JAVA BACKEND DEVELOPMENT PROGRAM

SQL Part 2

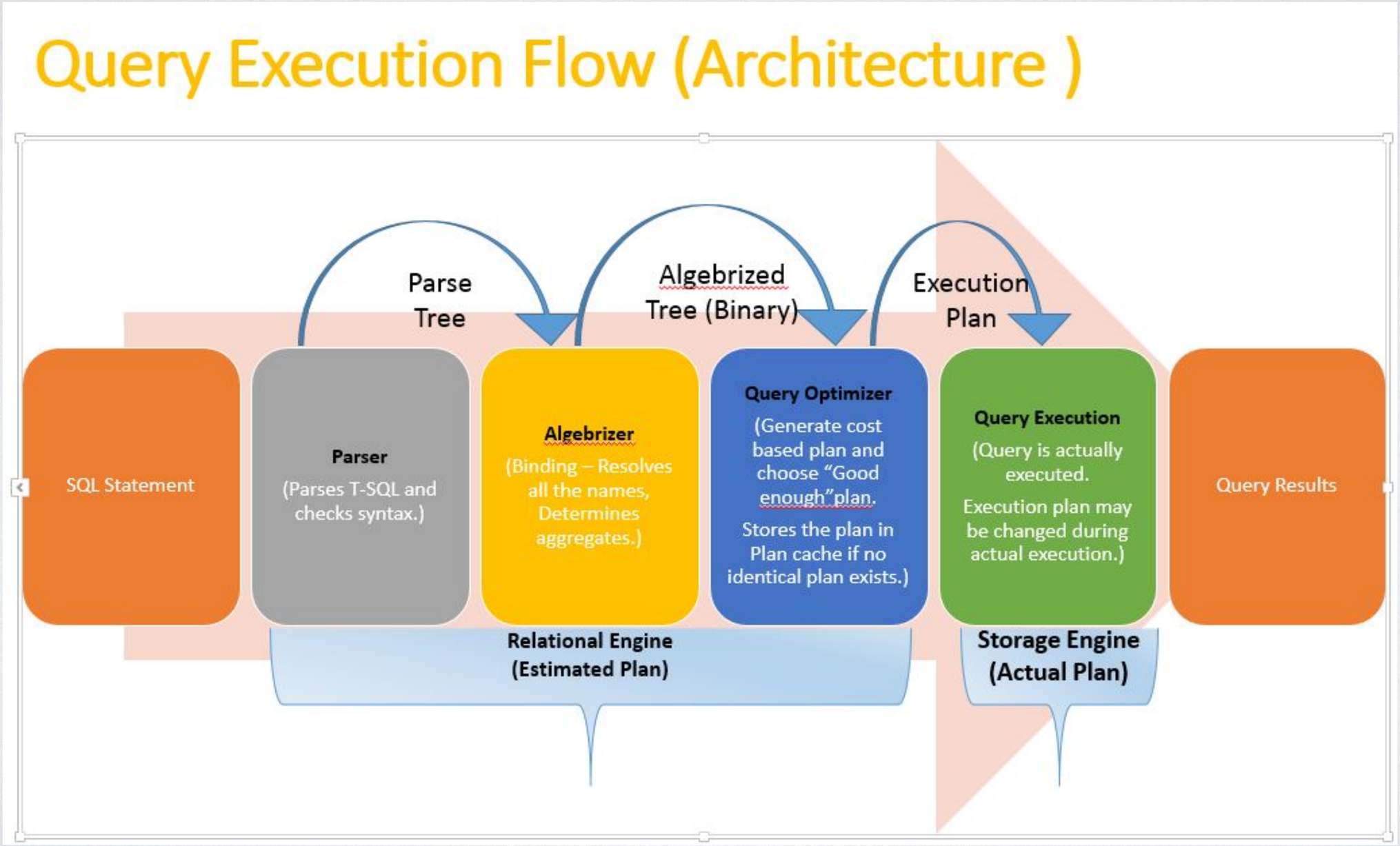
OUTLINE

- Aggregate functions
- Queries (WHERE, ORDER BY, LIMIT, GROUP BY, HAVING)
- Join
- Set Operations
- Sub queries
- View
- Variable
- Temporary Table
- Control Flow
- User-defined Procedure & Function
- Index & Balance Tree

QUERY EXECUTION

- Step 1:
  - Parser check query syntax
  - Break query to token --> (intermediate files)
- Step 2:
  - Query Optimizer creates best possible execution plan based on current resource utilization
- Step 3:
  - DB engine --> Run the query

QUERY EXECUTION FLOW



Aggregations

Can compute simple statistics using built-in SQL functions

. SUM
. AVG
. COUNT
. MAX
. MIN
. and more …

Aggregate Functions & NULL

For NULL values:

- COUNT(1) or COUNT(*) will count number of all records (including duplicates)
- COUNT(<attribute>) will count number of unique non-null records
- AVG, MIN, MAX, etc. ignore NULL values
- GROUP BY includes a row for NULL

WHERE

The WHERE clause is used to filter records.

It is used to extract only those records that fulfill a specified condition.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

WHERE

1<attribute> = <value>

2 <attribute> BETWEEN [value1] AND [value2]

3 <attribute> IN ([value1], [value2], ...)

4 <attribute> LIKE 'SST%'

5 <attribute> LIKE 'SST_'

6 <attribute> IS NULL and [attribute] IS NOT NULL

7 Logical combinations with AND and OR

8 Mathematical functions <>, !=, >, <, ...

9 Subqueries …

ORDER BY

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

LIMIT K

The LIMIT clause is used to specify the number of records to return.

It is useful on large tables with thousands of records. Returning a large number of records can impact performance.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

GROUP BY

The GROUP BY statement groups rows that have the same values into summary rows.

The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

HAVING

The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

GROUP BY Conceptually

- Given the following data

DRINKER	COFFEE	SCORE
Risa	Espresso	2
Chris	Cold Brew	1
Chris	Turkish Coffee	5
Risa	Cold Brew	4
Risa	Cold Brew	5

? What is each drinker's average coffee rating?

1 `GROUP BY` DRINKER

DRINKER	COFFEE	SCORE
Chris	Cold Brew	1
Chris	Turkish Coffee	5
Risa	Espresso	2
Risa	Cold Brew	4
Risa	Cold Brew	5

2 Aggregate

DRINKER	AVGSCORE
Chris	3
Risa	3.67

HAVING Conceptually

- Given the following data

DRINKER	COFFEE	SCORE
Risa	Espresso	2
Chris	Cold Brew	1
Chris	Turkish Coffee	5
Risa	Cold Brew	4
Risa	Cold Brew	5

? What is the highest rated type of coffee, on average, considering only coffees that have at least 3 ratings?

1 `GROUP BY` COFFEE

DRINKER	COFFEE	SCORE
Chris	Cold Brew	1
Risa	Cold Brew	4
Risa	Cold Brew	5
Chris	Turkish Coffee	5
Risa	Espresso	2

2 Aggregate

COFFEE	AVGSCORE
Cold Brew	3.33
Turkish Coffee	5
Espresso	2

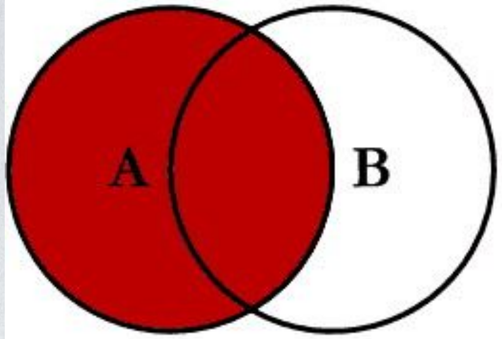
3 `HAVING` COUNT(*) >= 3

DRINKER	AVGSCORE
Cold Brew	3.33

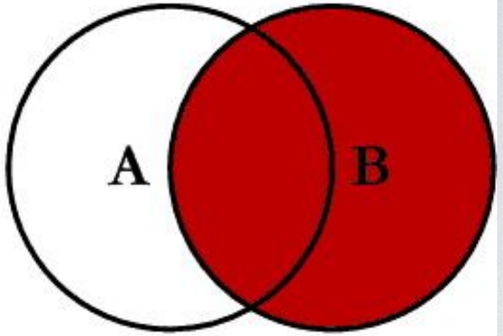
JOINS

- Joins are used to combine data sets on a row by row level based on matching columns
- Uses matching data in specified columns to combine or sort data
- Columns DO NOT have to have the same name
- Columns DO NOT need to be keys
- Scope table to table, table to view, table to synonyms.

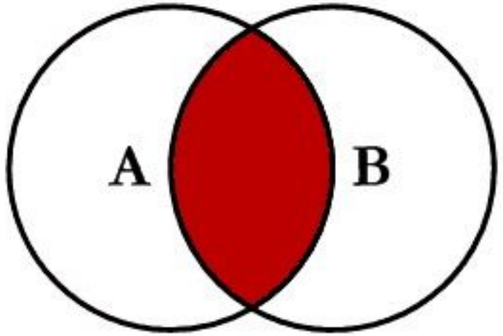
SQL JOINS



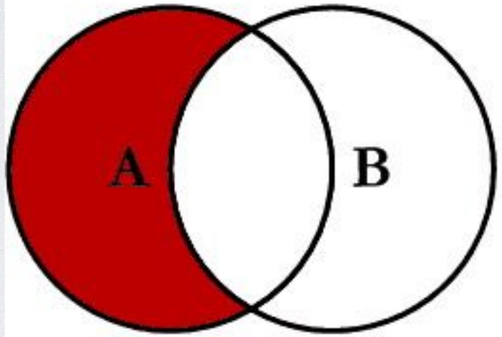

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



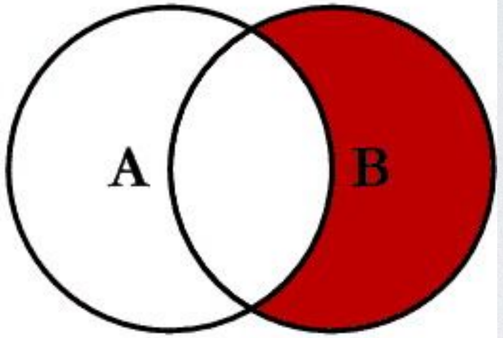

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



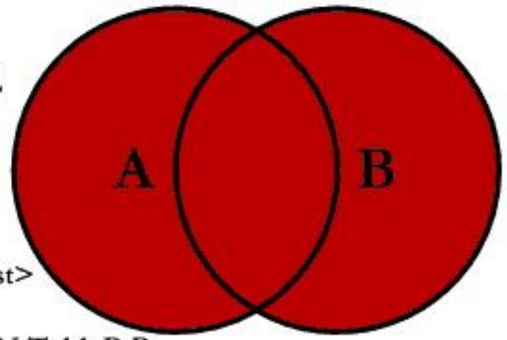

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



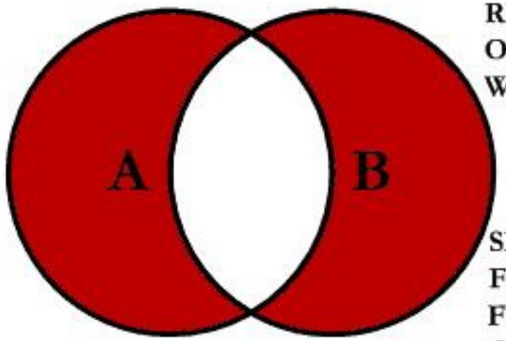

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```

JOINS

- Inner Join
  - Connects on only matching data
  - Will only display values that match on both sides of the table, all others are excluded
- Left Join
  - Displays the matching data that you'd see from inner join, and all the unique values from the left table
- Right Join
  - Same as left join, but now the unique values come from the right table

JOINS

- Full Outer
  - Display ALL values, matching and non-matching
  - MySQL doesn’t support Full Outer join. It can be achieved by using left join and right join
- Cross Join
  - Display every possible combination of all values in the designated columns
  - Cartesian Product
- Self Join
  - Join a table to itself in some regard

INNER JOIN

R INNER JOIN S ON <condition>

R JOIN S ON <condition>

R NATURAL JOIN S

- Used to match up tuples from different relations
- Includes only the relations with matching attribute values

LEFT/RIGHT OUTER JOIN

R LEFT OUTER JOIN S ON <condition>

R RIGHT OUTER JOIN S ON <condition>

- Used to match up tuples from different relations
- Includes all the relations from the "outer" side
- If there is no matching tuple, assigns NULLs
- Tip: Pick one direction and use it consistently

FULL OUTER JOIN

Used to match up tuples from different relations

Includes all the relations from both sides

If there is no matching tuple, assigns NULLs

Returns a relation with all the attributes of R • all the attributes of S

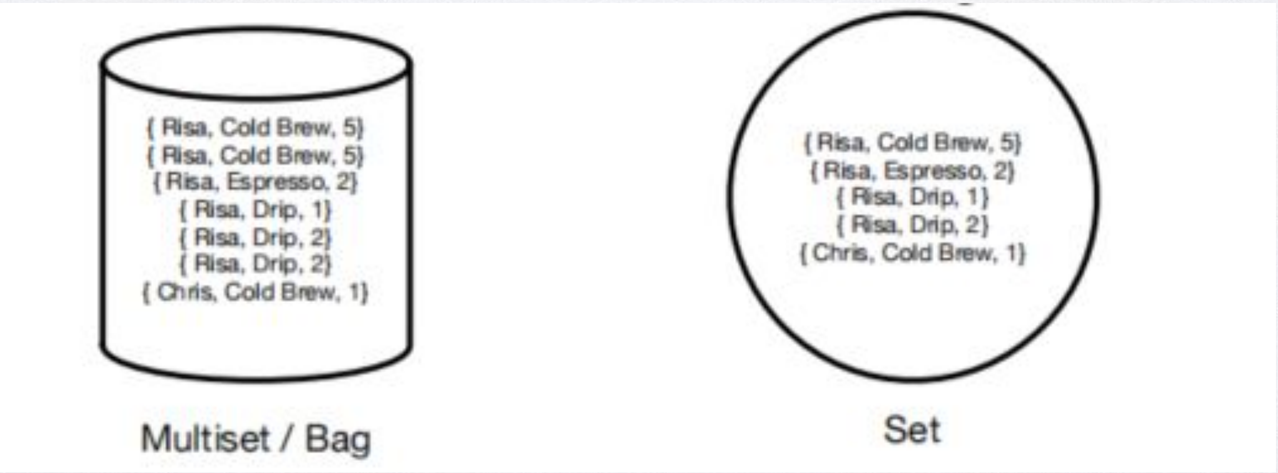
Self Join

R AS R1 JOIN R AS R2 ON R1.<att> <op> R2.<att>

- Used to match up tuples from relation R back to itself
- Any type of JOIN may be used
- Returns a relation with all the attributes of R • all the attributes of R

SET OPERATIONS

- Results are unordered multisets/bag
- It could be useful to perform operations (union, intersection, difference) on these
- Different RDBMs provide different levels of support
    - Multiset/bag stores duplicate records
    - Set stores unique records



UNION and UNION ALL

UNION- eliminates duplicates

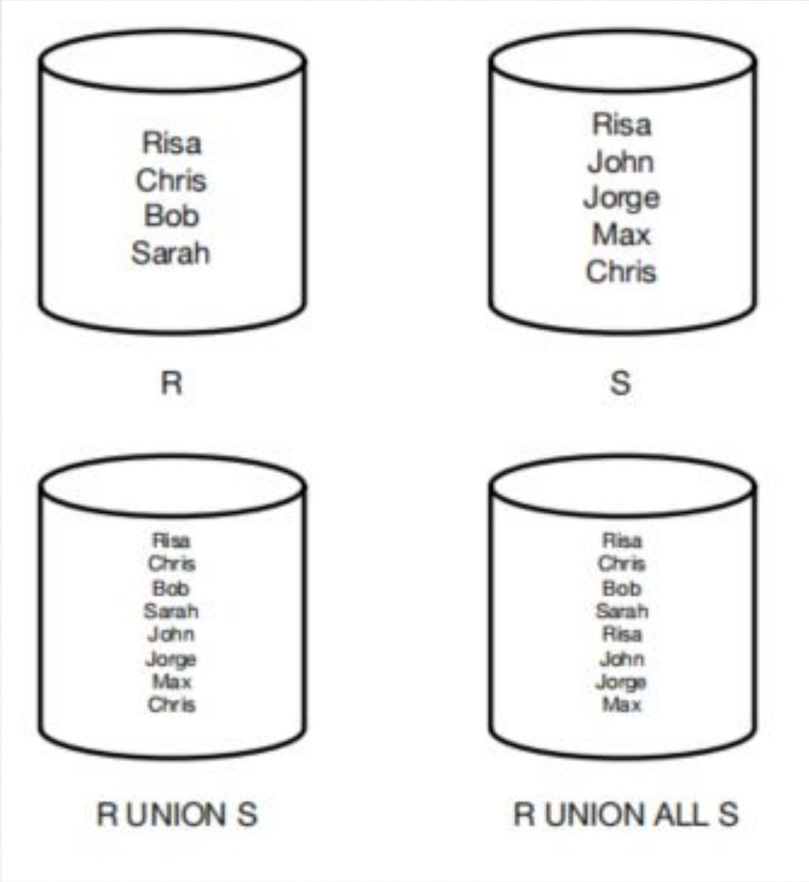
UNION ALL- does NOT eliminate

duplicates

Uses the column names from the first result set

Data types must match

Number of attributes must match



UNION and UNION ALL Example

- **STUDENT(NETID, FIRSTNAME, LASTNAME)**
- **FACULTY(NETID, FIRSTNAME, LASTNAME)**

```
SELECT lastName, firstName, 'student'

FROM Student

UNION

SELECT lastName, firstName, 'faculty'

FROM Faculty;
```

Intersection and Difference

Intersection - Implemented via INNER JOIN

Difference - Implemented via EXCEPT

Displays the values in the first select statement MINUS any values found in the second select statement

Subqueries

We can have a subquery in the WHERE clause

- It's linked with keywords
  - EXISTS / NOT EXISTS
    - If the subquery returns at least one tuple, the EXISTS clause evaluates to TRUE
  - <operand> IN / <operand> NOT IN
  - <operand> <comparison operator> ALL
  - <operand> <comparison operator> SOME/ANY

Subqueries - How do they work?

- Basically, we iterate over the tuples in the outer query and evaluate the inner query for each outer tuple
- Some can be evaluated once and the result is used in the outer query
    - Ex: a subquery that returns the number of CAFES that are frequented
- Some require the subquery to be evaluated for every value assignment in the outer query (correlated subquery)
    - Ex: a subquery that returns the number of CAFES that each DRINKER frequents

Subquery Example 1 IN

LIKES (DRINKER, COFFEE)

- Who likes 'Cold Brew'and 'Espresso'?
- Both queries return the same result
- Many subqueries can be written as JOINS, people tend to find it easier to reason about one way or the other

```
SELECT DISTINCT l.DRINKER
FROM LIKES l
WHERE l.COFFEE = 'Cold Brew'
        AND l.DRINKER IN (
                SELECT l2.DRINKER
                FROM LIKES l2
                   WHERE l2.COFFEE = 'Espresso')
```

```
SELECT DISTINCT l1.DRINKER
FROM LIKES l1, LIKES l2
WHERE l1.DRINKER = l2.DRINKER
      AND l1.COFFEE = 'Cold Brew'
      AND l2.COFFEE = 'Espresso'
```

Subquery Example 2 EXISTS

LIKES (DRINKER, COFFEE)

FREQUENTS (DRINKER, CAFE)

SERVES (CAFE, COFFEE)

Who goes to a cafe that serves 'Cold Brew'?

```
SELECT DISTINCT f.DRINKER
FROM FREQUENTS f, SERVES s
WHERE f.CAFE = s.CAFE
      AND s.COFFEE = 'Cold Brew'
```

```
SELECT DISTINCT f.DRINKER
FROM FREQUENTS f
WHERE EXISTS (
          SELECT s.CAFE
          FROM SERVES s
          WHERE s.COFFEE = 'Cold Brew'
          AND f.CAFE = s.CAFE)
```

Subquery Example 3

LIKES (DRINKER, COFFEE)

FREQUENTS (DRINKER, CAFE)

SERVES (CAFE, COFFEE)

Who likes all of the coffees that Risa likes?

- There doesn't exist a coffee Risa likes that is not also liked by these drinkers
- Every coffee Risa likes is liked by these drinkers BUT they might like other coffees as well

Subquery Example 3 Step 1

LIKES (DRINKER, COFFEE)

FREQUENTS (DRINKER, CAFE)

SERVES (CAFE, COFFEE)

Coffees that Risa likes

SELECT I2.COFFEE

FROM LIKES I2

WHERE I2.DRINKER = 'Risa'

Subquery Example 3 Step 2

LIKES (DRINKER, COFFEE)

FREQUENTS (DRINKER, CAFE)

SERVES (CAFE, COFFEE)

Who likes all of the coffees that Risa likes?

SELECT DISTINCT l.DRINKER

FROM LIKES l

WHERE NOT EXISTS (a coffee Risa likes that is not also liked by l.DRINKER)

Subquery Example 3 Step 3

- LIKES (DRINKER, COFFEE)
- FREQUENTS (DRINKER, CAFE)
- SERVES (CAFE, COFFEE)
- **Who likes all of the coffees that Risa likes?**

```
SELECT DISTINCT l.DRINKER
FROM LIKES l
WHERE NOT EXISTS (
        SELECT l2.COFFEE
        FROM LIKES l2
        WHERE l2.DRINKER = 'Risa' AND l2.COFFEE NOT IN (
          the set of coffees liked by l.DRINKER))
```

Subquery Example 3 Final

- LIKES (DRINKER, COFFEE)
- FREQUENTS (DRINKER, CAFE)
- SERVES (CAFE, COFFEE)
- **Who likes all of the coffees that Risa likes?**

```
SELECT DISTINCT I.DRINKER
FROM LIKES I
WHERE NOT EXISTS (
        SELECT I2.COFFEE
        FROM LIKES I2
        WHERE I2.DRINKER = 'Risa'
        AND I2.COFFEE NOT IN (
                SELECT I3.COFFEE
                FROM LIKES I3
                WHERE I3.DRINKER = I.DRINKER))
```

SOME/ANY

SOME/ANY is used like “expression boolOp {SOME, ANY }(subquery)”

SOME/ANY returns TRUE if there is at least 1 item in the subquery can make the boolOp evaluate to true

ALL predicate

ALL is used like “expression boolOp ALL (subquery)”

Similar to SOME

BoolOp must evaluate to true for everything in the subquery

ALL Example

RATES (DRINKER, COFFEE, SCORE)

```
SELECT DISTINCT r.DRINKER
FROM RATES r
WHERE r.SCORE < ALL (
        SELECT r2.SCORE
        FROM RATES r2
        WHERE r2.DRINKER = 'Risa')
```

Subqueries in FROM Clause

FREQUENTS (DRINKER, CAFE)

- Can have a subquery in FROM clause
- Treated as a temporary table
- MUST be assigned an alias

? Who goes to a cafe that serves 'Cold Brew'?

```
SELECT DISTINCT f.DRINKER
FROM FREQUENTS f, SERVES s
WHERE f.CAFE = s.CAFE
        AND s.COFFEE = 'Cold Brew'
```

```
SELECT DISTINCT f.DRINKER
FROM FREQUENTS f,
      (SELECT s.CAFE FROM SERVES s
      WHERE s.COFFEE = 'Cold Brew') s2
WHERE f.CAFE = s2.CAFE
```

Subquery in FROM Clause

FREQUENTS (DRINKER, CAFE)

Note: The code is a lot cleaner with a view!

```
CREATE VIEW CB_COFFEE AS
SELECT s.CAFE FROM SERVES s
    WHERE s.COFFEE = 'Cold Brew'

SELECT DISTINCT f.DRINKER
FROM FREQUENTS f, CB_COFFEE c
WHERE f.CAFE = c.CAFE
```

VIEW

- A view is often seen as a virtual table
- It is updated every time is referred to. (Different from temporary table, whose value was inserted at time of creation.)
- It displays data that you choose, but does not actually hold any data
- Good for security since you can prevent showing extra data
- DML operations just happen on the table.
  - You can modify data on view level, and the source data will be updated as well.

```
CREATE VIEW hiredate_view
AS
SELECT p.FirstName, p.LastName, e.BusinessEntityID, e.HireDate
FROM HumanResources.Employee e
JOIN Person.Person AS p ON e.BusinessEntityID = p.BusinessEntityID ;
GO
```

VARIABLES

- An object that can store a single data value in a specified data type
- Variables Scope
  - Local
  - session
  - Globe
- User-Defined Variables are displayed with an “@” symbol
- System Variables are displayed with an "@@" symbol

Variable Example

```
SET @var_name = expression
mysql>SET @var1 = 2+6;
mysql>SET @var2 := @var1-2;
mysql>SELECT @var1, @var2;

+-------+-------+
| @var1 | @var2 |
+-------+-------+
|   8   |   6   |
+-------+-------+
```

```
SET GLOBAL max_connections = 1000;
SET @@GLOBAL.max_connections = 1000;

SET SESSION sql_mode = 'TRADITIONAL';
SET @@SESSION.sql_mode =
'TRADITIONAL';
SET @@sql_mode = 'TRADITIONAL';
```

TEMPORARY TABLE

- Temporary tables are “temporary” tables
- Valid within session. Last as long as you’re session
- Two types of temporary tables
  - Global -- Can be used across sessions
  - Local -- Can be used only within current session
- Best used to working with temp data
- Stored in TempDB
- Cannot be partitioned

```
CREATE TEMPORARY TABLE table_name(
    column_1_definition,
    column_2_definition,
    ...,
    table_constraints
);
```

CONTROL FLOW

- Case When Statements (exactly like if-else in Java)
  - Uses a CASE and END block to specify a set of conditions and a series of multiple results depending on the data
  - Mainly focuses on changing the value for a single column based on another column
  - Example
  - When Weekday = 1 THEN ‘Sunday’

UD STORED PROCEDURE

- User Defined Stored Procedures are just Stored Procedures, but created by the user
- Contains statements including calling other stored procedures
- Can have different Input and Output Parameters
- Can only RETURN int
- Must be recompiled after time or changes

UD STORED PROCEDURE

- Parameters for Stored Procedures
  - Input Parameters
    - Specify variables to be taken in when using a stored procedure
  - Output Parameters
    - Used to output multiple values
    - Output value can be used in the procedure or batch that calls it
  - Default Parameter
    - Assigns the input or output parameter with a default value

UD STORED PROCEDURE

- Difference between Output Parameter and Return Value in Stored Procedure
  - Output parameter
    - An output parameter in a Stored Procedure is used to return any value.
    - An output parameter can return one or more values.
    - An output parameter returns data with any data type
  - Return value (NOT SUPPORTED BY MySQL)
    - Generally, a return value is used to convey success or failure.
    - A return value can return only one value.
    - The return value returns data of only an integer data type.

UD STORED PROCEDURE

```
DELIMITER //

CREATE PROCEDURE GetOfficeByCountry(
        IN countryName VARCHAR(255)
)
BEGIN
        SELECT *
        FROM offices
        WHERE country = countryName;
END //

DELIMITER ;
```

```
CALL GetOfficeByCountry('USA');
```

officeCode	city	phone	addressLine1	addressLine2	state	country	postalCode	territory
1	San Francisco	+1 650 219 4782	100 Market Street	Suite 300	CA	USA	94080	NA
2	Boston	+1 215 837 0825	1550 Court Place	Suite 102	MA	USA	02107	NA
3	NYC	+1 212 555 3000	523 East 53rd Street	apt. 5A	NY	USA	10022	NA

```
CALL GetOfficeByCountry();
```

Will result in Error

UD STORED PROCEDURE

```
DELIMITER $$

CREATE PROCEDURE GetOrderCountByStatus (
        IN  orderStatus VARCHAR(25),
        OUT total INT
)
BEGIN
        SELECT COUNT(orderNumber)
        INTO total
        FROM orders
        WHERE status = orderStatus;
END$$

DELIMITER ;
```

```
CALL GetOrderCountByStatus('Shipped',@total);
SELECT @total;
```

	@total
▶	303

USER DEFINED FUNCTION

- User Defined Functions are just like functions in SQL, but created by the user
- Can have Inputs, NO outputs though
- Can return different values
- Must have a return value
- Used for Complex Business formulas or other mathematical operations

USER DEFINED FUNCTIONS

- Scalar Function
  - Functions that returns a scalar(single) value
- Table-valued function (MySQL doesn't support for Table-Valued function)
  - Functions that returns a table
    - Inline table-valued function
      - No begin and end clause
      - Returns a query result directly
    - Multi-statement table-valued function
      - Contains function body that allows multiple statements inside the function

USER DEFINED FUNCTIONS

CREATE FUNCTION sf_name ([parameter(s)])

RETURNS data type

DETERMINISTIC

STATEMENTS

USER DEFINED FUNCTIONS

```
DELIMITER |
CREATE FUNCTION sf_past_movie_return_date (return_date DATE)
  RETURNS VARCHAR(3)
   DETERMINISTIC
    BEGIN
     DECLARE sf_value VARCHAR(3);
        IF curdate() > return_date
           THEN SET sf_value = 'Yes';
        ELSEIF  curdate() <= return_date
           THEN SET sf_value = 'No';
        END IF;
     RETURN sf_value;
    END|

SELECT `movie_id`,`membership_number`,`return_date`,CURDATE()
,sf_past_movie_return_date(`return_date`)  FROM `movierentals`;
```

movie_id	membership_number	return_date	CURDATE()	sf_past_movie_return_date('return_date')
1	1	NULL	04-08-2012	NULL
2	1	25-06-2012	04-08-2012	yes
2	3	25-06-2012	04-08-2012	yes
2	2	25-06-2012	04-08-2012	yes
3	3	NULL	04-08-2012	NULL

TABLE-VALUED FUNCTION

A table-valued function is a user-defined function that returns data of a table type.

The return type of a table-valued function is a table so the table-valued function can be used similar to how a normal table can be used.

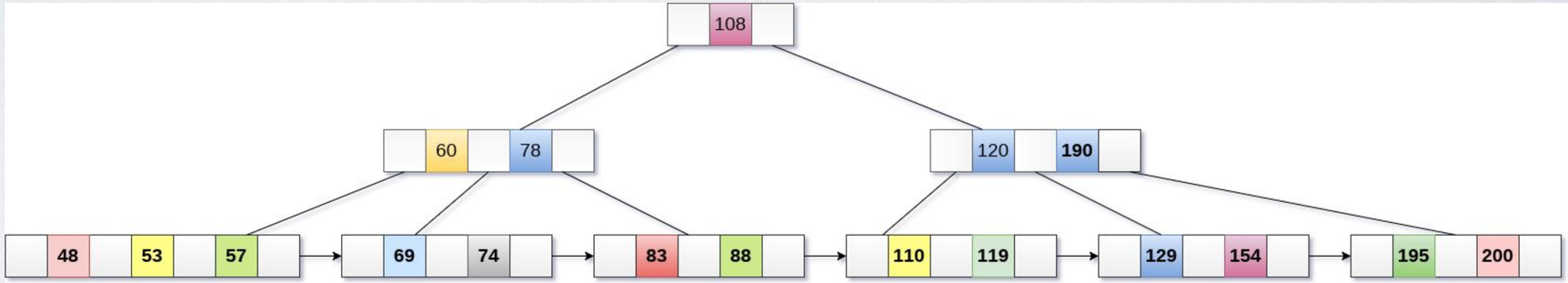
```
CREATE FUNCTION Sales.ufn_SalesByStore (@storeid int)
RETURNS TABLE
AS
RETURN
(
    SELECT P.ProductID, P.Name, SUM(SD.LineTotal) AS 'Total'
    FROM Production.Product AS P
    JOIN Sales.SalesOrderDetail AS SD ON SD.ProductID = P.ProductID
    JOIN Sales.SalesOrderHeader AS SH ON SH.SalesOrderID = SD.SalesOrderID
    JOIN Sales.Customer AS C ON SH.CustomerID = C.CustomerID
    WHERE C.StoreID = @storeid
    GROUP BY P.ProductID, P.Name
);
GO
```

INDEX

- It's used to sort and optimize data fetch time
- Operate similar to index in a book
- When created, an index will create a dynamic Balance Tree (B+ Tree)
- Keys =/= Indexes
- Tables without a Clustered Index are called HEAP Tables
- Indexes can use a Max of 16 Columns or 900B of data

BALANCE TREE

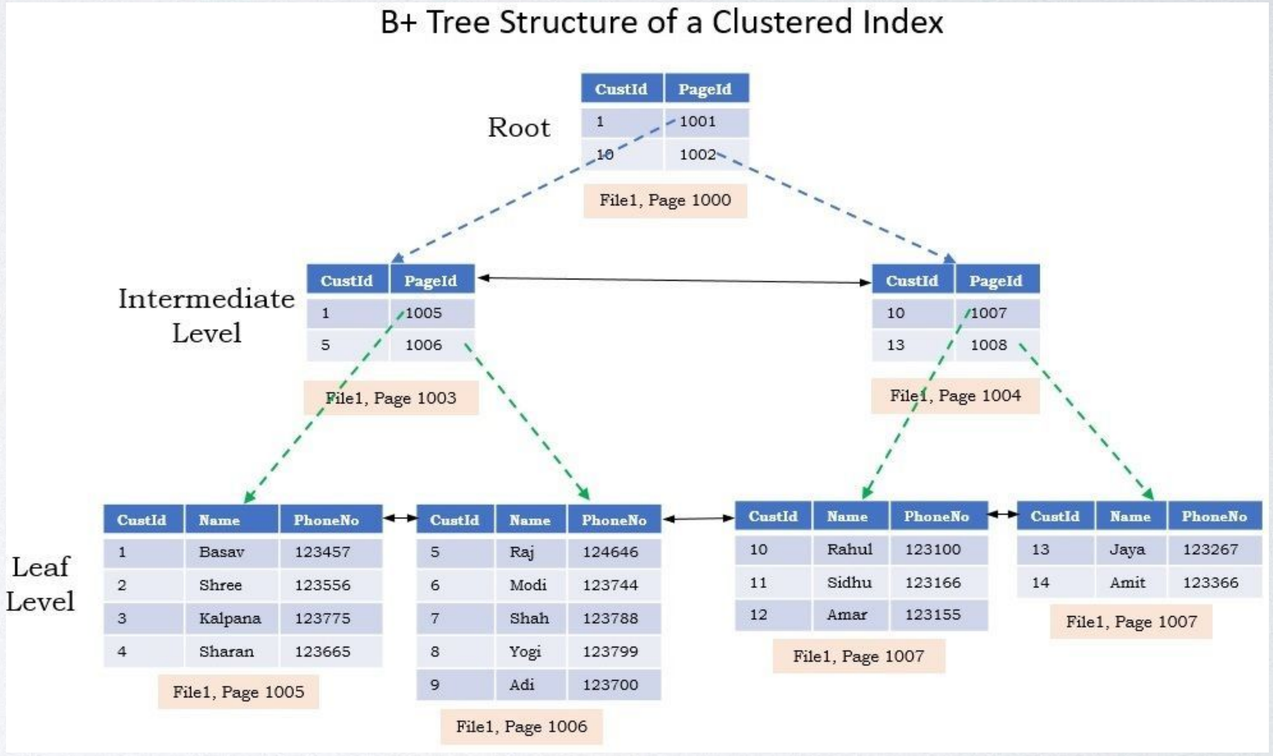
- Composed of 3 main levels
  - Root Level
  - Intermediate Level
  - Leaf Page Level
- Each Node is about 8KB in size
  - 8060B for data
  - 132B for pointers
  - 8192B in Total



- Each Index created will have a Balance tree structure to be used, but the type of Index will determine how data is stored in a Balance Tree
- Clustered Indexes will stored data in Leaf Pages and sort them based on the Key values of the column you choose.
- Non-Clustered Indexes will NOT store data in the Leaf Pages, instead they'll point to the rows they're referencing

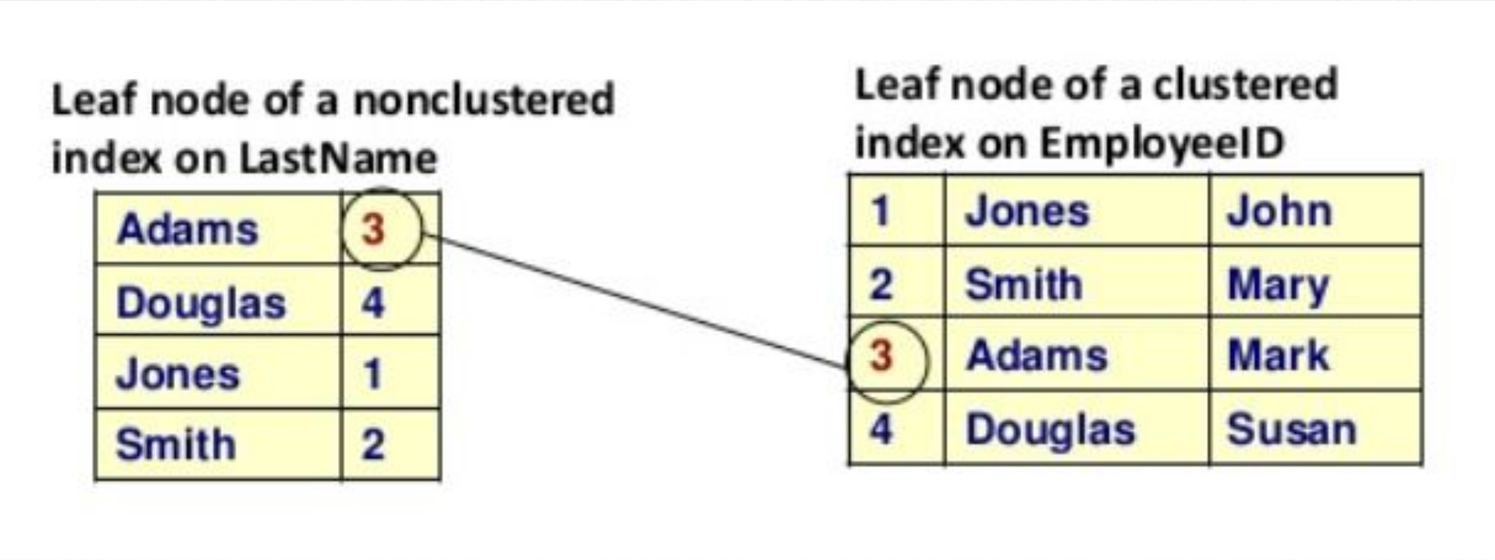
CLUSTERED INDEX

- A clustered index will physically move the data from the table into it's Balance Tree
- The data is now matching physically and logically
- Data is sorted based on ascending order for the column chosen, this becomes the clustering key
- This is why there can only be 1 Clustered Index on a table, data can only be physically sorted and stored once



NON-CLUSTERED INDEX

- Since Non-Clustered Indexes do not physically move or store data, there can be many on a single table.
  - Currently up to 999 different Indexes
- A Non-Clustered Index on a table with a Clustered Index must now grab data from the B-Tree of the CI.
- So data will come up through the Root of the CI and fall into the Leaf Pages of the NCI



Questions?