE and ON DELETE constraints define what actions to take to referencing rows if a referenced row is updated or deleted.
- For example, the second foreign key constraint above specifies what happens to customer orders, i.e., rows in the OrderHeader, that reference a CustomerID in the Customer table that is updated or deleted.
- If a CustomerID is updated in the Customers table then ON UPDATE CASCADE specifies that the corresponding value(s) in the OrderHeader table should be changed to the new CustomerID value in the referenced table.

* referenced row is the row that the foreign key is pointing at (Customer rows in the second constraint),
referencing row is the row with the foreign key (OrderHeader rows in our example)

DDL – CREATE – CONSTRAINTS

```
CREATE TABLE OrderHeader (
```

```
    .
```

```
    .
```

```
    FOREIGN KEY (BackOrderID) REFERENCES OrderHeader(OrderID) ON  
    UPDATE CASCADE ON DELETE SET NULL,
```

```
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID) ON  
    UPDATE CASCADE ON DELETE RESTRICT);
```

Options for ON UPDATE and ON DELETE:

- RESTRICT - prevents referenced parent row to be updated/deleted.
- NO ACTION (default) - if any referencing rows still exist when the constraint is checked, an error is raised.
- CASCADE - when a referenced row is updated/deleted, referencing row(s) are automatically updated/deleted as well.
- SET NULL or SET DEFAULT - when a referenced row is updated/deleted, referencing row(s) are set to NULL or their default values (which is NULL if not defined).

DDL – CREATE – INDEX

A database index is a data structure that can improve the speed of data retrieval, but they are costly to create and maintain.

Different RDBMS have different (but often very similar) types and options for indexes.

Most RDBMS (but not postgres) implement **non-clustered** and **clustered** indexes:

Non-Clustered Indexes

- are similar to an index in the back of a book where the index is a separate sorted structure that provides pointers (page numbers) to the location of content in the book.
- non-clustered indexes exist separately from a unsorted table (unordered heap). The index contains pointers to table rows in the heap (TID in postgres; TID stands for Tuple Identifiers).

Clustered Indexes

- are similar to a phonebook where the actual book content is organized
- clustered indexes affect the way the table is stored with the table data being organized in index order typically using a B-tree structure.
- when the table is modified, the table has to be re-sorted

DDL – CREATE – INDEX – **B-TREE AND HASH INDEXES**

- Postgres has two primary non-clustered indexes: B-tree and Hash.
- B-tree indexes are the default:

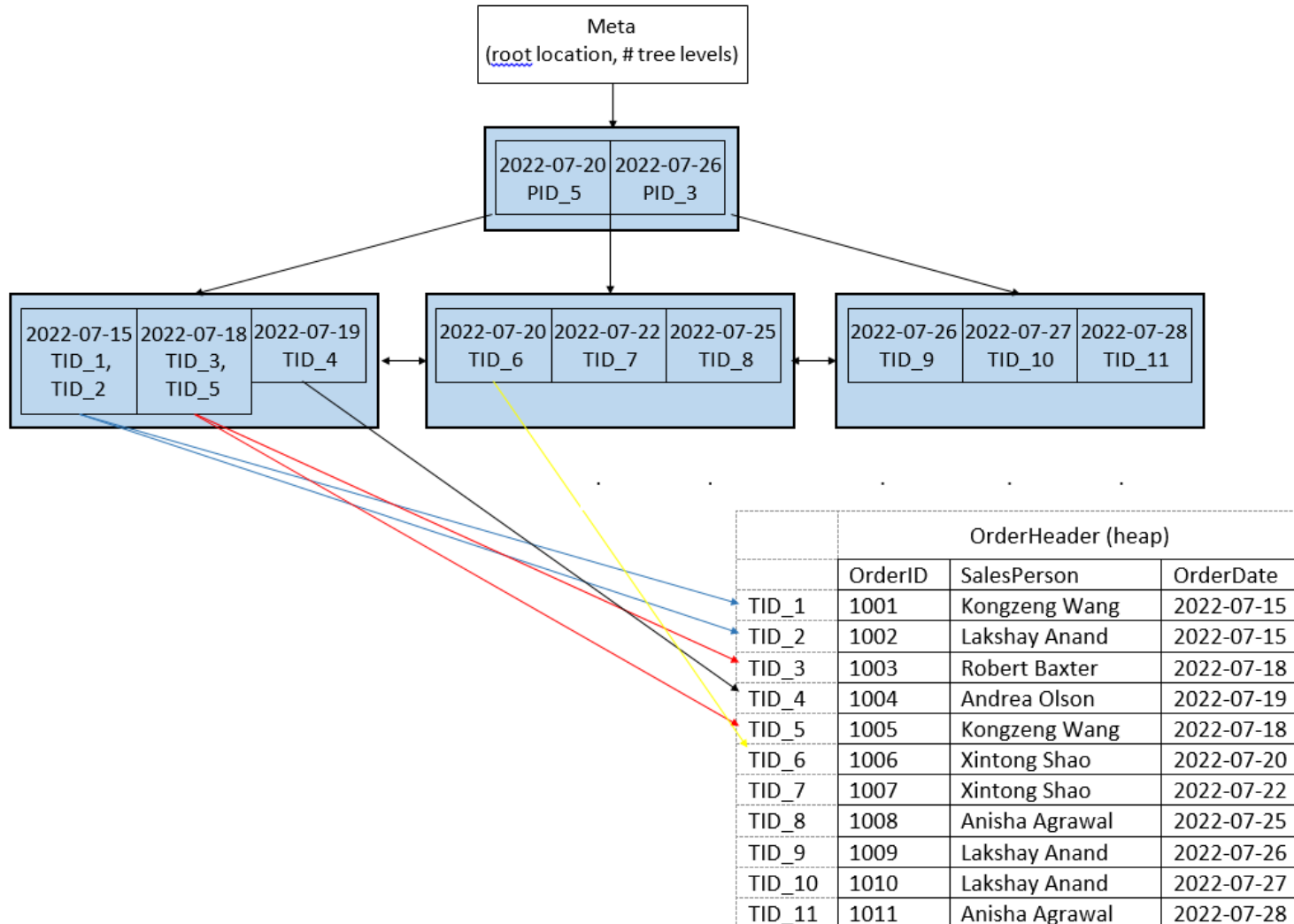
```
CREATE INDEX OrderID_idx ON OrderHeader (OrderID);
```

- Creates a B-tree index on the OrderID column in the OrderHeader table
 - This create statement is redundant as PRIMARY KEY (and UNIQUE) constraints automatically creates B-Tree indexes (and we earlier assigned a primary key constraint to OrderID).
- Other index types are selected by writing the keyword USING followed by the index type name (this example creates a Hash Index):

```
CREATE INDEX CustomerID_idx ON OrderHeader USING  
HASH (CustomerID);
```

POSTGRES **B-TREE INDEX** (WITH DEDUPLICATION)

B-Tree Index on OrderDate in the OrderHeader table:



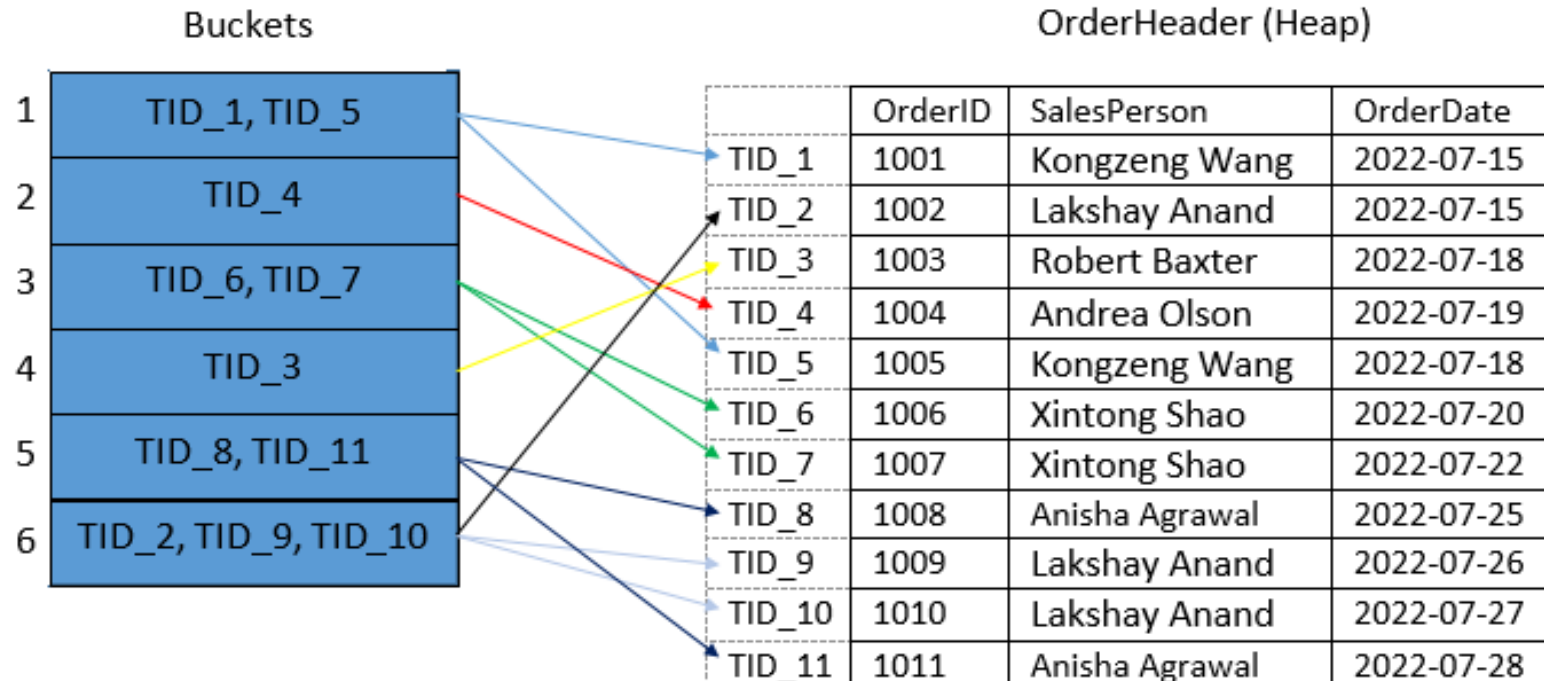
POSTGRES HASH INDEX

Hash Index on OrderHeader.SalesPerson.

The Hash Index uses hash function $f()$, which returns hash values as shown to the right.

For example, $f(\text{Kongzeng Wang}) = 1$.

x	$f(x)$
Kongzeng Wang	1
Lakshay Anand	6
Robert Baxter	4
Andrea Olson	2
Anisha Agrawal	5
Xintong Shao	3



DDL – CREATE – INDEX

- B-trees indexes are often useful when an indexed column is involved in a comparison using:

<	<=	=	>=	>
---	----	---	----	---

 - one of the following operators: constructs equivalent to combinations of these operators, such as BETWEEN, IN, and ON
 - IS NULL and IS NOT NULL,
 - pattern matching operator LIKE when the pattern is a constant and is anchored to the beginning of the string, for example, LIKE 'foo%'
- B-tree indexes can sometimes also be useful for retrieving data in sorted order.
- Hash indexes are often useful when an indexed column is involved in a comparison using the equal operator:

=

DDL – CREATE – INDEX – INDEX-ONLY SCANS

- **Index-only scans** answer queries from an index alone without any heap access (which makes them fast).
- Index-only scans require a **covering index**, which is an index that includes all the columns needed by a query, i.e., the query references only columns stored in the index.
- For example, assume we are writing a query that only uses OrderID and BackOrderID (the query does not reference any other columns in any part of the query), we can then create a covering index as follows:

```
CREATE INDEX OrderIDBackOrderID_idx ON OrderHeader (OrderID, BackOrderID);
```

- This creates a B-tree sorted first by OrderID and then by BackOrderID (since OrderID is UNIQUE, the index is not sorted by BackOrderID at all).
- Since queries typically need to retrieve more columns than just the ones they search on, postgres allows the inclusion of extra columns that are not part of the index key. This is done by adding an INCLUDE clause listing the extra columns:

```
CREATE INDEX BackOrderID_idx ON OrderHeader (BackOrderID) INCLUDE (OrderDate);
```


DDL – DROP AND ALTER

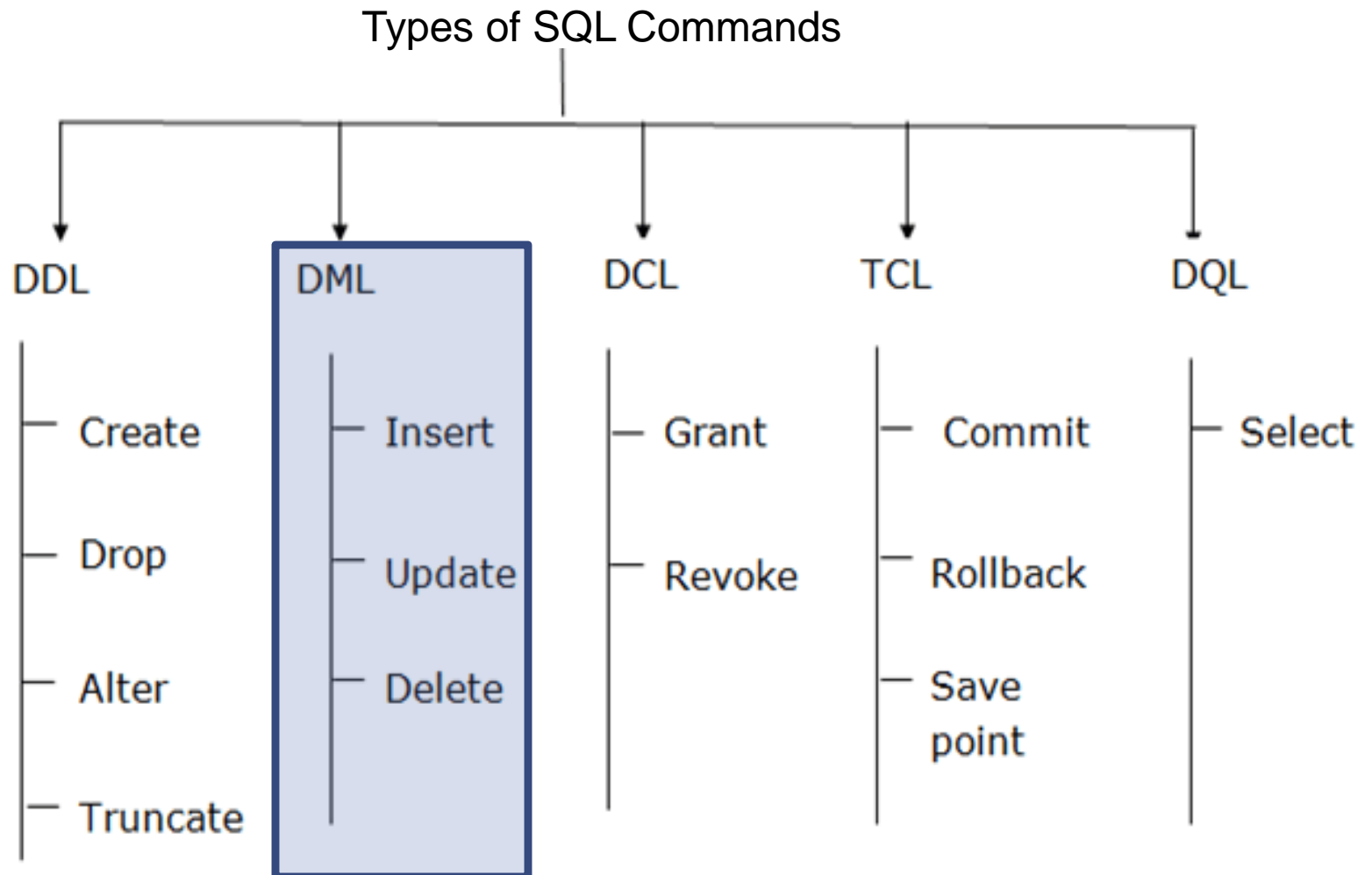
Common DDL Clauses:

- **CREATE**
- The **DROP** command is used to remove database objects:
`DROP TABLE Customer;`
- The **ALTER** command is used to make changes to database objects. Many different actions including ADD, e.g., ADD column and ADD table constraint, and DROP, e.g., DROP column and DROP constraint can be used:

```
ALTER TABLE Customer  
  ADD COLUMN Email TEXT,  
  DROP COLUMN PaymentDays,  
  ADD CONSTRAINT UniqueConstraint UNIQUE (CustomerName);
```

SQL DML STATEMENTS

DATA MANIPULATION LANGUAGE (DML)



DML - INSERT

- To add data to tables use INSERT INTO:

```
INSERT INTO OrderHeader (BackOrderID, OrderDate, CustomerID,  
SalesPerson) VALUES  
    (NULL, '2022-07-15', 201, 5),  
    (DEFAULT, '2022-07-18', 205, 8),  
    (1, '2022-07-18', 201, 5);
```

- Notice that OrderID was not included in the column list. When a column is excluded from the list or when DEFAULT is used as a value, then the default value will be used (if a default value is not explicitly defined for the column then the default is NULL).

DML - INSERT

- It is also possible to insert the result of a SELECT query:

```
INSERT INTO OrderHeader (BackOrderID, OrderDate, CustomerID, SalesPerson)
    SELECT BackOrderID, OrderDate, CustomerID, SalesPerson
    FROM OrderHeader;
```

- When inserting a lot of data into a table from file, considering using the more efficient postgres COPY FROM command:

```
COPY OrderHeader2 FROM '/Users/Public/Orders.csv'
    WITH NULL AS 'NULL' CSV HEADER QUOTE AS '"';
```

- (COPY TO can be used to copy data from a table or query to a file)

DML – UPDATE AND DELETE

- The **UPDATE** command is used to change values in tables. If the WHERE clause is omitted then all rows in the table are updated, otherwise only the rows that match the WHERE condition are updated:

```
UPDATE OrderHeader SET CustomerID = 202 WHERE OrderID = 2;
```

- The **DELETE** command is used to delete rows in tables. If WHERE is excluded then all rows will be deleted:

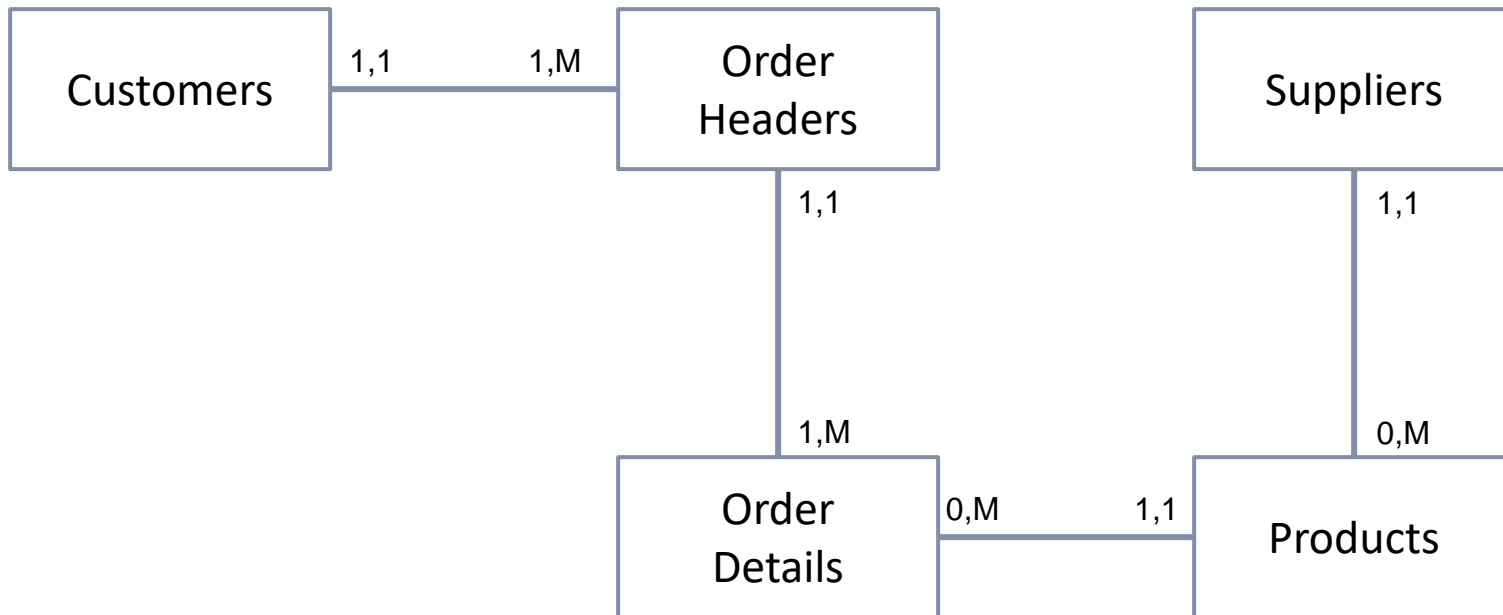
```
DELETE FROM OrderHeader WHERE OrderID = 2;
```

SELECT STATEMENTS OVERVIEW

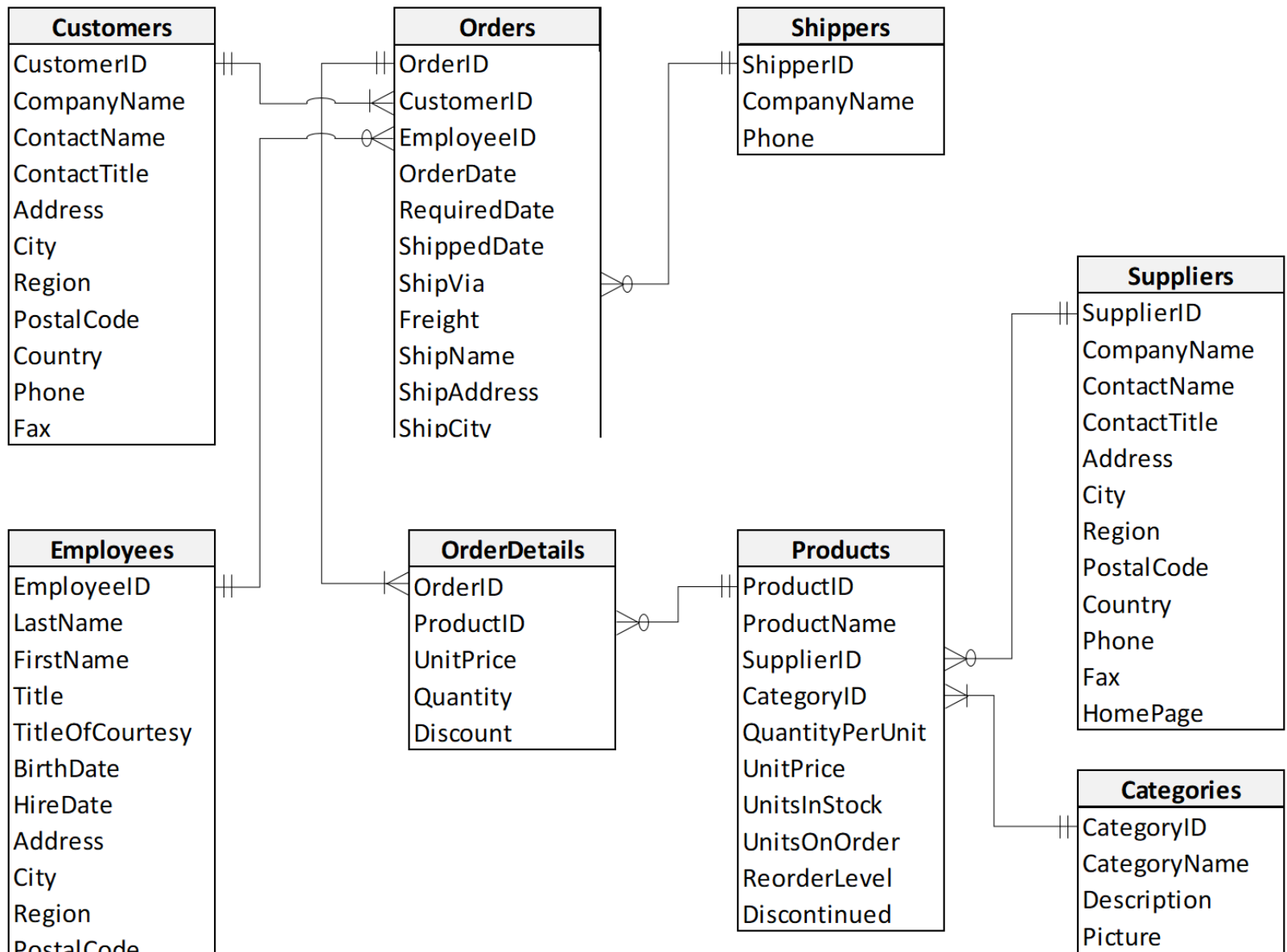
BUT FIRST A QUICK INTRODUCTION TO ENTITY
RELATIONSHIP DIAGRAMS

ENTITY RELATIONSHIP DIAGRAMS (ERD)

Basic structure of ERD diagrams



ERD FOR NORTHWIND



SELECT STATEMENTS OVERVIEW

SELECT STATEMENT

- In SQL, the **SELECT** statement is used to query or retrieve data from one or more tables in the database. The returned data is known as a result-set.

Common **SELECT** statement clauses:

- **SELECT** - the columns (and expressions) to be returned
- **FROM** - the table(s), view(s), or sub-query(ies) (also known as inner queries or nested queries) from which data will be obtained
- **WHERE** - the conditions under which a row will be included in the result
- **GROUP BY** - the column(s) to use to group rows into sets of summary rows based on values in the column
- **HAVING** - the conditions under which a group will be included
- **ORDER BY** - sorts the result according to specified criteria

EXAMPLE...

GROUP BY

- GROUP BY ProductID (I ordered this table by ProductID):

The diagram illustrates a GROUP BY operation on a table ordered by ProductID. The main table is divided into three groups based on ProductID: Product 3 (yellow), Product 4 (blue), and Product 5 (red). Brackets on the left side of the main table group the rows by ProductID. Arrows on the right side map these groups to a smaller summary table. The summary table has columns: orderid, productid, unitprice, and quantity. The rows in the summary table correspond to the groups: Product 3 (yellow), Product 4 (blue), and Product 5 (red). The unitprice and quantity columns in the summary table are marked with question marks, indicating they are aggregated values.

orderid	productid	unitprice	quantity
10289	3	8	30
10405	3	8	50
10485	3	8	20
10540	3	10	60
10591	3	10	14
10702	3	10	6
10742	3	10	20
10764	3	10	20
10849	3	10	49
10857	3	10	30
11017	3	10	25
11077	3	10	4
10309	4	17.6	20
10326	4	17.6	24
10336	4	17.6	18
10339	4	17.6	10
10344	4	17.6	35
10464	4	17.6	16
10511	4	22	50
10527	4	22	50
10533	4	22	50
10606	4	22	20
10635	4	22	10
10636	4	22	25
10654	4	22	12
10704	4	22	6
10726	4	22	25
10846	4	22	21
10913	4	22	30
10950	4	22	5
11000	4	22	25
11077	4	22	1
10258	5	17	65
10262	5	17	12
10290	5	17	20
10382	5	17	32
10635	5	21.35	15
10708	5	21.35	4
10848	5	21.35	30
10958	5	21.35	20
11030	5	21.35	70
11047	5	21.35	30

orderid	productid	unitprice	quantity
?	3	?	?
?	4	?	?
?	5	?	?

GROUP BY

- GROUP BY ProductID:

The diagram illustrates a GROUP BY operation on a table of orders. The main table is divided into three groups based on ProductID: Product 3 (yellow), Product 4 (blue), and Product 5 (red). Arrows from each group point to a summary table that contains question marks for the values to be calculated.

orderid	productid	unitprice	quantity
10289	3	8	30
10405	3	8	50
10485	3	8	20
10540	3	10	60
10591	3	10	14
10702	3	10	6
10742	3	10	20
10764	3	10	20
10849	3	10	49
10857	3	10	30
11017	3	10	25
11077	3	10	4
10309	4	17.6	20
10326	4	17.6	24
10336	4	17.6	18
10339	4	17.6	10
10344	4	17.6	35
10464	4	17.6	16
10511	4	22	50
10527	4	22	50
10533	4	22	50
10606	4	22	20
10635	4	22	10
10636	4	22	25
10654	4	22	12
10704	4	22	6
10726	4	22	25
10846	4	22	21
10913	4	22	30
10950	4	22	5
11000	4	22	25
11077	4	22	1
10258	5	17	65
10262	5	17	12
10290	5	17	20
10382	5	17	32
10635	5	21.35	15
10708	5	21.35	4
10848	5	21.35	30
10958	5	21.35	20
11030	5	21.35	70
11047	5	21.35	30

orderid	productid	unitprice	quantity
?	3	?	?
?	4	?	?
?	5	?	?

- What to do with ?
- exclude column
- use aggregate function:
 - AVG, COUNT, MAX, MIN, SUM
 - For example: AVG(UnitPrice)

ADDITIONAL FROM CLAUSE

- **JOIN ... ON** - used to merge two or more tables
 - **JOIN** is typically either
 - INNER JOIN (or simply JOIN) - returns records that have matching values in both tables
 - LEFT OUTER JOIN (or simply LEFT JOIN) - returns all records from the left table, and the matched records from the right table (unmatched records in the right table are not returned)

```
SELECT *  
FROM table_1_name [LEFT] JOIN table_2_name
```

- **ON** defines the columns to JOIN based on (rows in the joined tables are matched based on values in the columns defined in the ON statement)

```
SELECT *  
FROM table_1_name [LEFT] JOIN table_2_name  
ON table_1.column_name = table_2.column_name
```

- Older syntax using a list of tables and a where condition should be avoided:

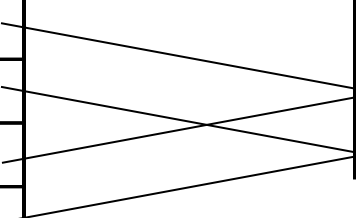
```
SELECT *  
FROM Table_1, Table_2  
WHERE Table_1.column_name = Table_2.column_name
```

It is not as readable (according to most people), it might produce different results when used with LEFT OUTER JOINS, and it might produce less efficient execution plans depending on the database.

EXAMPLES

In the following examples, let's assume we have the following two tables, where CustomerID can be used to join the two table:

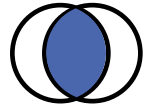
Orders		Customers	
OrderID	CustomerID	CustomerID	CustomerName
10252	11	10	Hanari Carnes
10253	12	11	QUICK-Stop
10254	11	12	Simons bistro
10255	12		
10256	13		



ADDITIONAL FROM CLAUSE

Example of [INNER] JOIN ... ON

returns records that have matching values in both tables



Orders		Customers	
OrderID	CustomerID	CustomerID	CustomerName
10252	11	10	Hanari Carnes
10253	12	11	QUICK-Stop
10254	11	12	Simons bistro
10255	12		
10256	13		

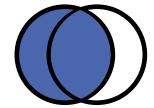


```
SELECT A.OrderID, A.CustomerID, B.CustomerID, B.CustomerName
FROM Orders AS A INNER JOIN Customers AS B
ON A.CustomerID = B.CustomerID
```



OrderID	CustomerID	CustomerID	CustomerName
10252	11	11	QUICK-Stop
10253	12	12	Simons bistro
10254	11	11	QUICK-Stop
10255	12	12	Simons bistro

ADDITIONAL FROM CLAUSE



Example of LEFT [OUTER] JOIN ... ON

returns all records (matched and unmatched) from the left table, and the matched records from the right table (unmatched right records are not returned)

Orders		Customers	
OrderID	CustomerID	CustomerID	CustomerName
10252	11	10	Hanari Carnes
10253	12	11	QUICK-Stop
10254	11	12	Simons bistro
10255	12		
10256	13		



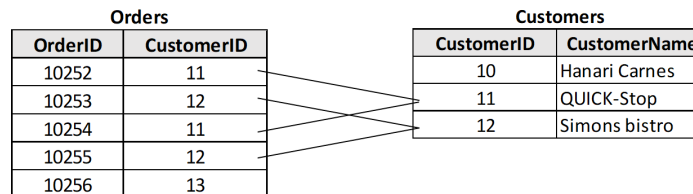
```
SELECT A.OrderID, A.CustomerID, B.CustomerID, B.CustomerName
FROM Orders AS A LEFT OUTER JOIN Customers AS B
ON A.CustomerID = B.CustomerID
```



OrderID	CustomerID	CustomerID	CustomerName
10252	11	11	QUICK-Stop
10253	12	12	Simons bistro
10254	11	11	QUICK-Stop
10255	12	12	Simons bistro
10256	13	NULL	NULL

ADDITIONAL JOIN TYPES

RIGHT [OUTER] JOIN ... ON:	Returns all records from the right table, and the matched records from the left table (unmatched records in the left table are not returned).
FULL [OUTER] JOIN ... ON:	Returns all records that have matching values in both tables and all the unmatched records from both the left and the right tables (returns all matched or unmatched rows from the tables on both
[CROSS] JOIN (no ON):	Does not perform a match and instead matches all left table records with all the right table records.



RIGHT OUTER JOIN ... ON

OrderID	CustomerID	CustomerID	CustomerName
NULL	NULL	10	Hanari Carnes
10252	11	11	QUICK-Stop
10253	11	11	QUICK-Stop
10254	12	12	Simons bistro
10255	12	12	Simons bistro

CROSS JOIN (no ON)

OrderID	CustomerID	CustomerID	CustomerName
10252	11	10	Hanari Carnes
10252	11	11	QUICK-Stop
10252	11	12	Simons bistro
10253	12	10	Hanari Carnes
10253	12	11	QUICK-Stop
10253	12	12	Simons bistro
10254	11	10	Hanari Carnes
10254	11	11	QUICK-Stop
10254	11	12	Simons bistro
10255	12	10	Hanari Carnes
10255	12	11	QUICK-Stop
10255	12	12	Simons bistro
10256	13	10	Hanari Carnes
10256	13	11	QUICK-Stop
10256	13	12	Simons bistro

FULL OUTER JOIN ... ON

OrderID	CustomerID	CustomerID	CustomerName
10252	11	11	QUICK-Stop
10253	12	12	Simons bistro
10254	11	11	QUICK-Stop
10255	12	12	Simons bistro
10256	13	NULL	NULL
NULL	NULL	10	Hanari Carnes

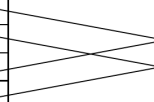
ADDITIONAL JOIN TYPES

- We will next take a look at some additional joins. These do not all have specific SQL clauses, but are referred to by names (you might hear one of these names in an interview)
 - NATURAL JOIN
 - SELF JOIN
 - EQUI and NON EQUI JOINS
 - SEMI-JOIN
 - ANTI-JOIN
 - CROSS-DATABASE JOIN

NAMED SPECIALIZED JOINS

NATURAL JOIN (no ON): Same as an INNER JOIN with an ON statement that includes all columns in the two tables with the same names. Natural joins merge two tables by their common column names without the column names having to be specified in an ON clause. Can be INNER, RIGHT, or LEFT.

Orders		Customers	
OrderID	CustomerID	CustomerID	CustomerName
10252	11	10	Hanari Carnes
10253	12	11	QUICK-Stop
10254	11	12	Simons bistro
10255	12		
10256	13		



```
SELECT A.OrderID, A.CustomerID, B.CustomerID, B.CustomerName
FROM Orders A
NATURAL INNER JOIN Customers B
```



OrderID	CustomerID	CustomerID	CustomerName
10252	11	11	QUICK-Stop
10253	12	12	Simons bistro
10254	11	11	QUICK-Stop
10255	12	12	Simons bistro

NAMED SPECIALIZED JOINS

SELF JOIN (table_1 a JOIN table_1 b) :	A self join is a join in which a table is joined with itself. The self join merges two copies of the same table together (there are no actual copies of the table but SELF JOIN functions as if there were). SELF JOIN can be INNER, RIGHT, LEFT, or CROSS.
EQUI and NON EQUI JOINS	The comparison operator in the ON clause typically is an equal(=) sign, but can also be >, <, >=, <=, and <>. When using an equal sign the join is an EQUI JOIN, when another comparison operator is used then the join is a NON EQUI JOIN.

Using the Orders table, for each customer and order, find the most recent previous order (do not show the first order placed by a customer). Assume higher order IDs represent newer orders.

Orders	
OrderID	CustomerID
10252	11
10253	12
10254	11
10255	12
10256	13

SELECT A.CustomerID, A.OrderID, max(B.OrderID) AS CustomerPreviousOrder
FROM **Orders** AS A JOIN **Orders** AS B
ON **A.CustomerID = B.CustomerID AND A.OrderID > B.OrderID**
GROUP BY A.CustomerID, A.OrderID

CustomerID	OrderID	CustomerPreviousOrder
11	10254	10252
12	10255	10253

Let's simplify by examining the result set before the NON-EQUI JOIN and GROUP BY.

SELF-JOIN AND EQUI-JOIN

Orders		Orders	
OrderID	CustomerID	OrderID	CustomerID
10252	11	10252	11
10253	12	10253	12
10254	11	10254	11
10255	12	10255	12
10256	13	10256	13



```
SELECT A.CustomerID, A.OrderID, max(B.OrderID) AS CustomerPreviousOrder, B.CustomerID
FROM Orders AS A JOIN Orders AS B
ON A.CustomerID = B.CustomerID AND A.OrderID > B.OrderID
GROUP BY A.CustomerID, A.OrderID
```



Result without the
NON-EQUI join and
GROUP BY statements
(and selecting all columns):

A.CustomerID	A.OrderID	B.OrderID	B.CustomerID
11	10252	10252	11
11	10252	10254	11
12	10253	10253	12
12	10253	10255	12
11	10254	10252	11
11	10254	10254	11
12	10255	10253	12
12	10255	10255	12
13	10256	10256	13

NON-EQUI JOIN

```
SELECT A.CustomerID, A.OrderID, max(B.OrderID) AS CustomerPreviousOrder, B.CustomerID
FROM Orders AS A JOIN Orders AS B
ON A.CustomerID = B.CustomerID AND A.OrderID > B.OrderID
GROUP BY A.CustomerID, A.OrderID
```

- If we now also apply the NON-EQUI join ($A.OrderID > B.OrderID$), we will only keep rows where $A.OrderID$ is greater than $B.OrderID$.
- For this small data set, the GROUP BY is not needed as there are no orders with more than one previous order id.

A.CustomerID	A.OrderID	B.OrderID	B.CustomerID
11	10252	10252	11
11	10252	10254	11
12	10253	10253	12
12	10253	10255	12
11	10254	10252	11
11	10254	10254	11
12	10255	10253	12
12	10255	10255	12
13	10256	10256	13



CustomerID	OrderID	CustomerPreviousOrder
11	10254	10252
12	10255	10253

NAMED SPECIALIZED JOINS

SEMI-JOIN Regular joins return all matches between two tables, which means that if a record in the first table matches multiple records in the second table then the record from the first table will be repeated for each match. If you instead simply want to show all records from the first table that have at least one match in the second table, but you do not want to duplicate the records from the first table then a SEMI-JOIN can be useful. Regular SQL, however, does not have a SEMI-JOIN clause. IN or EXISTS can instead be used (a regular LEFT JOIN where the right table is replaced by a subquery that uses DISTINCT, can also be used; note that this is different than performing a regular join and then using DISTINCT on the result, which is typically slower).

Return all customers in the Customers table that have at least one order (do not repeat the same customer id for each order the customer has placed).

Customers

CustomerID	CustomerName
10	Hanari Carnes
11	QUICK-Stop
12	Simons bistro

Orders

OrderID	CustomerID
10252	11
10253	12
10254	11
10255	12
10256	13



```
SELECT A.CustomerID, A.CustomerName
FROM Customers AS A INNER JOIN
    (SELECT DISTINCT CustomerID FROM Orders) AS B
ON A.CustomerID = B.CustomerID;
```



CustomerID	CustomerName
11	QUICK-Stop
12	Simons bistro

NAMED SPECIALIZED JOINS

OrderID	CustomerID
10252	11
10253	12
10254	11
10255	12
10256	13



Step 1.

```
SELECT A.CustomerID, A.CustomerName  
FROM Customers AS A INNER JOIN  
      (SELECT DISTINCT CustomerID FROM Orders) AS B  
ON A.CustomerID = B.CustomerID
```



CustomerID
11
12
13

The sub-query returns a result set with unique CustomerID values from the Orders table.

CustomerID	CustomerName
10	Hanari Carnes
11	QUICK-Stop
12	Simons bistro

Step 2.

CustomerID
11
12
13



```
SELECT A.CustomerID, A.CustomerName  
FROM Customers AS A INNER JOIN  
      (SELECT DISTINCT CustomerID FROM Orders) AS B  
ON A.CustomerID = B.CustomerID
```



The sub-query result set is joined with the Customers table based on CustomerID using an INNER JOIN.

CustomerID	CustomerName
11	QUICK-Stop
12	Simons bistro

NAMED SPECIALIZED JOINS

ANTI-JOIN ANTI-JOIN is essentially the opposite of a SEMI-JOIN; ANTI-JOIN returns one copy of each row in the first table for which no match is found. This is done by using a LEFT JOIN together with a WHERE condition that only returns rows from the right table where the right table column used in the ON statement IS NULL. It is also possible to use NOT EXISTS. However, NOT IN should be avoided if the second table contains NULL values in the matching column (easier to always avoid it for ANTI-JOIN).

Find all customers in the Customers table that do not have any orders.

Customers	
CustomerID	CustomerName
10	Hanari Carnes
11	QUICK-Stop
12	Simons bistro

Orders	
OrderID	CustomerID
10252	11
10253	12
10254	11
10255	12
10256	13



```
SELECT *  
FROM Customers AS A LEFT JOIN Orders AS B  
ON A.CustomerID = B.CustomerID  
WHERE B.CustomerID IS NULL;
```



CustomerID	CustomerName	OrderID	CustomerID
10	Hanari Carnes	NULL	NULL

Let's take a look at this step-by-step.

NAMED SPECIALIZED JOINS

Customers		Orders	
CustomerID	CustomerName	OrderID	CustomerID
10	Hanari Carnes	10252	11
11	QUICK-Stop	10253	12
12	Simons bistro	10254	11
		10255	12
		10256	13

Step 1.

```
SELECT *  
  FROM Customers AS A LEFT JOIN Orders AS B  
    ON A.CustomerID = B.CustomerID  
 WHERE B.CustomerID IS NULL;
```

Without the WHERE condition, the LEFT JOIN returns all the matched records and all the unmatched records from the left table.

CustomerID	CustomerName	OrderID	CustomerID
10	Hanari Carnes	NULL	NULL
11	QUICK-Stop	10252	11
11	QUICK-Stop	10254	11
12	Simons bistro	10253	12
12	Simons bistro	10255	12

Step 2.

```
SELECT *  
  FROM Customers AS A LEFT JOIN Orders AS B  
    ON A.CustomerID = B.CustomerID  
 WHERE B.CustomerID IS NULL;
```

The WHERE B.CustomerID IS NULL then selects only the rows with NULL values in B.CustomerID, i.e., the first row from Step 1.

CustomerID	CustomerName	OrderID	CustomerID
10	Hanari Carnes	NULL	NULL