

# CA218

# Supplementary SQL Notes

Martin Crane

# SQL SELECT

- Used to extract data from a database
- This is a table called "Persons" from the Northwind database

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- The command **SELECT \* FROM Persons** will select all the records/columns in the "Persons" table, and store them in a results table called the Result-Set
- Keep in Mind That...SQL is not case sensitive

# SQL **SELECT** Syntax

- The command **SELECT column\_name(s) FROM table\_name** is used to extract certain columns from a table
- Example: **SELECT LastName,FirstName FROM Persons**

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- The result-set is:

LastName	FirstName
Hansen	Ola
Svendson	Tove
Pettersen	Kari

- The asterisk (\*) is a quick way of selecting all columns!

# SQL SELECT DISTINCT Statement

- In a table, some columns may have duplicate values. This is no problem but often only want to list different/distinct values in table.
- The **DISTINCT** keyword is used to return only distinct/different values.
- SQL SELECT DISTINCT Syntax:
  - `SELECT DISTINCT column_name(s)  
FROM table_name`
- Example: Now we want to select only the distinct values from the column named "City" from the "Persons" table.
  - `SELECT DISTINCT City FROM Persons`

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- The result-set is:

Sandnes
Stavanger

# SQL WHERE Clause

- The **WHERE** clause is used to extract only those records that fulfil a specified criterion.
- SQL **WHERE** Syntax:
  - `SELECT column_name(s)`  
`FROM table_name`  
`WHERE column_name operator value`
- Ex: Select only persons living in city "Sandnes" from table "Persons".
  - `SELECT * FROM Persons`  
`WHERE City='Sandnes'`

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- Result-set is:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes

# SQL WHERE Clause (cont'd)

SQL DML COMMANDS

SQL SELECT.

## Quotes Around Text Fields & Operators

- SQL uses single quotes around text values (most db systems also accept double quotes). Numeric values should not be enclosed in quotes.
- Example: Now we want to select only the persons living in the city "Sandnes" from the table "Persons" whose first name is "Tove"
  - `SELECT * FROM Persons WHERE FirstName='Tove'`
- Example: Want to select only entries from the table "Born" where year of birth is 1965 (note absence of quotes around year)
  - `SELECT * FROM Born WHERE Year=1965`
- With the WHERE clause, the following operators can be used:
  - = (equals), <> (not equal to),
  - > (greater than), < (less than), >= (GT or equal), <= (LT or equal)
  - **BETWEEN** (Between an inclusive range)
  - **LIKE** (Search for a pattern)
  - **IN** (If know exact value want to return for at least one of the columns)

# SQL AND & OR Operators

- **AND** operator displays a record if both 1st **&** 2nd condition are true.
- **OR** operator displays a record if either 1st **or** 2nd condition are true.
- "Persons" Table

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- Ex. Select only persons with first name "Tove" & last name "Svendson"
  - **SELECT \* FROM Persons**  
**WHERE FirstName='Tove'**  
**AND LastName='Svendson'**
- Result-set is:

P_Id	LastName	FirstName	Address	City
2	Svendson	Tove	Borgvn 23	Sandnes

# SQL AND & OR Operators (cont'd)

- "Persons" Table

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- Ex. Select only persons with first name "Tove" OR first name "Ola":

```
- SELECT * FROM Persons  
WHERE FirstName='Tove'  
OR FirstName='Ola'
```

- Result-set is:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes

# SQL AND & OR Operators (cont'd)

- Can also combine AND & OR (use brackets to form complex expressions).

- "Persons" Table

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- Ex. Select only persons with last name "Svendson" & first name "Tove" OR "Ola":

```
- SELECT * FROM Persons WHERE  
  LastName='Svendson'  
  AND (FirstName='Tove' OR FirstName='Ola')
```

- Result-set is:

P_Id	LastName	FirstName	Address	City
2	Svendson	Tove	Borgvn 23	Sandnes

## SQL DML COMMANDS

### SQL SELECT.

# SQL ORDER BY Keyword

- **ORDER BY** keyword used to sort the result-set by a specified column.
- Sorts records in *ascending* order by default. If want *descending* order, you can use the **DESC** keyword.

- SQL **ORDER BY** Syntax:

```
- SELECT column_name(s)  
  FROM table_name  
  ORDER BY column_name(s) ASC|DESC
```

- "Persons" Table

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- Ex. Select all persons from table above but sort persons by last name.

```
- SELECT * FROM Persons  
  ORDER BY LastName
```

- Result-set is:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
2	Svendson	Tove	Borgvn 23	Sandnes

# SQL SELECT TOP Statement

- **TOP** clause is used to specify the number of records to return.
- Can be useful on large tables with '000s of records as Returning a large number of records can impact on performance.
- SQL **TOP** Syntax:

```
- SELECT TOP number|percent column_name(s)  
FROM table_name
```

- "Persons" Table

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- Ex. want to select only the two first records in the table above.

```
- SELECT TOP 2 * FROM Persons
```

- Result-set is:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes

- Can also use `SELECT TOP 50 PERCENT * FROM Persons`

## SQL DML COMMANDS

### SQL SELECT.

# SQL LIKE Operator

- The LIKE operator is used to search for a specified pattern in a column..
- SQL LIKE Syntax:

```
- SELECT column_name(s)
  FROM table_name
  WHERE column_name LIKE pattern
```

- "Persons" Table

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- Ex. Want to select persons living in city ending with "s" from "Persons" table. (% can be used to define wildcards (missing letters in pattern))

```
- SELECT * FROM Persons
  WHERE City LIKE '%s'
```

- Result-set is:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes

- Note: Can also have `WHERE column_name NOT LIKE pattern`

# SQL **LIKE** (cont'd): Wildcards

- Wildcards can substitute for 1/more characters when searching for data in a database. Must be used with the SQL LIKE operator.
- With SQL, the following wildcards can be used:
  - % A substitute for zero or more characters
  - \_ A substitute for exactly one character
  - [charlist] Any single character in charlist
  - [!charlist] Any single character not in charlist
- "Persons" Table as before
- Ex. Want to select persons with a last name that do not start with "b" or "s" or "p" from the "Persons" table.
  - `SELECT * FROM Persons`  
`WHERE LastName LIKE '[!bsp]%'`

- Result-set is:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes

# SQL IN Operator

- The **IN** operator allows you to specify multiple values in a WHERE clause.
- SQL **IN** Syntax:

```
- SELECT column_name(s)  
  FROM table_name  
  WHERE column_name IN (value1,value2,...)
```

- "Persons" Table

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- Ex. Want to select persons with last name "Hansen" or "Pettersen" .

```
- SELECT * FROM Persons  
  WHERE LastName IN ('Hansen','Pettersen')
```

- Result-set is:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- The **IN** Operator can also be used for *Nested* Queries, where a **SELECT** query is nested inside a **SELECT**, **UPDATE**, **INSERT**, or **DELETE** SQL query.

# SQL **IN** Operator (cont'd): Nested Queries

- Here is a simple example of SQL nested query:

```
- SELECT Model FROM Product
   WHERE ManufacturerID IN
       (SELECT ManufacturerID FROM Manufacturer
        WHERE Manufacturer = 'Dell')
```

- The nested query above will select all models from the "Product" table manufactured by Dell:
- More Nested Query examples to follow later...

## SQL DML COMMANDS

### SQL SELECT.

# SQL BETWEEN Operator

- **BETWEEN** operator is used in a **WHERE** clause to select a range of data between two values.

- SQL **BETWEEN** Syntax:

```
- SELECT column_name(s)
  FROM table_name
  WHERE column_name
  BETWEEN value1 AND value2
```

- "Persons" Table

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- Ex. Select persons with last name alphabet'ly btw "Hansen" & "Pettersen"

```
- SELECT * FROM Persons
  WHERE LastName
  BETWEEN 'Hansen' AND 'Pettersen'
```

- Result-set is:

- Most times works in range [value1,value2] but variations apply!

# SQL ALIAS

- With `SQL`, an alias name can be given to a table or to a column. Can be a good thing if have very long or complex table names or column names.
- SQL **ALIAS** Syntax for tables:

```
- SELECT column_name(s)
   FROM table_name
   AS alias_name
```

- SQL **ALIAS** Syntax for Columns:

```
- SELECT column_name AS alias_name
   FROM table_name
```

- Ex. Given a table "Persons" & another "Product\_Orders". We will give the table aliases of "p" an "po" respectively. Want to list all orders that "Ola Hansen" is responsible for.

```
SELECT po.OrderID, p.LastName, p.FirstName
FROM Persons AS p,
Product_Orders AS po
WHERE p.LastName='Hansen' AND p.FirstName='Ola'
```

- The same `SELECT` statement without aliases:

```
SELECT Product_Orders.OrderID, Persons.LastName, Persons.FirstName
FROM Persons,
Product_Orders
WHERE Persons.LastName='Hansen' AND Persons.FirstName='Ola'
```

# SQL JOIN

- `JOIN` is used in an SQL statement to query data from two or more tables, based on a relationship between certain columns in these tables.
- Tables in a database are often related to each other with keys.

- "Persons" Table

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- Note that the "P\_Id" column is the primary key in the "Persons" table.
- Next, we have the "Orders" table:

Order_Id	OrderNo	P_Id
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	15

- Note that "Order\_Id" column is the PK in "Orders" & "P\_Id" column refers to persons in "Persons" table without using their names.
- Notice that the r'ship between the two tables is the "P\_Id" column.

# SQL JOIN (cont'd)

- Different flavours of SQL JOIN exist:
  - **INNER JOIN**: Return rows when there is at least one match in both tables
  - **LEFT JOIN**: Return all rows from the left table, even if there are no matches in the right table
  - **RIGHT JOIN**: Return all rows from the right table, even if there are no matches in the left table
  - **FULL JOIN**: Return rows when there is a match in one of the tables

## SQL DML COMMANDS

### SQL SELECT.

# SQL (**INNER**)JOIN Keyword

- **INNER JOIN** keyword returns rows when there is at least one match in both tables.
- SQL **INNER JOIN** Syntax:
  - **SELECT** column\_name(s)  
**FROM** table\_name1  
**INNER JOIN** table\_name2  
**ON** table\_name1.column\_name=table\_name2.column\_name

- "Persons" Table & "Orders" Table

P_Id	LName	FName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Order_Id	OrderNo	P_Id
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	15

- Want to list all persons with any orders:
  - **SELECT** Persons.LastName, Persons.FirstName, Orders.OrderNo  
**FROM** Persons  
**INNER JOIN** Orders  
**ON** Persons.P\_Id=Orders.P\_Id  
**ORDER BY** Persons.LastName

# SQL (INNER) JOIN (cont'd)

- Results set is:

LName	FName	OrderNo
Hansen	Ola	22456
Hansen	Ola	24562
Pettersen	Kari	44678
Pettersen	Kari	77895

- The **INNER JOIN** keyword returns rows when there is at least one match in both tables. If there are rows in "Persons" that do not have matches in "Orders", those rows will NOT be listed.
- The above could also be written using **WHERE** as:
  - `SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo  
FROM Persons  
WHERE Persons.P_Id=Orders.P_Id  
ORDER BY Persons.LastName`
- The above can also be written (eg in ORACLE) with **USING** as:
  - `SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo  
FROM Persons  
INNER JOIN Orders  
USING(P_Id)  
ORDER BY Persons.LastName`
- NOTE: In SQL, **JOIN** ALWAYS means equijoin NOT natural join so there are duplicate rows produced (of **P\_Id** in this case)

# SQL UNION Operator

- **UNION** operator is used to combine result-sets of two or more **SELECT** statements.
- Notice that each **SELECT** statement within the **UNION** must have the same number of columns. Columns must also have similar data types. Also, the columns in each **SELECT** statement must be in the same order.
- The **UNION** operator selects only distinct values by default. To allow duplicate values, use **UNION ALL**.
- SQL **UNION** Syntax:
  - **SELECT column\_name(s) FROM table\_name1**  
**UNION**  
**SELECT column\_name(s) FROM table\_name2**
- "Employees\_Norway" Table & "Employees\_USA" Table

E_id	E_Name
1	Hansen, Ola
2	Svendson, Tove
3	Svendson, Stephen
4	Pettersen, Kari

E_id	E_Name
1	Turner, Sally
2	Kent, Clark
3	Svendson, Stephen
4	Scott, Stephen

# SQL UNION Operator (cont'd)

- Now we want to list all the different employees in Norway and USA.
- We use the following SELECT statement:

```
- SELECT E_Name FROM Employees_Norway  
UNION  
SELECT E_Name FROM Employees_USA
```

- Results Set is:

E_Name
Hansen, Ola
Svendson, Tove
Svendson, Stephen
Pettersen, Kari
Turner, Sally
Kent, Clark
Scott, Stephen

# SQL Aggregate Functions

- SQL has many built-in functions for performing calculations on data.
- Aggregate functions return a single value, calculated from values in a column:
  - `AVG(column_name)` - Returns average value in a column
  - `COUNT(column_name)` - Returns number of (non-null) rows in a column
  - `FIRST(column_name)` - Returns first value in a column
  - `LAST(column_name)` - Returns last value in a column
  - `MAX(column_name)` - Returns largest value in a column
  - `MIN(column_name)` - Returns smallest value in a column
  - `SUM(column_name)` - Returns sum of all values in a column
- `COUNT(*)` - Returns number of (null & non-null) rows in a table
- Will see examples of these later in the notes.

# SQL GROUP BY Clause

- **GROUP BY** statement is used in conjunction with the aggregate functions to group the result-set by one or more columns (or valid expression).
- Need to **GROUP BY** all the other selected columns, i.e., all columns except the one(s) operated on by the arithmetic operator.
- SQL **GROUP BY** Syntax:
  - `SELECT column_name1[,column_name2],aggregate_function(column_name1)`  
`FROM table_name`  
`WHERE column_name operator value`  
`GROUP BY column_name1 [, column_name2...]`
- SQL **GROUP BY** Example:
- Have the following "Orders" table:

O_Id	OrderDate	OrderPrice	Customer
1	2008/11/12	1000	Hansen
2	2008/10/23	1600	Nilsen
3	2008/09/02	700	Hansen
4	2008/09/03	300	Hansen
5	2008/08/30	2000	Jensen
6	2008/10/04	100	Nilsen

- Now we want to find the total sum (total order) of each customer.
- We will have to use the **GROUP BY** statement to group the customers.

# SQL GROUP BY Clause (cont'd)

- We use the following SQL statement:
  - `SELECT Customer, SUM(OrderPrice) FROM Orders  
GROUP BY Customer`

- The result-set will look like this:

Customer	Sum(OrderPrice)
Hansen	2000
Nilsen	1700
Jensen	2000

- What happens if we omit the GROUP BY statement? ie:

- `SELECT Customer, SUM(OrderPrice) FROM Orders`

- The result-set will look like this:

Customer	Sum(OrderPrice)
Hansen	5700
Nilsen	5700
Hansen	5700
Hansen	5700
Jensen	5700
Nilsen	5700

- Why? The SELECT statement has 2 columns specified (Customer and SUM(OrderPrice)). The "SUM(OrderPrice)" returns a single value (total sum of "OrderPrice" column), while "Customer" returns 6 values (1 value for each row in "Orders" table). This will not give us the correct result.

# SQL HAVING

- The **HAVING** clause was added to SQL because the **WHERE** keyword could not be used with aggregate functions.
- Also, **HAVING** is used in conjunction with the **SELECT** clause to specify a search condition for a group. The **HAVING** clause behaves like the **WHERE** clause, but is applicable to groups.
- SQL **HAVING** Syntax:

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
HAVING aggregate_function(column_name) operator value
```

- Example: want to find if any of the customers have a total order of less than 2000. Use the following SQL statement:
  - ```
SELECT Customer,SUM(OrderPrice) FROM Orders
GROUP BY Customer
HAVING SUM(OrderPrice)<2000
```
- Example: want to find if the customers "Hansen" or "Jensen" have a total order of more than 1500. Use the following SQL statement:
  - ```
SELECT Customer,SUM(OrderPrice) FROM Orders
WHERE Customer='Hansen' OR Customer='Jensen'
GROUP BY Customer
HAVING SUM(OrderPrice)>1500
```

# SQL INSERT INTO Statement

- The **INSERT INTO** statement is used to insert new records in a table
- SQL **INSERT INTO** Syntax:
  - `INSERT INTO table_name (column1, column2, column3,...)  
VALUES (value1, value2, value3,...)`
- Example: Have the empty "Persons" table:

P_Id	LastName	FirstName	Address	City
------	----------	-----------	---------	------

- Want to insert a new row in the "Persons" table.
  - `INSERT INTO Persons (P_Id, LastName, FirstName, Address, City)  
VALUES (4, 'Nilsen', 'Johan', 'Bakken 2', 'Stavanger')`
- "Persons" table now looks like:

P_Id	LastName	FirstName	Address	City
4	Nilsen	Johan	Bakken 2	Stavanger

# SQL UPDATE Statement

- The **UPDATE** statement is used to update existing records in a table.
- SQL **UPDATE** Syntax:
  - **UPDATE** `table_name`  
**SET** `column1=value, column2=value2,...`  
**WHERE** `some_column=some_value`
- **Note:** **WHERE** clause in the **UPDATE** syntax! It specifies which records should be updated. If you omit the **WHERE** clause, all records will be updated!
- Example of **UPDATE** statement: Have the "Persons" table:

P_Id	LName	FName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove		

- Now we want to update the person "Svendson, Tove" in the "Persons" table
  - **UPDATE** `Persons`  
**SET** `Address='Nissestien 67', City='Sandnes'`  
**WHERE** `LastName='Svendson' AND FirstName='Tove'`

- "Persons" table now looks like:

P_Id	LName	FName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Nissestien 67	Sandnes

# SQL DELETE Statement

- The **DELETE** statement is used to delete records in a table.

- SQL **DELETE** Syntax:

- **DELETE FROM table\_name**  
**WHERE some\_column=some\_value**

- Example: Have the "Persons" table:

P_Id	LName	FName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Nissestien 67	Sandnes

- Want to delete the person "Svendson, Tove" in "Persons" table:

- **DELETE FROM Persons**  
**WHERE LastName='Svendson' AND FirstName='Tove'**

- "Persons" table now looks like:

P_Id	LName	FName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes

- If want to delete everything from "Persons" table:

- **DELETE \* FROM Persons**

Or

- **DELETE FROM Persons**

# SQL CREATE DATABASE Statement

- The **CREATE DATABASE** statement is used to create a database.
- SQL **CREATE DATABASE** Syntax:
  - **CREATE DATABASE database\_name**
- Example: Want to create a database called "my\_db":
  - **CREATE DATABASE my\_db**
- Database tables can be added with the **CREATE TABLE** statement.

# SQL CREATE TABLE Statement

- The **CREATE TABLE** statement is used to create a table in a database
- SQL **CREATE TABLE** Syntax:
  - **CREATE TABLE** `table_name`  
(  
  `column_name1 data_type [column-constraints]`,  
  `column_name2 data_type [column-constraints]`,  
  `[table_constraint {tableconstraint}]`,  
  `[table_constraint {tableconstraint}]`)
- **column-constraints/table\_constraint**: an optional list of constraints with no separators (more later when we talk about constraints)
- Example: Want to create a table "Persons" with 5 columns: P\_Id, LastName, FirstName, Address, and City.

```
– CREATE TABLE Persons  
(  
  P_Id int,  
  LastName varchar(255),  
  FirstName varchar(255),  
  Address varchar(255),  
  City varchar(255)  
)
```

- The empty "Persons" table will now look like this:

P_Id	LastName	FirstName	Address	City
------	----------	-----------	---------	------

- The table can be populated using the **INSERT INTO** Statement

# SQL CREATE INDEX Statement

- The `CREATE INDEX` statement is used to create indexes in tables.
- Indexes allow the database application to find data fast; without reading the whole table.
- SQL `CREATE INDEX` Syntax (duplicate values allowed):
  - `CREATE INDEX index_name  
ON table_name (column_name)`
- SQL `CREATE UNIQUE INDEX` Syntax (duplicate values NOT allowed):
  - `CREATE UNIQUE INDEX index_name  
ON table_name (column_name)`
- **Note:** Updating a table with indexes takes more time than updating a table without. So should only create indexes on columns (and tables) that will be frequently searched against.

# SQL DROP Statement

- Indexes, tables, and databases can easily be deleted/removed with the **DROP** statement.
- The **DROP INDEX** statement is used to delete an index in a table.
  - **DROP INDEX index\_name ON table\_name**
- The **DROP TABLE** statement is used to delete a table.
  - **DROP TABLE table\_name**
- The **DROP DATABASE** statement is used to delete a DB.
  - **DROP DATABASE database\_name**

# SQL ALTER TABLE Statement

- The **ALTER TABLE** statement is used to add, delete, or modify columns in an existing table.
- To add a column in a table, use the following syntax:
  - **ALTER TABLE table\_name**  
**ADD column\_name datatype**
- To delete a column in a table, use the following syntax
  - **ALTER TABLE table\_name**  
**DROP COLUMN column\_name**
- To change the data type of a column in a table, use the following syntax:
  - **ALTER TABLE table\_name**  
**ALTER COLUMN column\_name datatype**

# SQL Table Constraints

- Constraints are used to limit the type of data that can go into a table.
- Constraints can be specified when a table is created (with the **CREATE TABLE** statement) or after the table is created (with the **ALTER TABLE** statement, about which more below).
- We will focus on the following constraints:
  - **NOT NULL**
  - **UNIQUE**
  - **PRIMARY KEY**
  - **FOREIGN KEY**
  - **CHECK**
  - **DEFAULT**

# SQL NOT NULL Constraint

- The **NOT NULL** constraint enforces a column to NOT accept **NULL** values.
- The **NOT NULL** constraint enforces a field to always contain a value. This means that you cannot insert a new record, or update a record without adding a value to this field.
- Following SQL enforces the "P\_Id" & "LastName" columns to not accept NULL values:
  - ```
CREATE TABLE Persons
(
  P_Id int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Address varchar(255),
  City varchar(255)
)
```

# SQL **UNIQUE** Constraint

- The **UNIQUE** constraint uniquely identifies each record in a database table.
- The **UNIQUE** and **PRIMARY KEY** constraints both provide a guarantee for uniqueness for a column or set of columns.
- A **PRIMARY KEY** constraint automatically has a **UNIQUE** constraint def'd on it.
- Note: can have many **UNIQUE** constraints per table, but only one **PRIMARY KEY** constraint per table.
- The following SQL creates a **UNIQUE** constraint on "P\_Id" column when "Persons" table is created:

```
- CREATE TABLE Persons
(
  P_Id int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Address varchar(255),
  City varchar(255),
  UNIQUE(P_Id)
)
```

# SQL PRIMARY KEY Constraint

- The **PRIMARY KEY** constraint uniquely ids each record in a database table.
- Primary keys must contain unique values.
- A primary key column cannot contain NULL values.
- Each table should have a primary key, & each table can have only one primary key.
- The following SQL creates a PRIMARY KEY on the "P\_Id" column when "Persons" table is created:

```
- CREATE TABLE Persons
(
  P_Id int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Address varchar(255),
  City varchar(255),
  PRIMARY KEY(P_Id)
)
```

## SQL DDL COMMANDS

### SQL Constraints.

# SQL PRIMARY KEY Constraint (cont'd)

- To allow naming of a PRIMARY KEY constraint, and for defining a **PRIMARY KEY** constraint on multiple columns, use the following SQL syntax:

```
- CREATE TABLE Persons
(
  P_Id int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Address varchar(255),
  City varchar(255),
  CONSTRAINT pk_PersonID PRIMARY KEY (P_Id,LastName)
)
```

# SQL FOREIGN KEY Constraint

- A FOREIGN KEY in one table points to a PRIMARY KEY in another table. Primary keys must contain unique values.
- Suppose we have two tables "Persons" table & "Orders" table and that "P\_Id" column in the "Orders" table points to the "P\_Id" column in the "Persons" table.
- Then "P\_Id" column in the "Persons" table is the PRIMARY KEY in the "Persons" table and "P\_Id" column in the "Orders" table is a FOREIGN KEY in the "Orders" table .
- The following SQL creates a FOREIGN KEY on the "P\_Id" column when the "Orders" table is created:
  - ```
CREATE TABLE Orders
(
  O_Id int NOT NULL,
  OrderNo int NOT NULL,
  P_Id int,
  PRIMARY KEY (O_Id),
  FOREIGN KEY (P_Id) REFERENCES Persons(P_Id)
)
```
- The **FOREIGN KEY** constraint also prevents invalid data being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

# SQL CHECK Constraint

- **CHECK** constraint used to limit value range that can be placed in a column.
- If **CHECK** constraint def'd on a single column allows only certain values for this column.
- If **CHECK** constraint def'd on a table it can limit the values in certain columns based on values in other columns in the row.
- Each table should have a primary key, & each table can have only one primary key.
- Following SQL creates a CHECK constraint on the "P\_Id" column when the "Persons" table is created to specify that column "P\_Id" must only include integers greater than 0:

```
- CREATE TABLE Persons
(
  P_Id int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Address varchar(255),
  City varchar(255),
  CHECK (P_Id>0)
)
```

## SQL DDL COMMANDS

### SQL Constraints.

# SQL CHECK Constraint (cont'd)

- To allow naming of a **CHECK** constraint, and for defining a **CHECK** constraint on multiple columns, use the following SQL syntax:

```
- CREATE TABLE Persons
(
  P_Id int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Address varchar(255),
  City varchar(255),
  CONSTRAINT chk_Person CHECK (P_Id>0 AND City='Sandnes')
)
```

# SQL DEFAULT Constraint

- **DEFAULT** constraint is used to insert a default value into a column
- Value will be added to all new records, if no other value is specified.
- Following SQL creates a **DEFAULT** constraint on the "City" column when the "Persons" table is created:

```
- CREATE TABLE Persons
(
  P_Id int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Address varchar(255),
  City varchar(255) DEFAULT 'Sandnes'
)
```

# SQL Referential Trigger Actions

- Referential Integrity means that for each row in a referencing table, its foreign key must match an existing primary key in the referenced table.
- It can be violated when tuples are inserted/deleted or when a foreign key attribute value is modified.
- With SQL2 the DBA can specify the action to be taken if a referential integrity constraint is violated by attaching a *referential trigger action clause* to any FK constraint.
- The options include **SET NULL**, **CASCADE**, and **SET DEFAULT**. An option must be qualified with either **ON DELETE** or **ON UPDATE**.
- Recall the CREATE TABLE syntax for creating a table S:
  - **CREATE TABLE S**  
(  
  **dno INT NOT NULL DEFAULT 1,**  
  **CONSTRAINT deptcons FOREIGN KEY(dno) REFERENCES departments(dnum),**  
  **ON DELETE SET DEFAULT ON UPDATE CASCADE)**
- **CASCADE** Whenever rows in the referenced table are deleted or updated, the respective rows of the child (referencing) table with a matching foreign key column will get deleted or updated as well.
- **SET NULL/DEFAULT** FK values in referencing row are set to **NULL/DEFAULT** when referenced row is updated or deleted. **SET NULL** only possible if the respective columns in the referencing table are nullable.