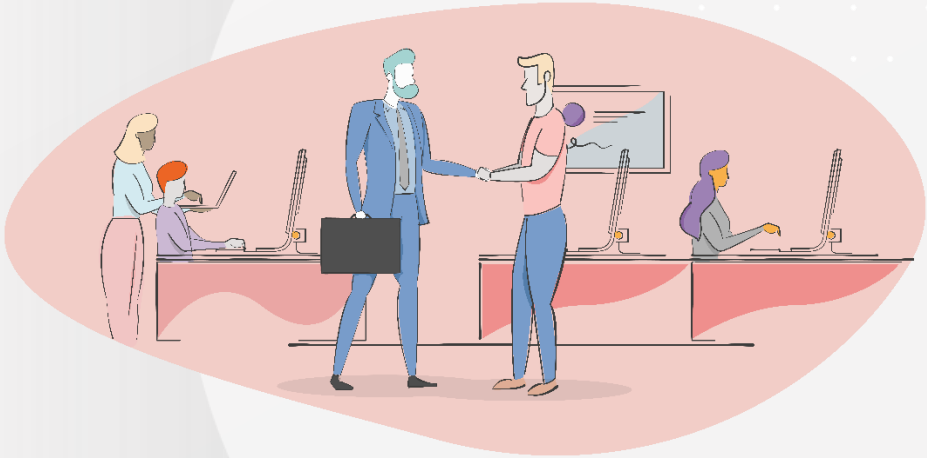
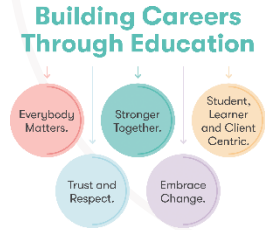
Advanced SQL



L5 Data Engineer Higher Apprenticeship
Module 2 / 12 (“**Databases and Data Lakes**”)
Topic 3 / 5

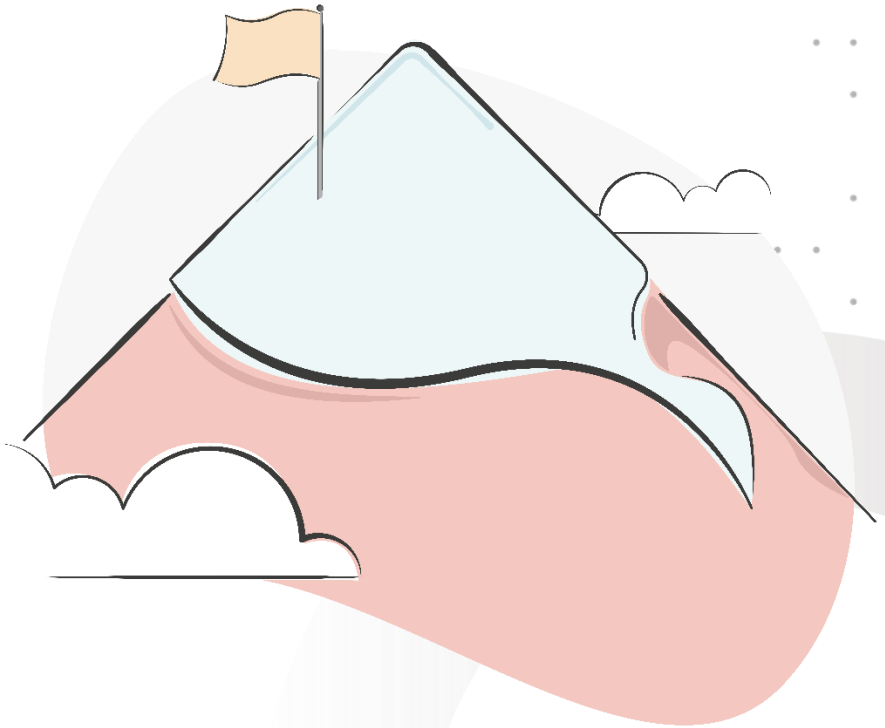
rev. 1 (2024)

Learning objectives



By the end of today's webinar, you will be able to:

- **Apply** standard industry tools and best practices for designing, maintaining and optimising the performance of databases, data lakes, warehouses
- **Explain** the JOIN concept in SQL and its usefulness.
- **Evaluate** how views, nested queries and aggregation can fulfill advanced business and user requirements for data engineering.



BPP

Section Agenda

What you will learn about in this section:

1. Types of SQL JOINS
2. Subquery
3. SQL practice

BPP

Types of SQL JOIN

EQUI JOIN

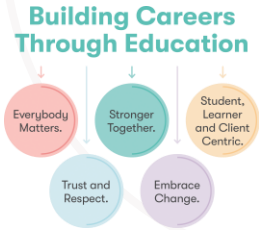
- EQUI JOIN is a simple SQL join.
- Uses the equal sign(=) as the comparison operator for the condition

NON EQUI JOIN

- NON EQUI JOIN uses comparison operator other than the equal sign.
- The operators uses like >, <, >=, <= with the condition.

BPP

Types of SQL EQUI JOIN

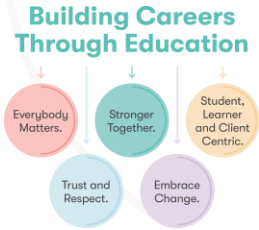


INNER JOIN

- Returns only matched rows from the participating tables.
- Match happened only at the key record of participating tables

BPP

INNER JOIN



The INNER JOIN selects all rows from both participating tables as long as there is a match between the columns.

An SQL INNER JOIN is same as JOIN clause, combining rows from two or more tables.

BPP

Example: INNER JOIN

table_A

A	M
1	m
2	n
4	o

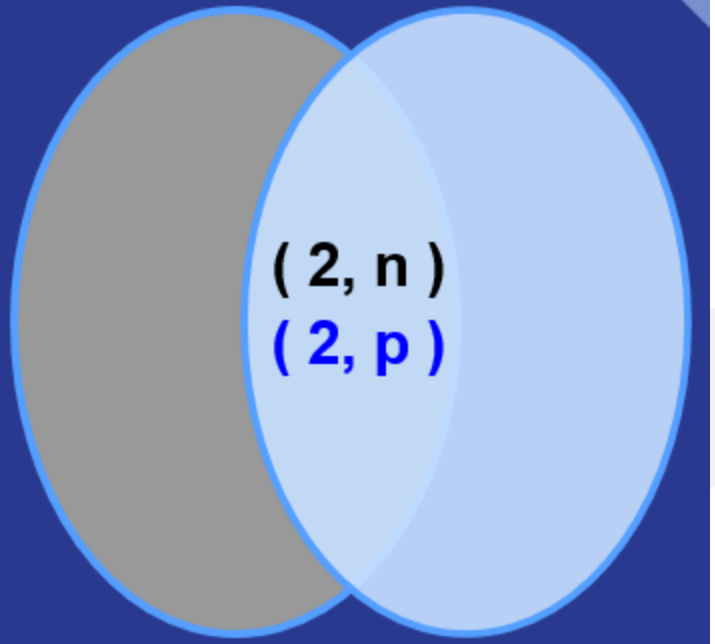
table_B

A	N
2	p
3	q
5	r

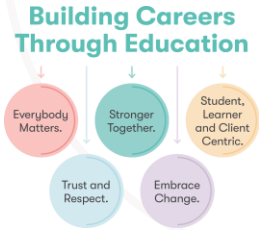
```
SELECT * FROM table_A
INNER JOIN table_B
ON table_A.A=table_B.A;
```

A	M	A	N
2	n	2	p

Output



LEFT JOIN or LEFT OUTER JOIN



- The SQL LEFT JOIN, joins two tables and fetches rows based on a condition, which are matching in both the tables.

- The unmatched rows will also be available from the RIGHT table before the JOIN clause.

BPP

Example: LEFT JOIN or LEFT OUTER JOIN

A	M
1	m
2	n
4	o

table_A

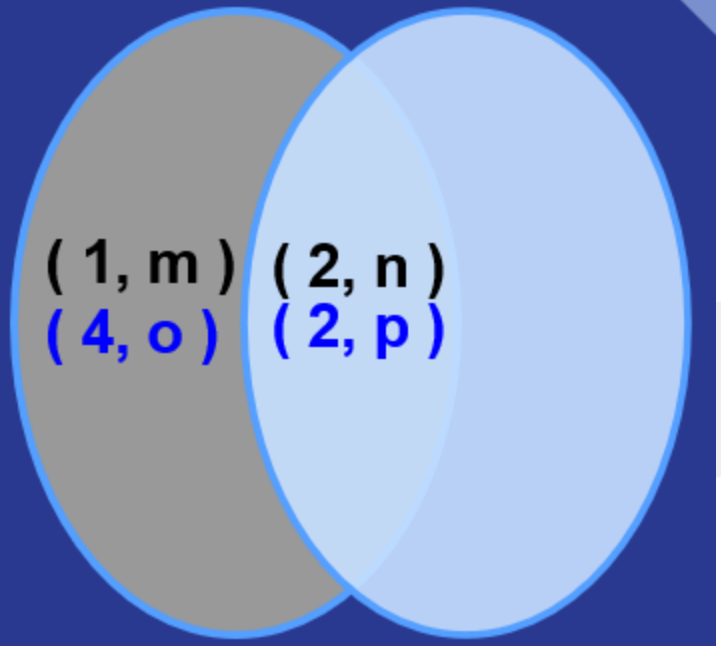
A	N
2	p
3	q
5	r

table_B

```
SELECT * FROM table_A
LEFT JOIN table_B
ON table_A.A=table_B.A;
```

A	M	A	N
2	n	2	p
1	m	null	null
4	o	null	null

Output



RIGHT JOIN or RIGHT OUTER JOIN

- The SQL RIGHT JOIN, joins two tables and fetches rows based on a condition, which are matching in both the tables.

- The unmatched rows will also be available from the LEFTtable written after the JOIN clause.

BPP

Example: RIGHT JOIN or RIGHT OUTER JOIN

A	M
1	m
2	n
4	o

table_A

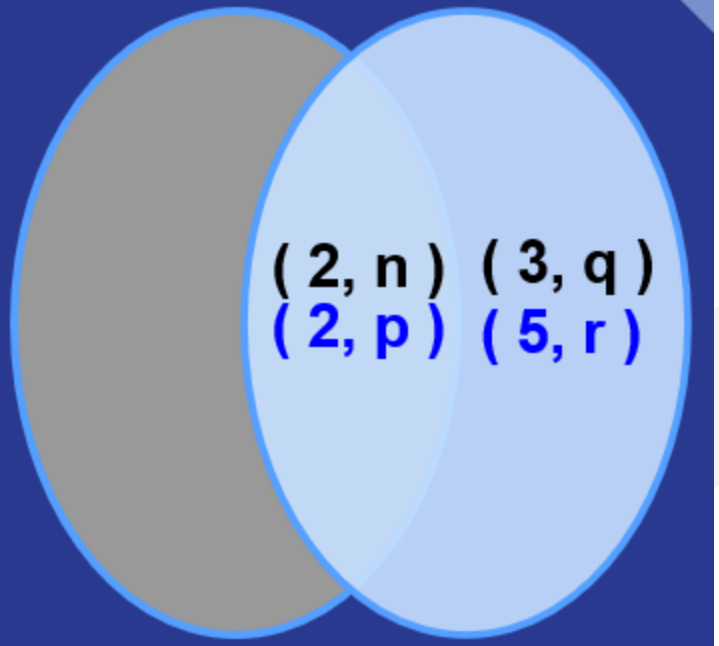
A	N
2	p
3	q
5	r

table_B

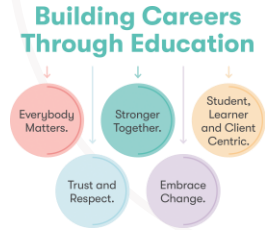
```
SELECT * FROM table_A
RIGHT JOIN table_B
ON table_A.A=table_B.A;
```

A	M	A	N
2	n	2	p
null	null	3	q
null	null	5	r

Output



CROSS JOIN



- The SQL CROSS JOIN produces a result set which is the number of rows in the first table multiplied by the number of rows in the second table, if no WHERE clause is used along with CROSS JOIN.

- This kind of result is called as Cartesian Product.

- If, WHERE clause is used with CROSS JOIN, it functions like an INNER JOIN.

BPP

Example: CROSS JOIN

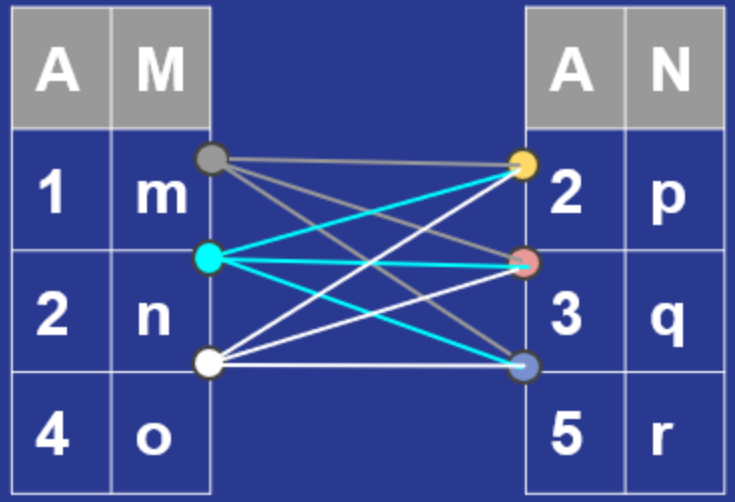
A	M
1	m
2	n
4	o

table_A

A	N
2	p
3	q
5	r

table_B

```
SELECT *
FROM table_A
CROSS JOIN  table_B;
```

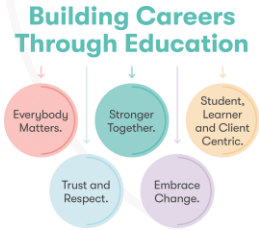


A	M	A	N
1	m	2	p
2	n	2	p
4	o	2	p
1	m	3	q
2	n	3	q
4	o	3	q
1	m	5	r
2	n	5	r
4	o	5	r

Output

BPP

Union



The UNION operator is used to combine the result-set of two or more SELECT statements.

- Every SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types
- The columns in every SELECT statement must also be in the same order

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

Subquery

In SQL a Subquery can be simply defined as a query within another query. In other words we can say that a Subquery is a query that is embedded in WHERE clause of another SQL query.

Important rules for Subqueries:

1. 1)You can place the Subquery in a number of SQL clauses: WHERE clause, HAVING clause, FROM clause.

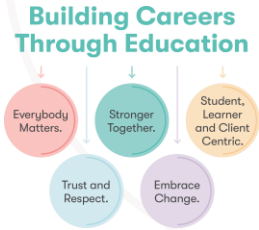
2. Subqueries can be used with SELECT, UPDATE, INSERT, DELETE statements along with expression operator. It could be equality operator or comparison operator such as =, >, =, <= and Like operator.

3. The subquery generally executes first, and its output is used to complete the query condition for the main or outer query.

4. Subquery must be enclosed in parentheses.

BPP

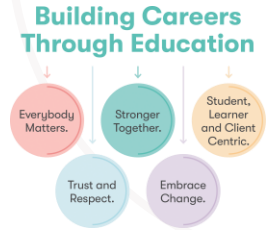
Syntax




```
SELECT column_name
FROM table_name
WHERE column_name expression operator
    ( SELECT COLUMN_NAME  from TABLE_NAME   WHERE ... );
```

BPP

Recap - SQL practice with your tutor



Foundational Language for Data Management

SQL (Structured Query Language) is the standard language for relational database management and data manipulation. It allows users to create, retrieve, update, and delete database records efficiently.

Ubiquitous and Standardised

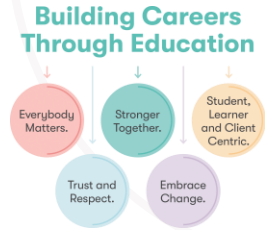
SQL is supported by virtually all relational database systems like MySQL, PostgreSQL, Oracle, and SQL Server, making it a critical skill for data professionals.

Enhanced Data Retrieval

SQL provides powerful but straightforward means to retrieve data from databases through SELECT queries, enabling complex analytics and reporting.

BPP

Recap – Why practice SQL



Data Manipulation and Administration

Beyond data retrieval, SQL is instrumental in structuring and managing large quantities of data, supporting operations like inserting new data, updating existing data, and performing transactional processes.

Integration with Other Technologies

SQL databases easily integrate with numerous reporting and analytics tools, making SQL a pivotal part of data-driven decision-making processes in businesses.

Advanced Data Analysis and Business Intelligence

SQL will be used to extract and analyze data, forming the basis for decision-making in business intelligence and data analytics topics.

BPP

Practice with tutor

Building Careers Through Education

Everybody Matters. Stronger Together. Student, Learner and Client Centric. Trust and Respect. Embrace Change.

https://sqlbolt.com/lesson/select_queries_with_joins

https://sqlbolt.com/lesson/select_queries_with_outer_joins

https://sqlbolt.com/lesson/select_queries_with_nulls

BPP

Building Careers Through Education

Everybody Matters. Stronger Together. Student, Learner and Client Centric. Trust and Respect. Embrace Change.

Break

BPP

Section Agenda

What you will learn about in this section:

1. Aggregation operators
2. GROUP BY
3. GROUP BY: with HAVING, semantics
4. ACTIVITY

Aggregation


```
SELECT AVG(price)
FROM   Product
WHERE  maker = "Toyota"
```

```
SELECT COUNT(*)
FROM   Product
WHERE  year > 1995
```

- SQL supports several **aggregation** operations:
  - SUM, COUNT, MIN, MAX, AVG

Except COUNT, all aggregations apply to a single attribute

BPP

Aggregation: COUNT

COUNT applies to duplicates, unless otherwise stated

```
SELECT COUNT(category)
FROM   Product
WHERE  year > 1995
```

Note: Same as COUNT(*). Why?

We probably want:

```
SELECT COUNT(DISTINCT category)
FROM   Product
WHERE  year > 1995
```

BPP

More examples


```
Purchase(product, date, price, quantity)
```

```
SELECT SUM(price * quantity)
FROM   Purchase
```

```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = 'bagel'
```

What do these mean?

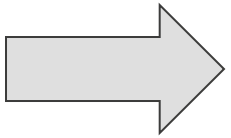
BPP

Simple Aggregations

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = 'bagel'
```



50 (= 1*20 + 1.50*20)

Grouping and Aggregation

Purchase(product, date, price, quantity)

```
SELECT   product,
          SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Find total sales after 10/1/2005 per product.

Let's see what this means…

BPP

Grouping and Aggregation

Semantics of the query:

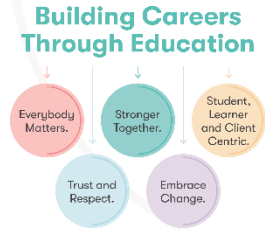
1. Compute the FROM and WHERE clauses

2. Group by the attributes in the GROUP BY

3. Compute the SELECT clause: grouped attributes and aggregates

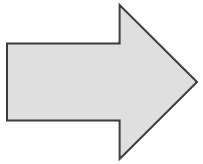
BPP

1. Compute the FROM and WHERE clauses




```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

FROM



Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

2. Group by the attributes in the GROUP BY


```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

GROUP BY

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

BPP

3. Compute the SELECT clause: grouped attributes and aggregates


```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

SELECT

Product	TotalSales
Bagel	50
Banana	15

BPP

HAVING Clause


```
SELECT   product, SUM(price*quantity)
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
HAVING   SUM(quantity) > 100
```

Same query as before, except that we consider only products that have more than100 buyers

HAVING clauses contains conditions on **aggregates**

Whereas WHERE clauses condition on ***individual tuples…***

BPP

General form of Grouping and Aggregation


```
SELECT    S
FROM      R1,…,Rn
WHERE     C1
GROUP BY  a1,…,ak
HAVING    C2
```

- **S** = Can ONLY contain attributes $a_1,\ldots,a_k$ and/or aggregates over other attributes
- **C_1** = is any condition on the attributes in $R_1,\ldots,R_n$
- **C_2** = is any condition on the aggregate expressions

Why?

BPP

General form of Grouping and Aggregation


```
SELECT    S
FROM      R1,…,Rn
WHERE     C1
GROUP BY  a1,…,ak
HAVING    C2
```

Evaluation steps:

1. Evaluate FROM-WHERE: apply condition C_1 on the attributes in $R_1,\ldots,R_n$
2. GROUP BY the attributes $a_1,\ldots,a_k$
3. Apply condition C_2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

BPP

Group-by v.s. Nested Query

Author(login, name)

Wrote(login, url)

- Find authors who wrote ³ 10 documents:
- Attempt 1: with nested queries

```
SELECT DISTINCT Author.name
FROM   Author
WHERE  COUNT(
          SELECT Wrote.url
     FROM   Wrote
     WHERE  Author.login = Wrote.login) > 10
```

This is SQL by a novice

Group-by v.s. Nested Query

- Find all authors who wrote at least 10 documents:
- Attempt 2: SQL style (with GROUP BY)

```
SELECT   Author.name
FROM     Author, Wrote
WHERE    Author.login = Wrote.login
GROUP BY Author.name
HAVING   COUNT(Wrote.url) > 10
```

No need for DISTINCT: automatically from GROUP BY

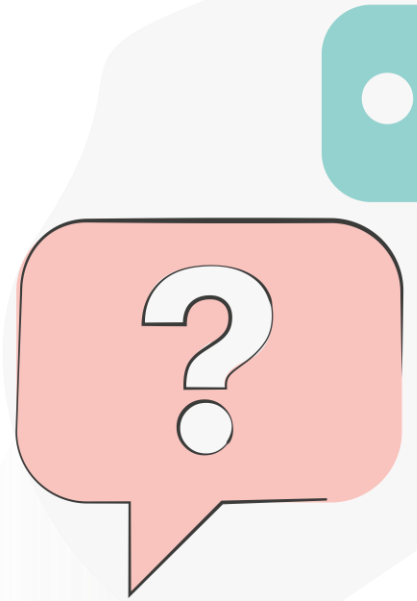
This is SQL by an expert

Group-by v.s. Nested Query

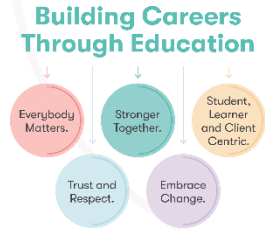
Which way is more efficient?

- **Attempt #1-** *With nested:* How many times do we do a SFW query over all of the Wrote relations?

- **Attempt #2-** *With group-by*: How about when written this way?



Practice with tutor



https://sqlbolt.com/lesson/select_queries_with_aggregates

https://sqlbolt.com/lesson/select_queries_with_aggregates_pt_2

https://sqlbolt.com/lesson/select_queries_order_of_execution

BPP

Building Careers Through Education

Everybody Matters. Stronger Together. Student, Learner and Client Centric. Trust and Respect. Embrace Change.

Break

BPP

Section Agenda

What you will learn about in this section:

1. Aggregation operators
2. GROUP BY
3. GROUP BY: with HAVING, semantics
4. ACTIVITY

Section Agenda

Data Definition in SQL

So far we have seen the ***Data Manipulation Language***, DML
Next: ***Data Definition Language*** (DDL)

1. **Data types:**
   - Defines the types.

2. **Data definition:** defining the schema.

   - Create tables
   - Delete tables
   - Modify table schema

3. **Indexes:** to improve performance

BPP

Data Types in SQL

Characters:

- CHAR(20) -- fixed length
- VARCHAR(40) -- variable length

Numbers:

- INT, REAL plus variations

Times and dates:

- DATE, DATETIME

Reusing domains:

```
CREATE  DOMAIN address AS VARCHAR(55)
```

BPP

Creating Tables

Example:

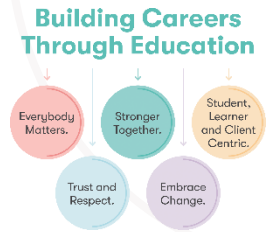
```
CREATE  TABLE Person(

        name                        VARCHAR(30),
        social-security-number      INT,
        age                         SHORTINT,
        city                        VARCHAR(30),
        gender                      BIT(1),
        Birthdate                   DATE

 );
```

BPP

Deleting or Modifying a Table



Deleting:

Example:

```
DROP Person;
```

Exercise with care !!

Altering: (adding or removing an attribute).

Example:

```
ALTER TABLE   Person
        ADD   phone  CHAR(16);

ALTER  TABLE   Person
        DROP  age;
```

What happens when you make changes to the schema?

BPP

Default Values

Specifying default values:

```
CREATE  TABLE Person(
        name        VARCHAR(30),
        ssn         INT,
        age         SHORTINT        DEFAULT     100,
        city        VARCHAR(30)     DEFAULT     'Southampton',
        gender      CHAR(1)         DEFAULT     '?',
        Birthdate   DATE
```

The default of defaults: NULL

BPP

Indexes

REALLY important for speeding up query processing time.

Suppose we have a relation

Person (name, age, city)

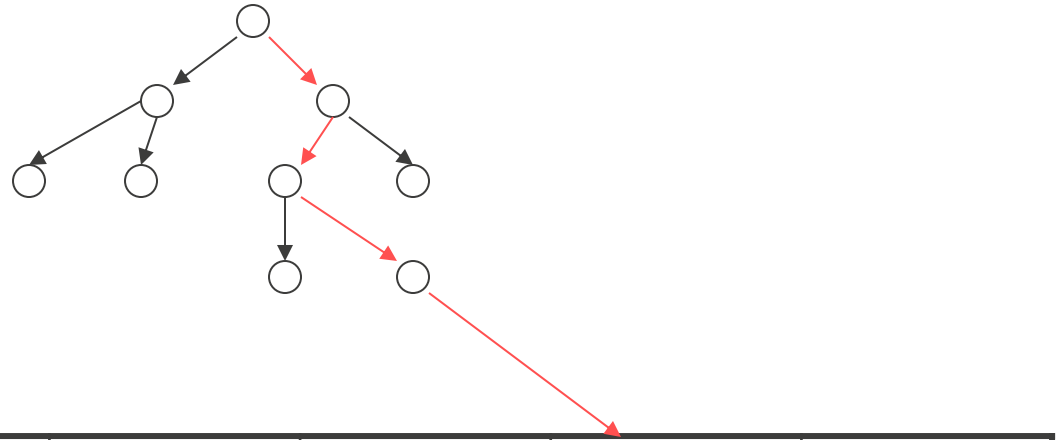
```
SELECT *
FROM   Person
WHERE  name = “Smith”
```

Sequential scan of the file Person may take a long time

BPP

Indexes

Create an index on name:



Adam	Betty	Charles	….	Smith	….

B+ trees have fan-out of 100s: max 4 levels !

Creating Indexes

Syntax:

```
CREATE INDEX  nameIndex ON Person(name)
```

BPP

Creating Indexes

Indexes can be created on more than one attribute:

Example:

```
CREATE INDEX doubleindex ON
                    Person (age, city)
```

Helps in:

```
SELECT *
FROM    Person
WHERE age = 55 AND city = “Southampton”
```

But not in:

```
SELECT *
FROM    Person
WHERE city = “Southampton”
```

BPP

Creating Indexes

Indexes can be useful in range queries too:

```
CREATE INDEX ageIndex ON  Person (age)
```

B+ trees help in:

```
SELECT *
FROM Person
WHERE age > 25 AND age < 28
```

Why not create indexes on everything?

BPP

Defining Views

Views are relations, except that they are not physically stored.

For presenting different information to different users

Employee(ssn, name, department, project, salary)

```
CREATE VIEW  Developers AS
   SELECT name, project
   FROM  Employee
   WHERE department = “Development”
```

Payroll has access to Employee, others only to Developers

BPP

A Different View

Person(name, city)
Purchase(buyer, seller, product, store)
Product(name, maker, category)

```
CREATE VIEW  Southampton-view  AS

      SELECT  buyer, seller, product, store
      FROM     Person, Purchase
      WHERE   Person.city = “Southampton”    AND
                Person.name = Purchase.buyer
```

We have a new virtual table:
Southampton-view(buyer, seller, product, store)

A Different View

We can later use the view:

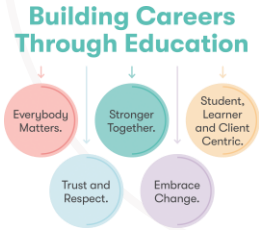
```
SELECT   name, store
FROM     Southampton-view, Product
WHERE    Southampton-view.product = Product.name  AND
         Product.category = “shoes”
```

What Happens When We Query a View ?


```
SELECT   name, Southampton-view.store
FROM     Product, Southampton-view
WHERE    Southampton-view.product = Product.name  AND
         Product.category = “shoes”
```

```
SELECT     name, Southampton-view.store
FROM       Product,
           (SELECT     buyer, seller, product, store
           FROM        Person, Purchase
           WHERE       Person.city = “Southampton”    AND
                       Person.name = Purchase.buyer)
           AS Southampton-view
WHERE      Southampton-view.product = Product.name  AND
           Product.category = “shoes”
```

Types of Views



Virtual views

- Used in databases
- Computed only on-demand – slow at runtime
- Always up to date

Materialized views

- Used in data warehouses
- Precomputed offline – fast at runtime
- May have stale (old) data

BPP

Updating Views

How can I insert a tuple into a table that doesn't exist?

Employee(ssn, name, department, project, salary)

```
CREATE VIEW  Developers AS
  SELECT name, project
  FROM  Employee
  WHERE department = "Development"
```

If we make the following insertion:

```
INSERT INTO  Developers
VALUES("Joe", "Optimizer")
```

Is there anything missing?

It becomes:

```
INSERT INTO  Employee
VALUES(NULL, "Joe", NULL, "Optimizer", NULL)
```

BPP

Non-Updatable Views


```
CREATE VIEW  Southampton-view  AS

      SELECT  seller, product, store
      FROM     Person, Purchase
      WHERE   Person.city = "Southampton"    AND
                Person.name = Purchase.buyer
```

How can we add the following tuple to the view?

("Joe", "Shoe Model 12345", "Nine West")

We need to add "Joe" to Person first, but how do we know this?

BPP

Non-Updatable Views

- When we need to update several tables
- When the SELECT uses a column more than once
- When DISTINCT is used
- When there is an Aggregate, GROUP BY, HAVING
- When there is UNION (ALL)

BPP

Thank you

Do you have any questions, comments, or feedback?