JAVA FULL STACK PROGRAM

Session: SQL Day 2

OUTLINE

Our First Query

- **Given the following relations:**
  - LIKES (DRINKER, COFFEE)
  - FREQUENTS (DRINKER, CAFE)
  - SERVES (CAFE, COFFEE)
- **Who goes to a cafe serving Cold Brew?**
  - SELECT
  - FROM
  - WHERE

Our First Query

- **Given the following relations:**
  - LIKES (DRINKER, COFFEE)
  - FREQUENTS (DRINKER, CAFE)
  - SERVES (CAFE, COFFEE)
- **Who goes to a cafe serving Cold Brew?**
  - SELECT DISTINCT f.DRINKER
  - FROM FREQUENTS AS f JOIN SERVES AS s ON f.CAFE = s.CAFE
  - WHERE s.COFFEE = 'Cold Brew'
  - **? What happens without DISTINCT?**

AS

```
SELECT DISTINCT f.DRINKER

FROM FREQUENTS AS f JOIN SERVES AS s ON f.CAFE = s.CAFE

WHERE s.COFFEE = 'Cold Brew'
```

What does AS do?

- The AS command is used to rename a column or table with an alias.
- An alias only exists for the duration of the query.
- Is optional
- To create a more meaningful name

JOIN

```
SELECT DISTINCT f.DRINKER

FROM FREQUENTS AS f JOIN SERVES AS s ON f.CAFE = s.CAFE

WHERE s.COFFEE = 'Cold Brew'
```

What kind of join is this?

JOIN

```
SELECT DISTINCT f.DRINKER

FROM FREQUENTS AS f JOIN SERVES AS s ON f.CAFE = s.CAFE

WHERE s.COFFEE = 'Cold Brew'
```

Can also be written

```
SELECT DISTINCT f.DRINKER

FROM FREQUENTS AS f, SERVES AS s

WHERE f.CAFE = s.CAFE AND s.COFFEE = 'Cold Brew'
```

SELECT - FROM - WHERE

SELECT <attribute list>

FROM <tables>

WHERE <conditions>

```
SELECT f.*

FROM FREQUENTS f
```

DRINKER	CAFE
Chris	Double Trouble
Chris	Tout Suite
Risa	Java Lava
Risa	Double Trouble

SELECT Clauses

Attribute	**Example**	**Explanation**
*	*	All attributes from all relations
<table name>.*	FREQUENTS.*	All the attributes from relation <table name>
<alias name>.*	f.*	All the attributes from the relation aliased to f
Attribute list	d.lastName, d.firstName	Only the specified attributes
<Math equation>	1 + 3	Evaluates the expression
<constant>	‘CPA’ 3	Returns the specified constant

More SELECT Clauses

Attribute	Example	Explanation

<function>	NOW()
	CONCAT(<attribute and string constants)
	COALESCE(<attributes and constants>)
		Current datetime
		Concatenates the values
		Returns the first non-NULL argument

CONCAT

FREQUENTS (DRINKER, CAFE)

```
SELECT CONCAT(f.DRINKER, ' ', f.CAFE) AS goesTo

FROM FREQUENTS AS f
```

Also

```
SELECT f.DRINKER || ' ' || CAFE AS goesTo

FROM FREQUENTS AS f
```

COALESCE

Returns the first non-NULL value in the list

```
COALESCE(ATTRIBUTEA, ATTRIBUTEB, 'UNKNOWN')
```

FROM Clause

- List Relation(s)/Table(s)/View(s)
- Specify how they are related
- Be explicit!
  - (otherwise you get the Cartesian Product / Cross Join)

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.

BASIC TYPES OF JOINS

- Inner Join
  - Connect on only matching data
  - Will only display values that match on both sides of the table, all others are excluded
- Left Join
  - Display 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
- 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

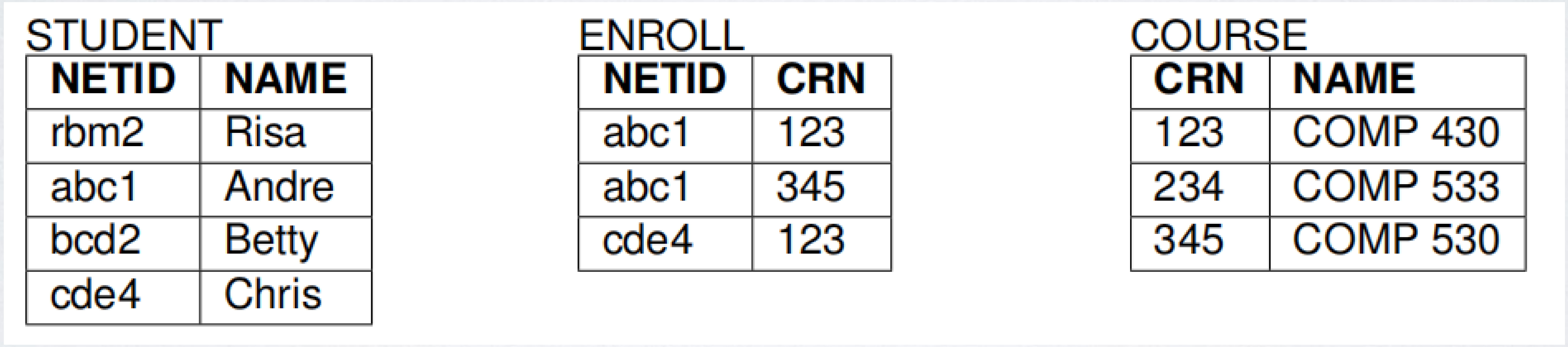
- Used to match up tuples from different relations
- Includes only the relations with matching attribute values

INNER JOIN

COURSE (CRN, NAME)

ENROLL (NETID, CRN)

STUDENT (NETID,NAME)



STUDENT

NETID	NAME
rbm2	Risa
abc1	Andre
bcd2	Betty
cde4	Chris

ENROLL

NETID	CRN
abc1	123
abc1	345
cde4	123

COURSE

CRN	NAME
123	COMP 430
234	COMP 533
345	COMP 530

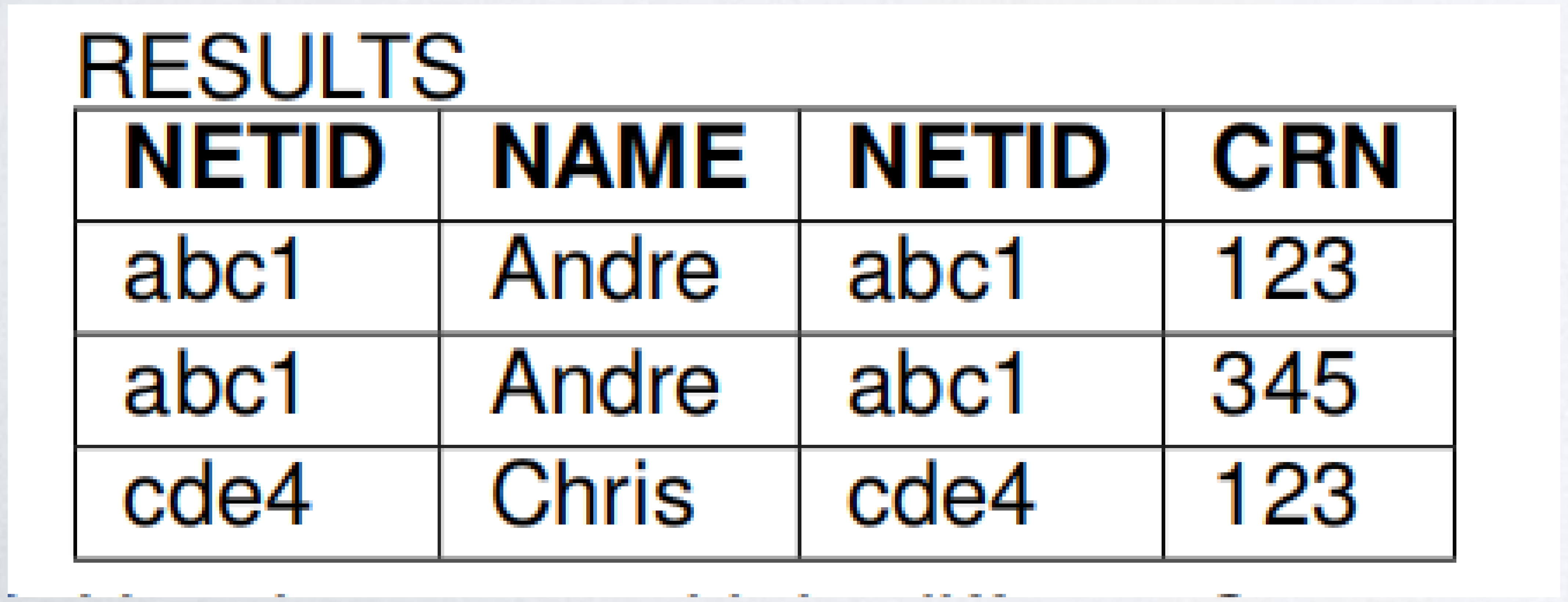
? Who has enrolled in a course, and which course(s)?

INNER JOIN

- COURSE (CRN, NAME)
- ENROLL (NETID, CRN)
- STUDENT (NETID, NAME)
- **? Who has enrolled in a course, and which course(s)?**

SELECT *

FROM STUDENT s INNER JOIN ENROLL e ON s.NETID = e.NETID



RESULTS

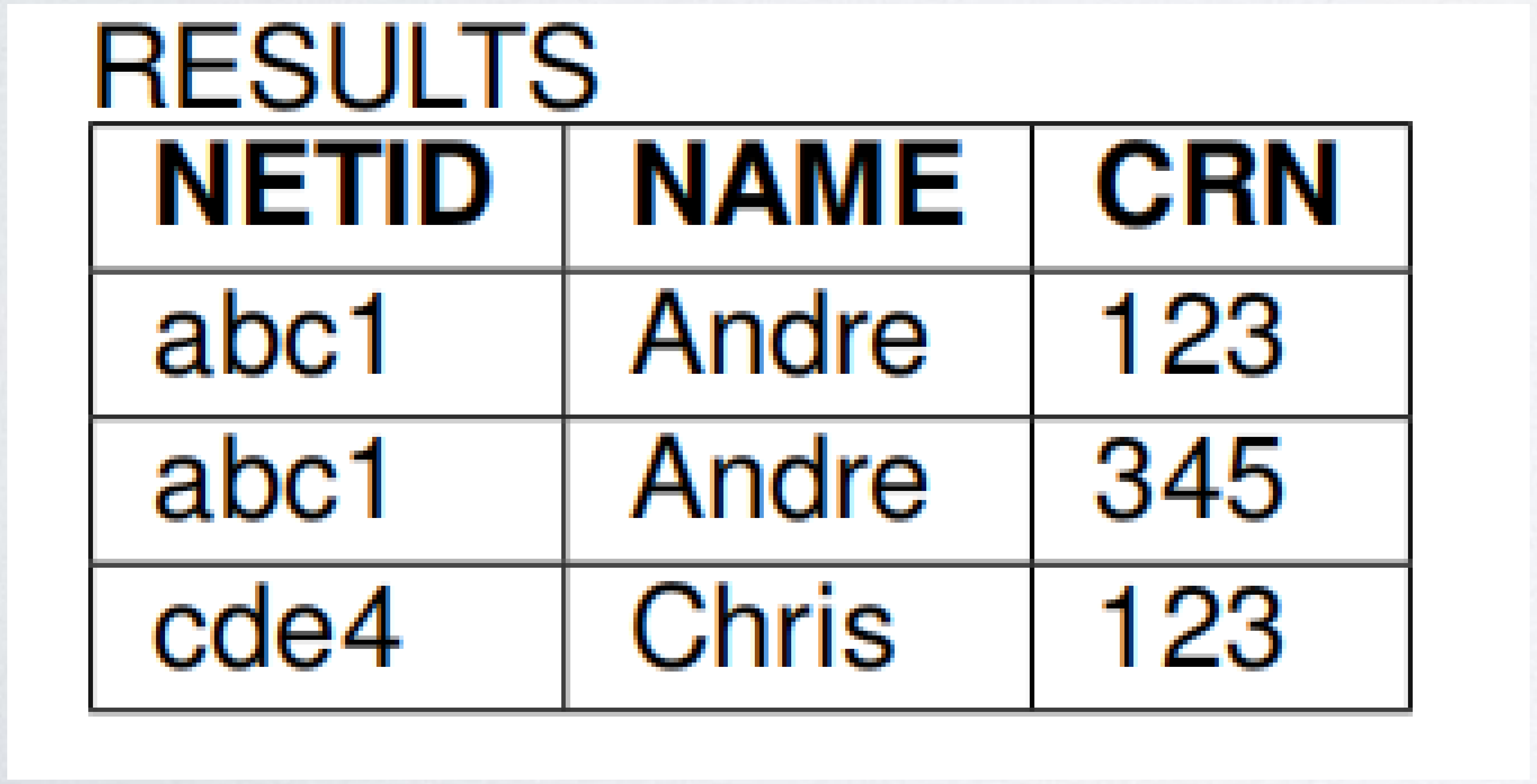
NETID	NAME	NETID	CRN
abc1	Andre	abc1	123
abc1	Andre	abc1	345
cde4	Chris	cde4	123

How is a natural join different?

NATURAL JOIN

SELECT *

FROM STUDENT s NATURAL JOIN ENROLL e



RESULTS

NETID	NAME	CRN
abc1	Andre	123
abc1	Andre	345
cde4	Chris	123

LEFT/RIGHT OUTER JOIN

R LEFT OUTER JOIN S ON <condition>

R RIGHT OUTER JOIN S ON <condition>

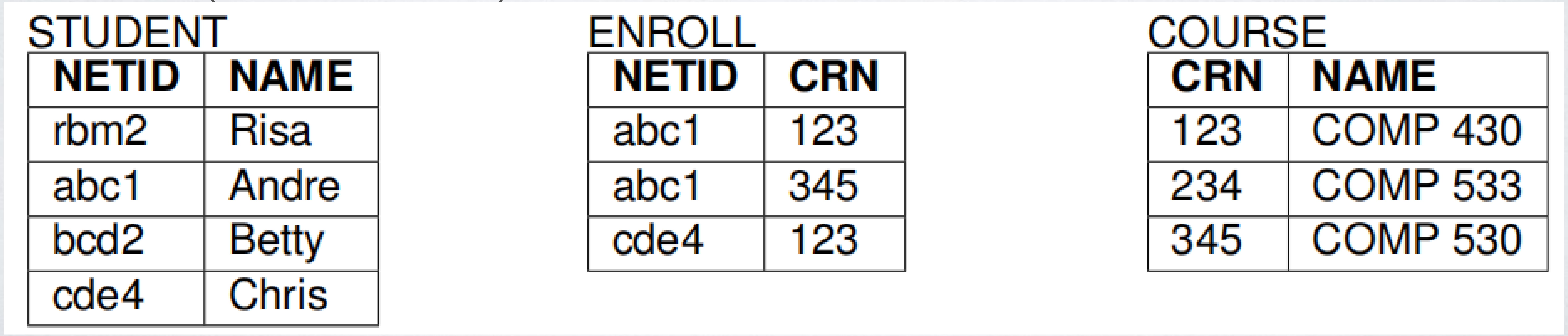
- Used to match up tuples from different relations
- Includes all the relations from the "outer" side
- If there is no matching tuple, assigns NULLs
- Tip: Pick one direction and use it consistently

Left Outer Join

COURSE (CRN, NAME)

ENROLL (NETID, CRN)

STUDENT (NETID, NAME)



STUDENT

NETID	NAME
rbm2	Risa
abc1	Andre
bcd2	Betty
cde4	Chris

ENROLL

NETID	CRN
abc1	123
abc1	345
cde4	123

COURSE

CRN	NAME
123	COMP 430
234	COMP 533
345	COMP 530

Which students haven’t enrolled in any courses?

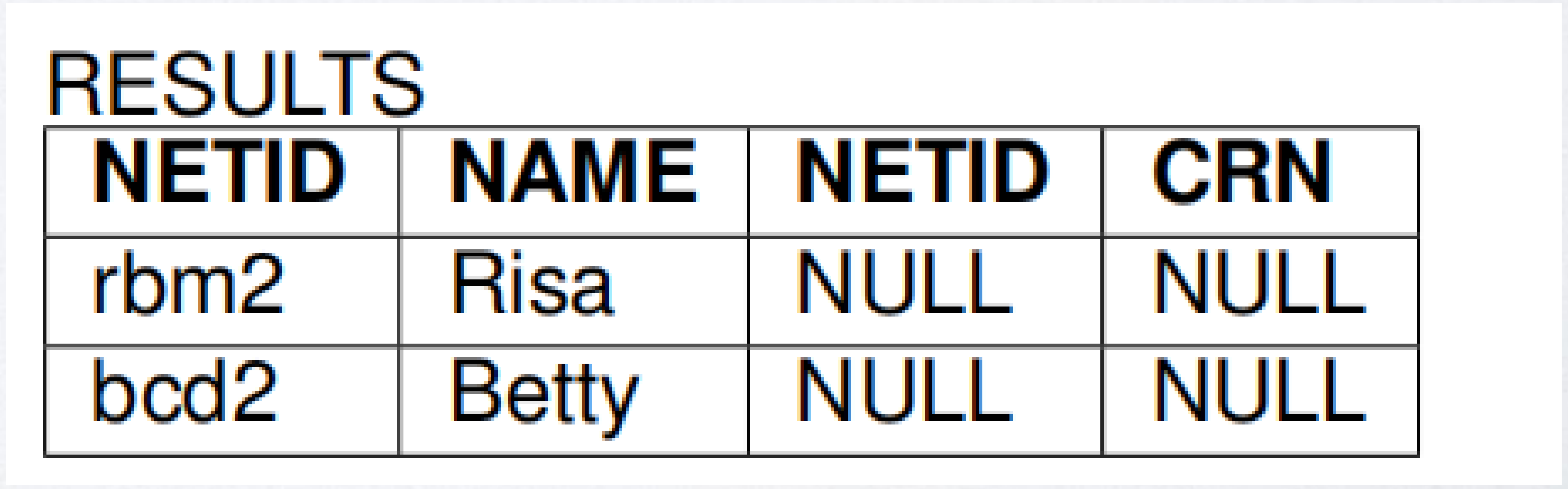
Left Outer Join

- COURSE (CRN, NAME)
- ENROLL (CRN, NETID)
- STUDENT (NETID, NAME)
- **Which students haven't enrolled in any courses?**

```
SELECT *

FROM STUDENT s LEFT OUTER JOIN ENROLL e ON s.NETID = e.NETID

WHERE e.CRN IS NULL
```



RESULTS

NETID	NAME	NETID	CRN
rbm2	Risa	NULL	NULL
bcd2	Betty	NULL	NULL

Right Outer Join

COURSE (CRN, NAME)

ENROLL (NETID, CRN)

STUDENT (NETID, NAME)

? What question does this query answer?

```
SELECT *

FROM ENROLL e RIGHT OUTER JOIN COURSE c ON e.CRN = c.CRN

WHERE e.CRN IS NULL
```

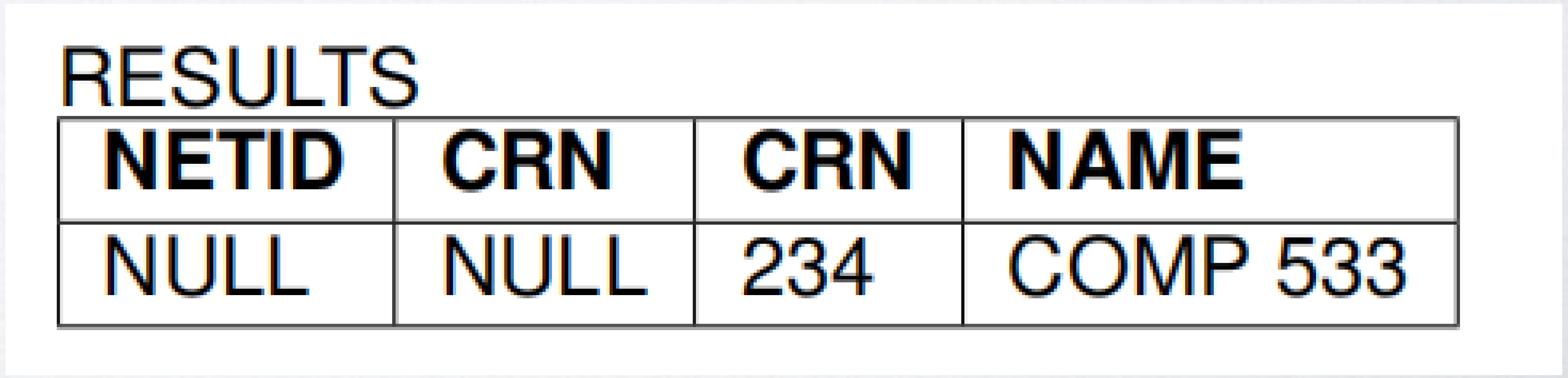
Right Outer Join

- COURSE (CRN, NAME)
- ENROLL (NETID, CRN)
- STUDENT (NETID, NAME)
- **? What question does this query answer?**

```
SELECT *

FROM ENROLL e RIGHT OUTER JOIN COURSE c ON e.CRN = c.CRN

WHERE e.CRN IS NULL
```



RESULTS

NETID	CRN	CRN	NAME
NULL	NULL	234	COMP 533

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

Full Outer Join

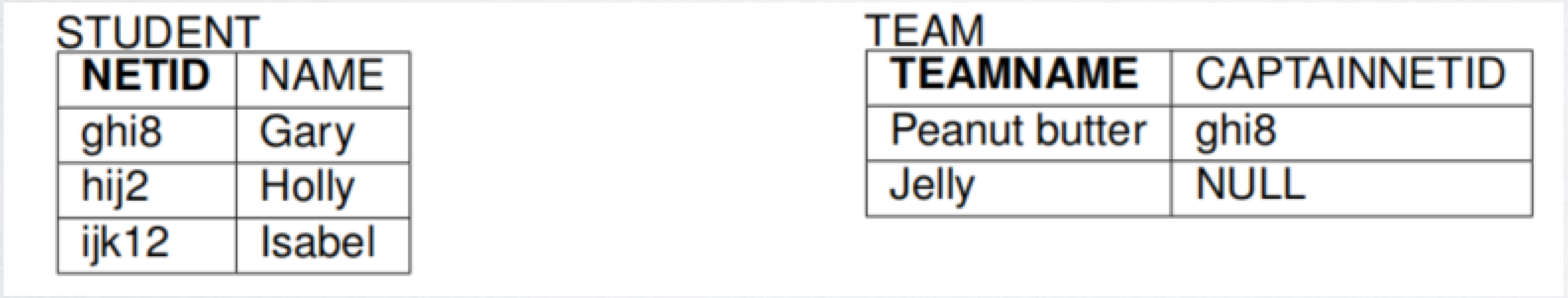
```
SELECT * FROM t1
LEFT JOIN t2 ON t1.id = t2.id
UNION
SELECT * FROM t1
RIGHT JOIN t2 ON t1.id = t2.id
```

On the other hand, if you wanted to see duplicates for some reason, you could use UNION ALL.

```
SELECT * FROM t1
LEFT JOIN t2 ON t1.id = t2.id
UNION ALL
SELECT * FROM t1
RIGHT JOIN t2 ON t1.id = t2.id
WHERE t1.id IS NULL
```

Full Outer Join

```
SELECT s.NAME, t.TEAMNAME
FROM STUDENT s LEFT OUTER JOIN TEAM t ON s.NETID = t.CAPTAINNETID
UNION
SELECT s.NAME, t.TEAMNAME
FROM STUDENT s RIGHT OUTER JOIN TEAM t ON s.NETID = t.CAPTAINNETID
```



STUDENT

NETID	NAME
ghi8	Gary
hij2	Holly
ijk12	Isabel

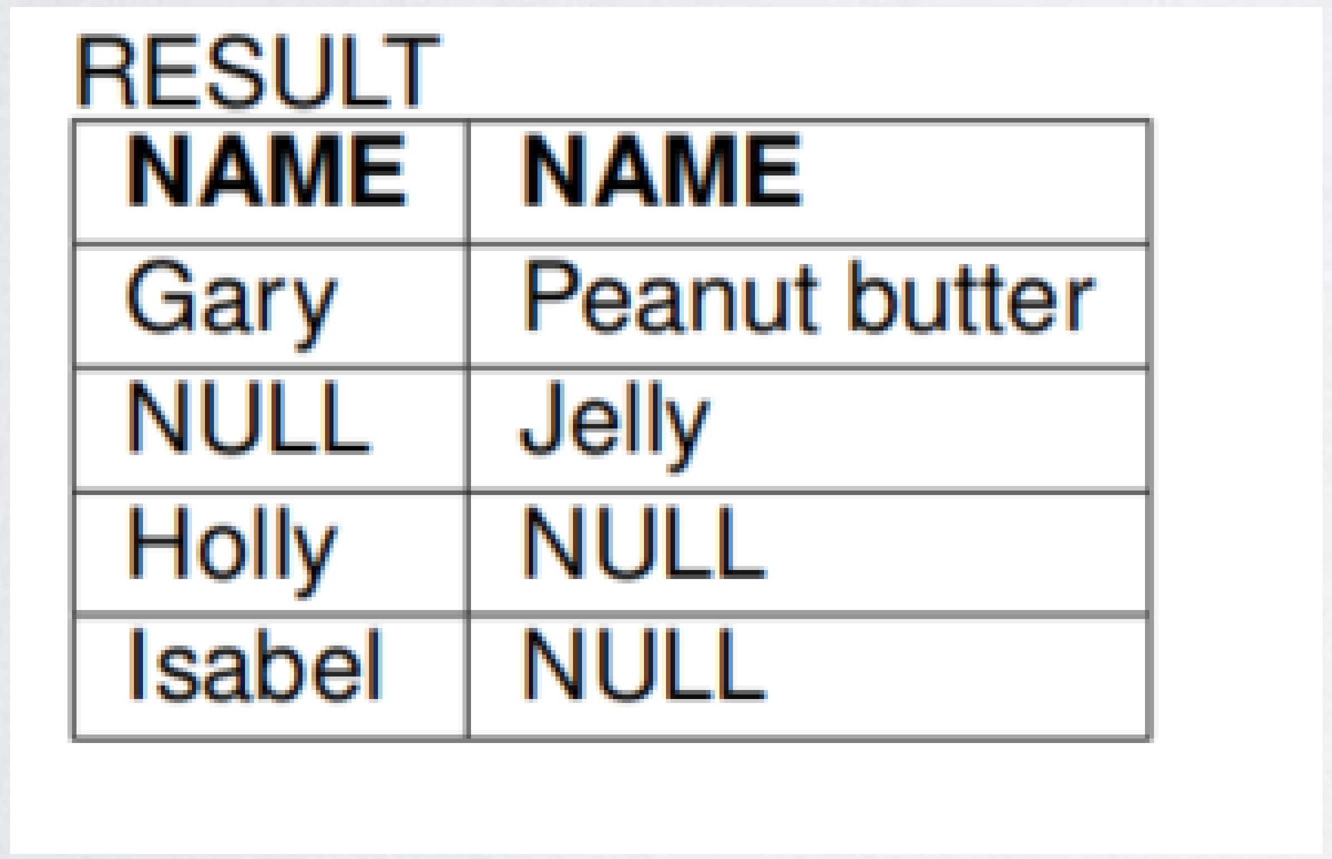
TEAM

TEAMNAME	CAPTAINNETID
Peanut butter	ghi8
Jelly	NULL

What does this expression represent?

Full Outer Join

```
SELECT s.NAME, t.TEAMNAME
FROM STUDENT s LEFT OUTER JOIN TEAM t ON s.NETID = t.CAPTAINNETID
UNION
SELECT s.NAME, t.TEAMNAME
FROM STUDENT s RIGHT OUTER JOIN TEAM t ON s.NETID = t.CAPTAINNETID
```



RESULT

NAME	NAME
Gary	Peanut butter
NULL	Jelly
Holly	NULL
Isabel	NULL

Self Join

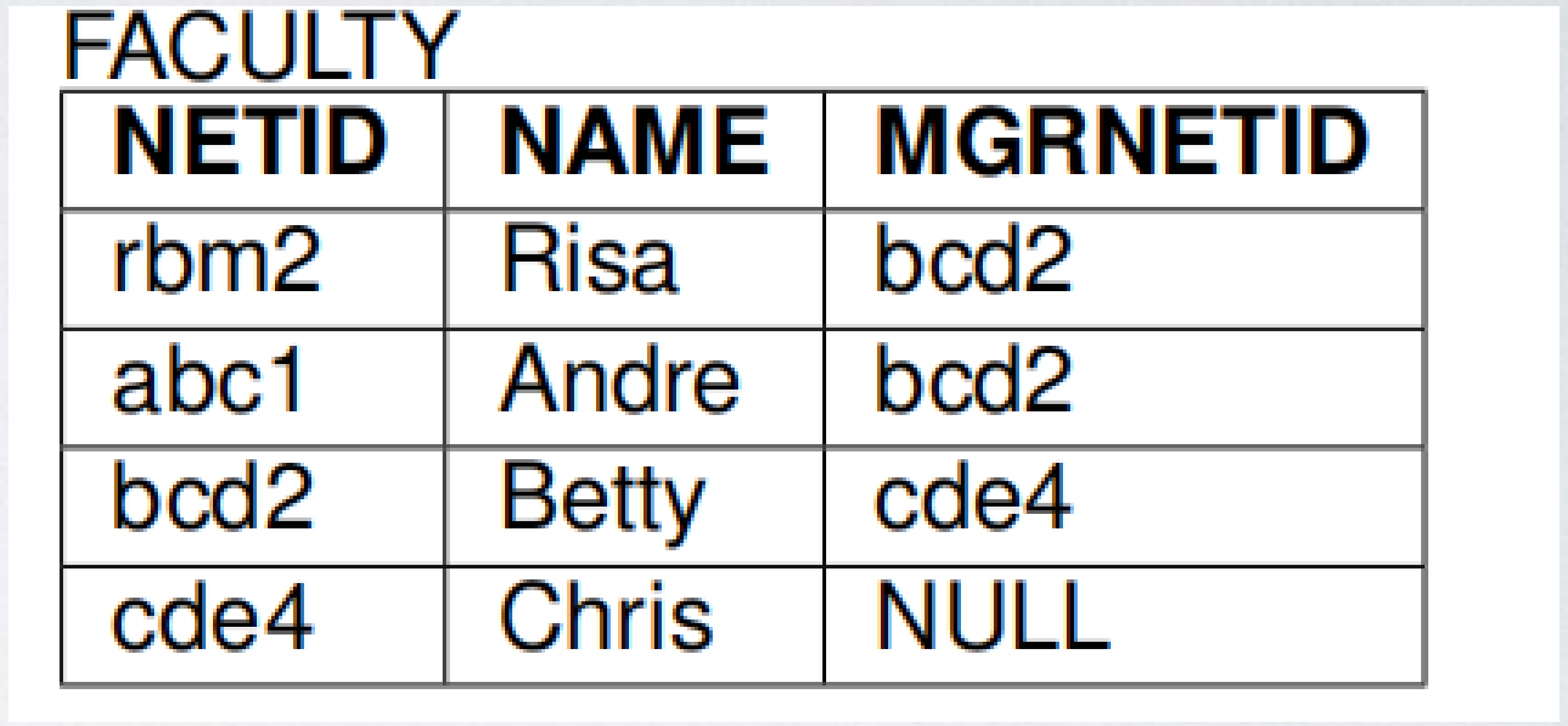
R AS R1 JOIN R AS R2 ON R1.<att> <op> R2.<att>

- 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

Self Join

FACULTY (NETID, NAME, MGRNETID)

FACULTY



FACULTY

NETID	NAME	MGRNETID
rbm2	Risa	bcd2
abc1	Andre	bcd2
bcd2	Betty	cde4
cde4	Chris	NULL

```
SELECT f.NAME, Mgr.Name
FROM FACULTY f JOIN FACULTY Mgr ON f.MGRNETID = Mgr.NETID
```

What does this expression represent?

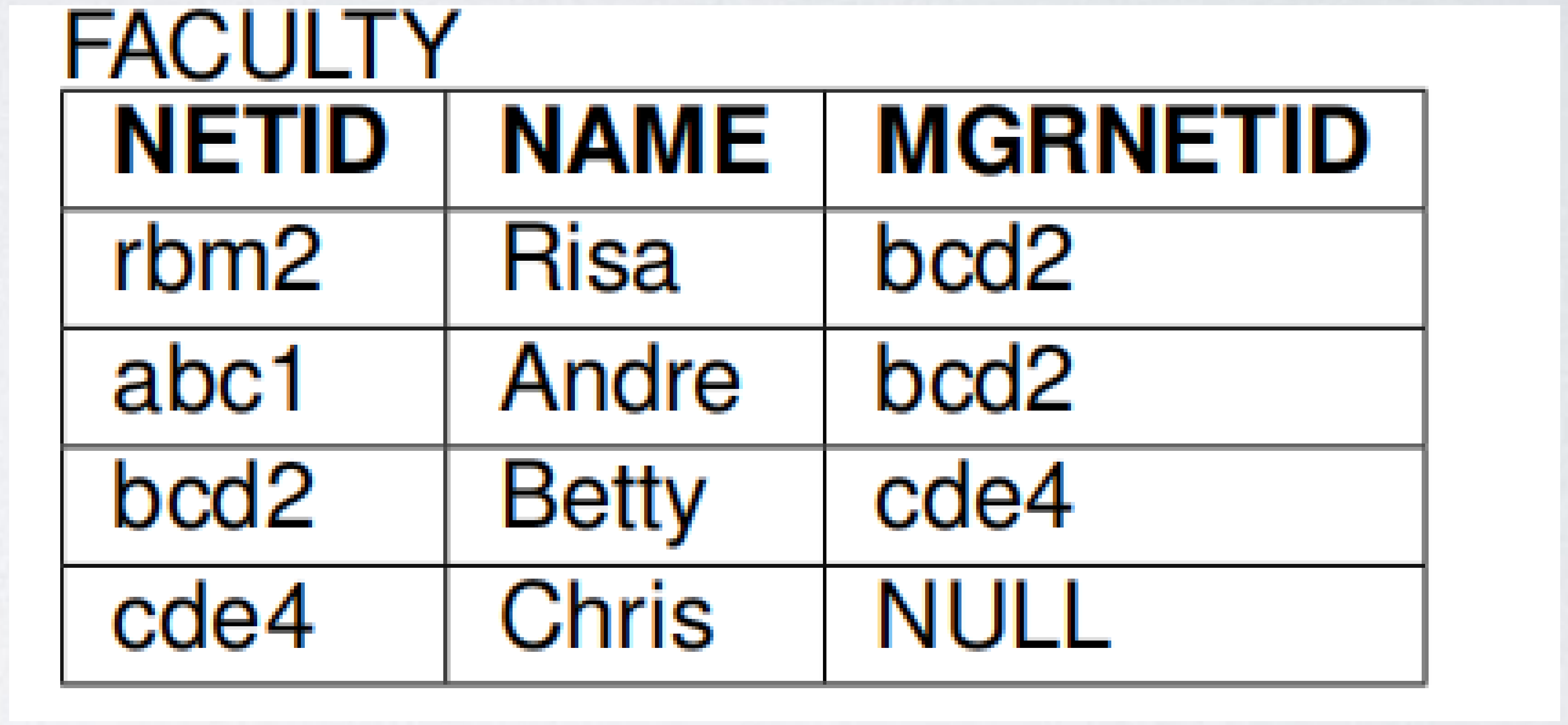
A Every faculty member paired with every manager

B Every faculty member who has a manager, paired with that manager

Self Join

FACULTY (NETID, NAME, MGRNETID)

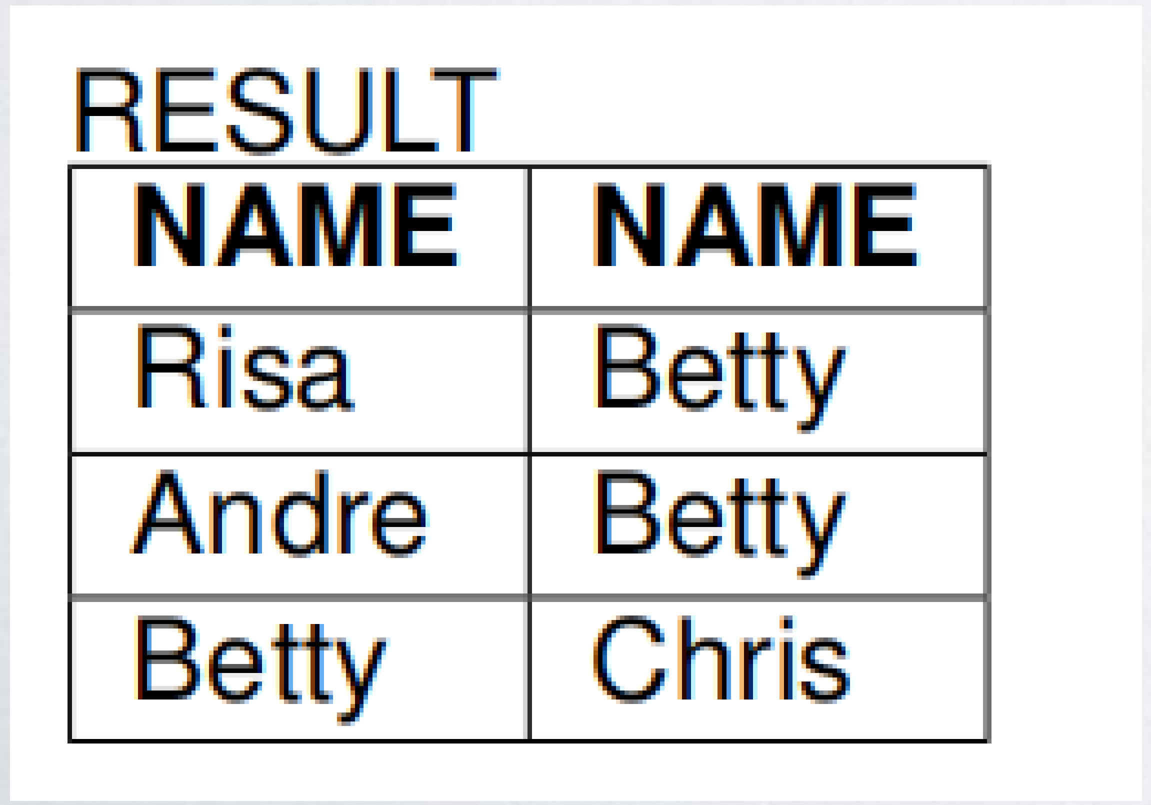
FACULTY



FACULTY

NETID	NAME	MGRNETID
rbm2	Risa	bcd2
abc1	Andre	bcd2
bcd2	Betty	cde4
cde4	Chris	NULL

```
SELECT f.NAME, Mgr.Name
FROM FACULTY f JOIN FACULTY Mgr ON f.MGRNETID = Mgr.NETID
```

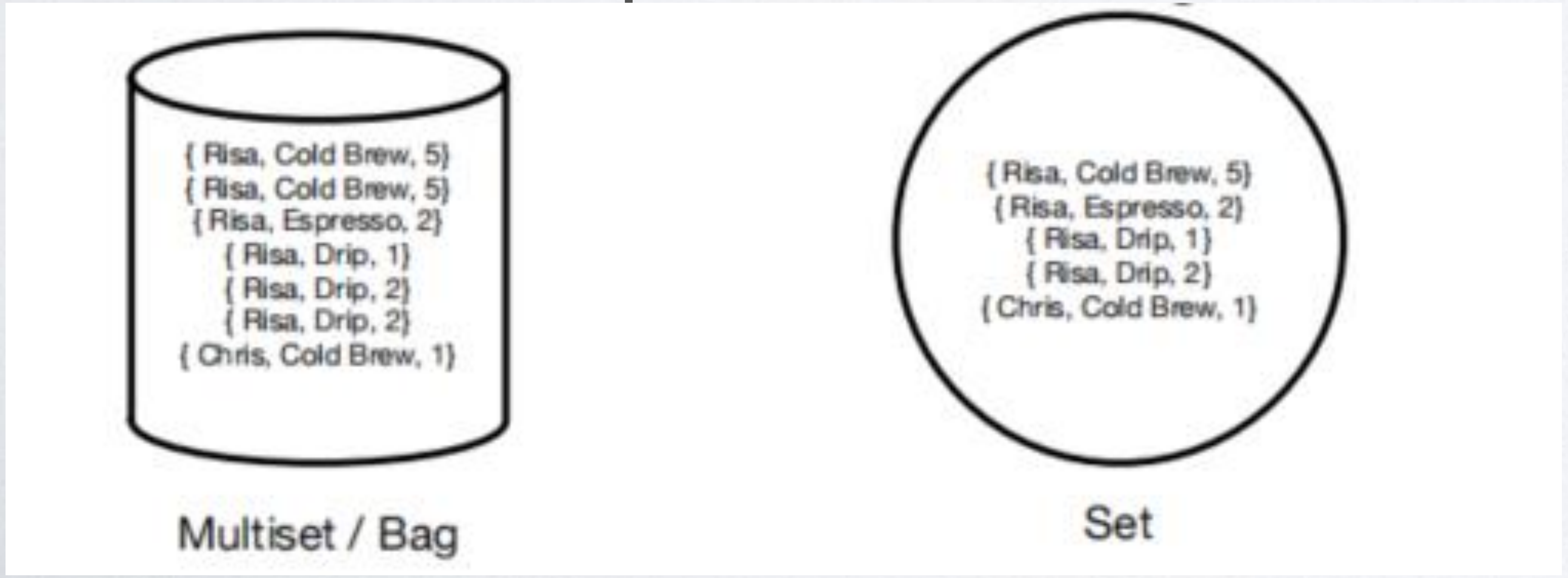


RESULT

NAME	NAME
Risa	Betty
Andre	Betty
Betty	Chris

SET OPERATIONS

- Results are unordered multisets/bag
- It could be useful to perform operations on these
  - Union
  - Intersection
  - Difference
- Different RDBMs provide different levels of support



UNION and UNION ALL

UNION- eliminates duplicates

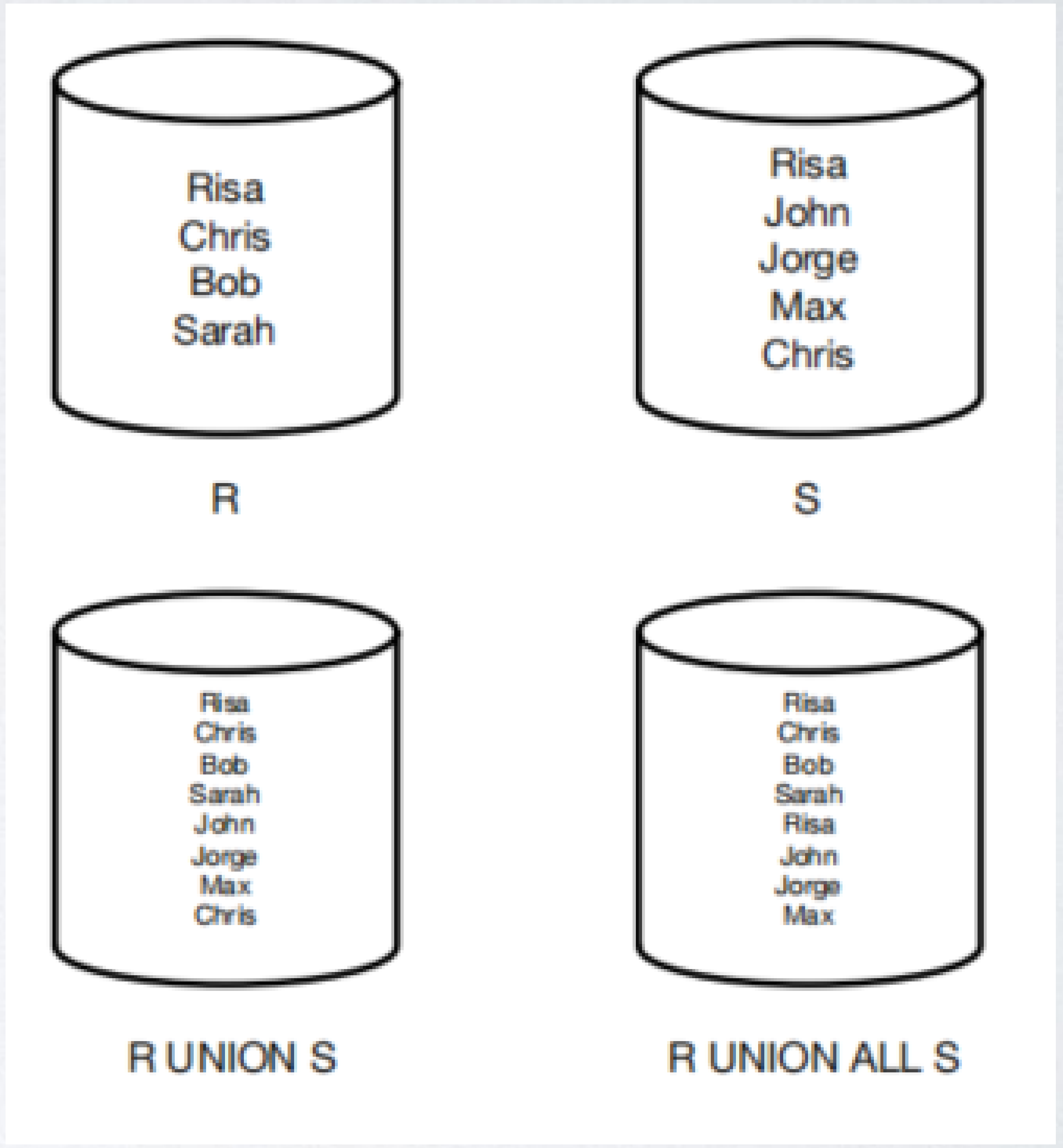
UNION ALL- does NOT eliminate

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
  - Display the values in the first select statement MINUS any values found in the second select statement

SELECT - FROM - WHERE

SELECT <attribute list>

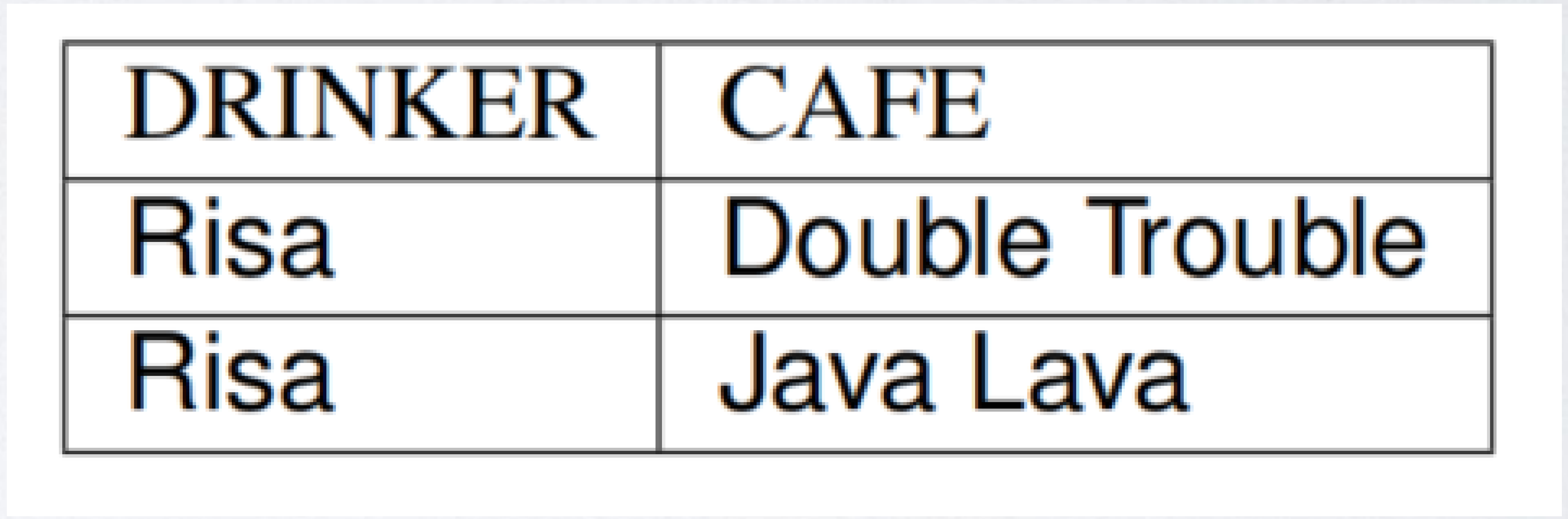
FROM <tables>

WHERE <conditions>

SELECT *

FROM FREQUENTS f

WHERE f.drinker = 'Risa'



DRINKER	CAFE
Risa	Double Trouble
Risa	Java Lava

WHERE Clause

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 …

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 l2.COFFEE

FROM LIKES l2

WHERE l2.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 l.DRINKER
FROM LIKES l
WHERE NOT EXISTS (
        SELECT l2.COFFEE
        FROM LIKES l2
        WHERE l2.DRINKER = 'Risa'
        AND l2.COFFEE NOT IN (
              SELECT l3.COFFEE
              FROM LIKES l3
              WHERE l3.DRINKER = l.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

SOME/ANY Example

Given the relation:

RATES (DRINKER, COFFEE, SCORE)

Ratings go from low to high, with increasing values indicating higher levels of liking the coffee.

Of the coffees Risa has rated, list the coffees that are not Risa's favorite.

SOME/ANY Example

- Given the relation:
- RATES (DRINKER, COFFEE, SCORE)
- **Of the coffees Risa has rated, list the coffees that are not Risa’s favorite**

```
SELECT r.COFFEE
FROM RATES r
WHERE r.DRINKER = 'Risa' AND r.SCORE < SOME (
        SELECT r2.SCORE
        FROM RATES r2
        WHERE r2.DRINKER = 'Risa' )
```

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')
```

? What does this query return?

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’?

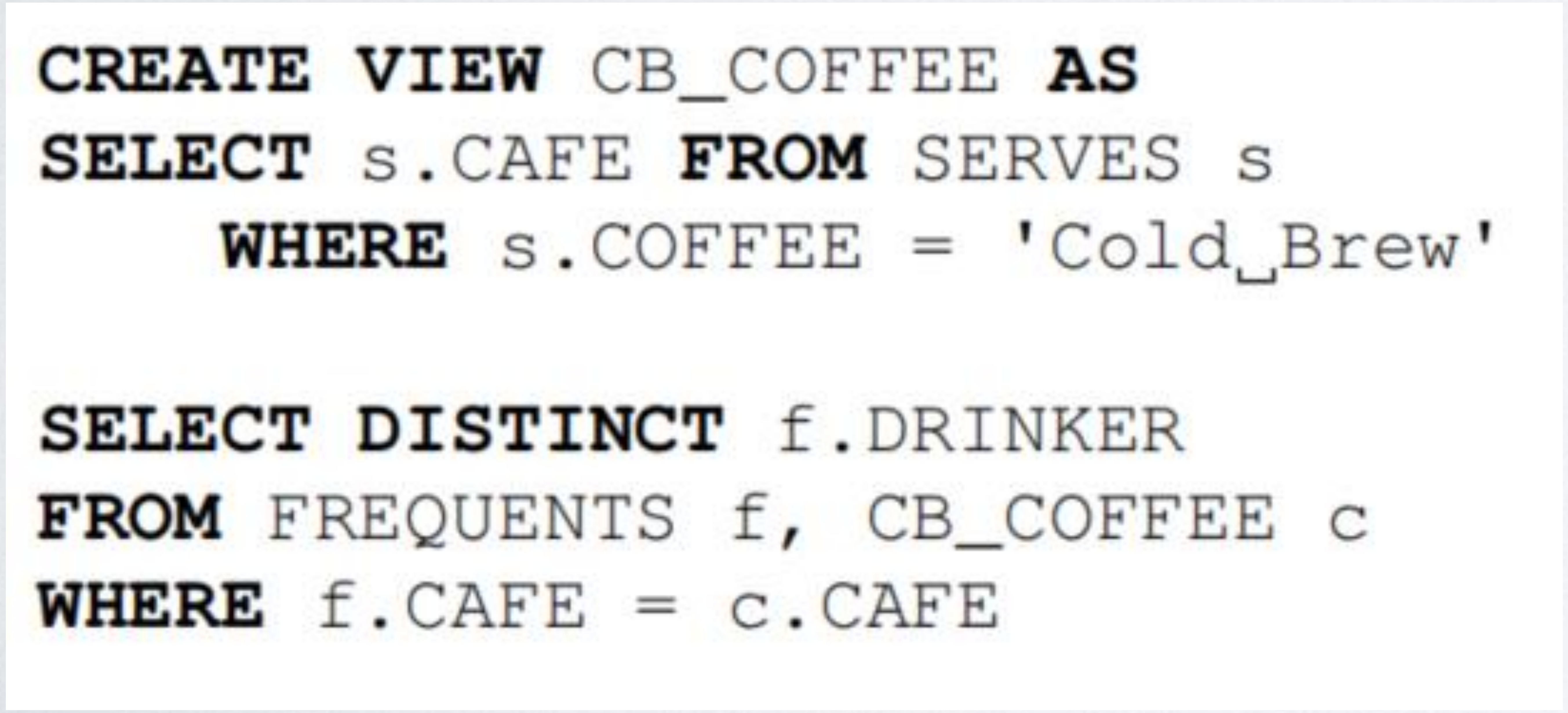
```
SELECT DISTINCT f.DRINKER
FROM FREQUENTS f, SERVES s
WHERE f.CAFE = s.CAFE
        AND s.COFFEE = 'Cold Brew'
```

```
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!

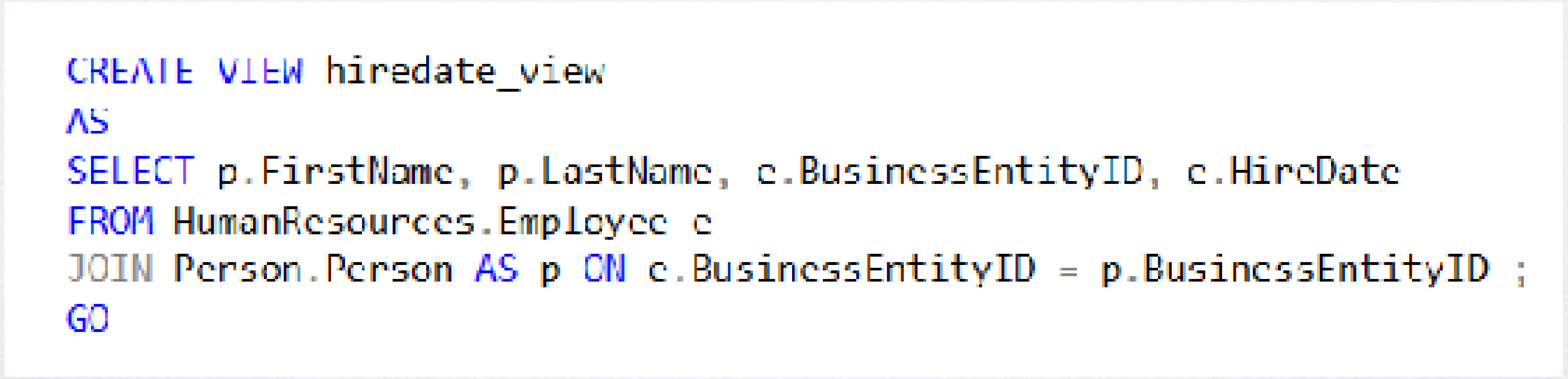


```
CREATE VIEW CB_COFFEE AS
SELECT s.CAFE FROM SERVES s
    WHERE s.COFFEE = 'Cold␣Brew'

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
```

Views

List the coffees that are not Risa’s favorite

```
CREATE VIEW RISA_COFFEES AS
SELECT *
FROM RATES r
WHERE r.DRINKER = 'Risa'

SELECT r.COFFEE
FROM RISA_COFFEES r
WHERE r.SCORE < SOME (
    SELECT r2.SCORE
    FROM RISA_COFFEES r2)
```

Aggregations

Can compute simple statistics using built-in SQL functions

- SUM
- AVG
- COUNT
- MAX
- MIN
- etc.

Our First Aggregation

RATES (DRINKER, COFFEE, SCORE)

? What is the average coffee rating given by Risa?

```
SELECT AVG (r.SCORE)

FROM RATES r

WHERE r.DRINKER = 'Risa'
```

COUNT DISTINCT

RATES (DRINKER, COFFEE, SCORE)

How many coffees has Risa rated?

? Does this work?

```
SELECT COUNT (*)
FROM RATES r
WHERE r.DRINKER = 'Risa'
```

COUNT DISTINCT

RATES (DRINKER, COFFEE, SCORE)

Count the number of different types of coffee drinks that Risa has rated

? Does this work?

```
SELECT COUNT (DISTINCT r.COFFEE)

FROM RATES r

WHERE r.DRINKER = 'Risa'
```

GROUP BY

RATES (DRINKER, COFFEE, SCORE)

? What is the average rating for each coffee?

```
SELECT r.COFFEE, AVG (r.SCORE)

FROM RATES r

GROUP BY r.COFFEE
```

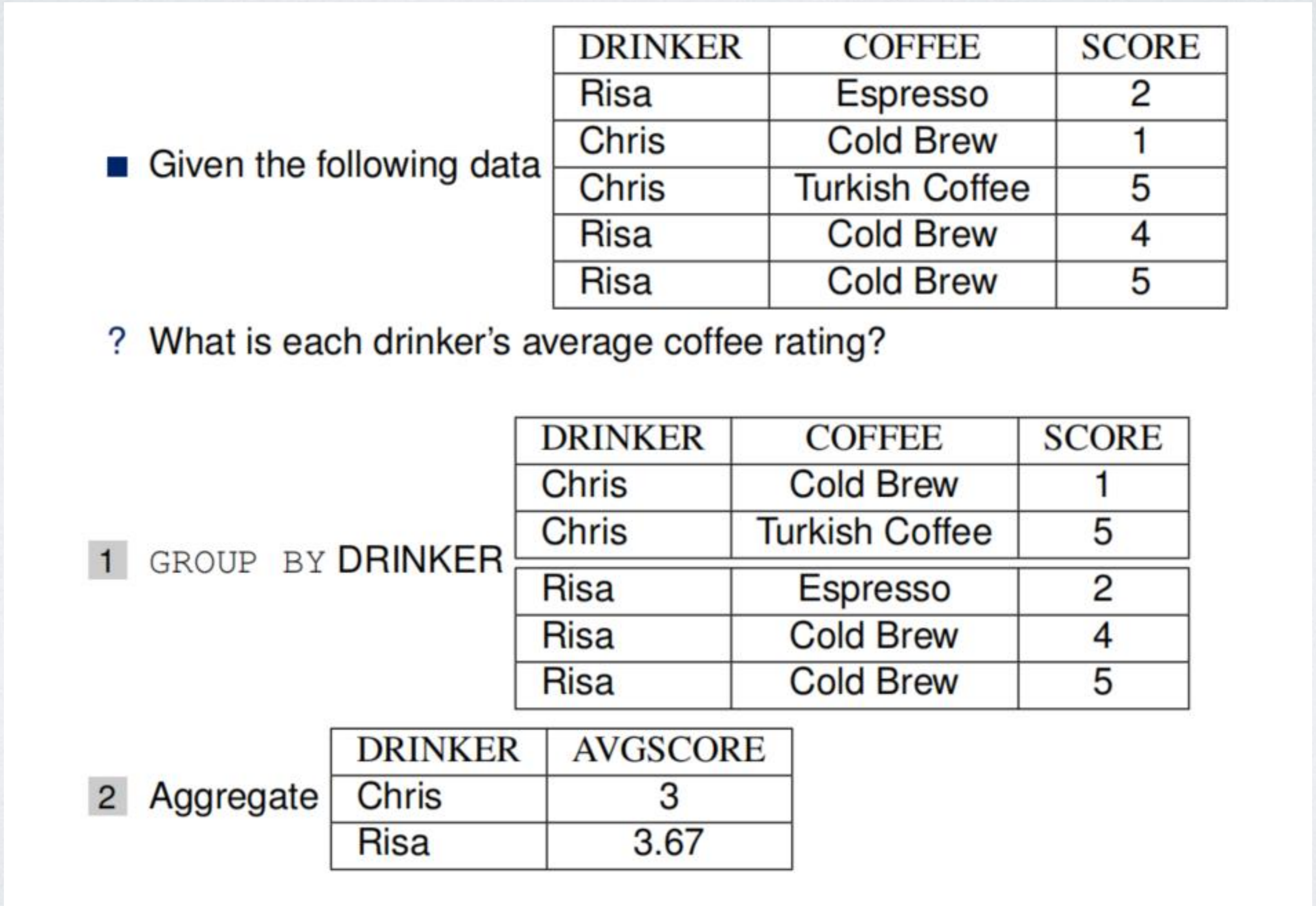
GROUP BY

- RATES (DRINKER, COFFEE, SCORE)
- **? What is the average rating for each coffee?**

```
SELECT r.COFFEE, AVG (r.SCORE)
FROM RATES r
GROUP BY r.COFFEE
```

- Note: If you have an attribute outside of an aggregate function in an
- aggregate query
- Example: r.COFFEE here
- Then you must have grouped by that attribute
- Or the query will not compile

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

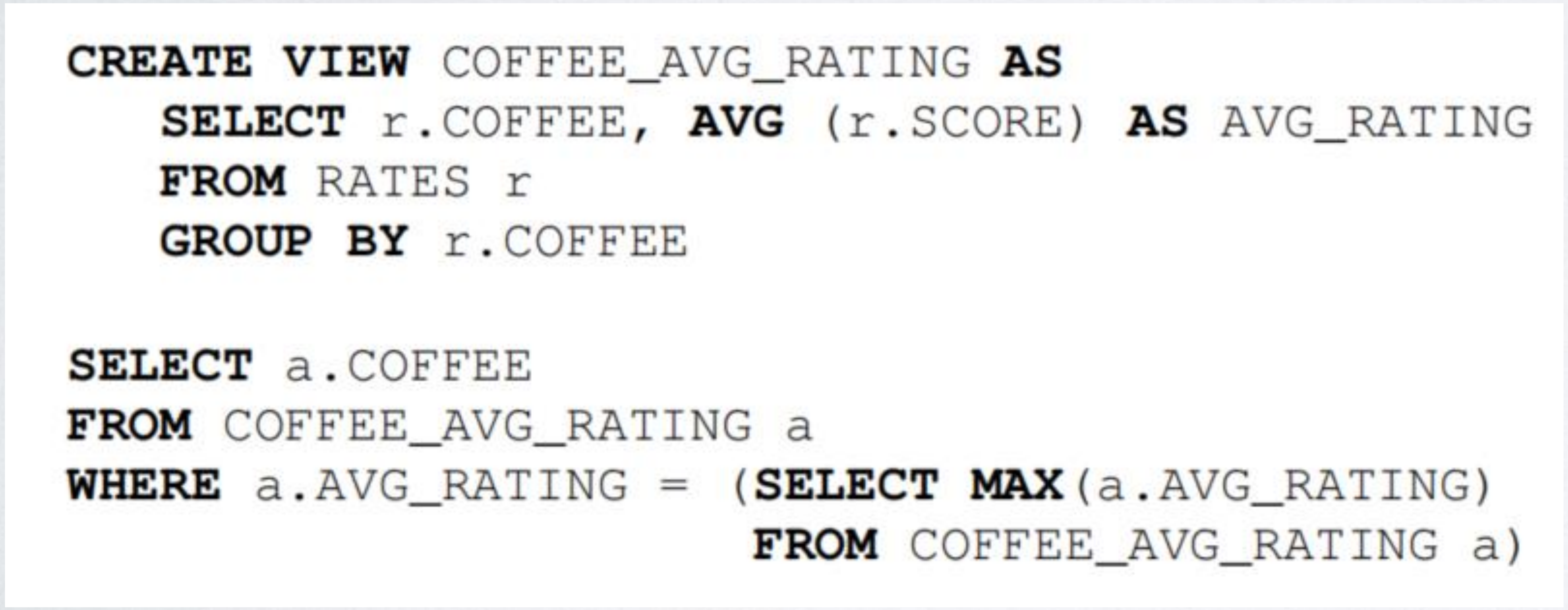
2 Aggregate

DRINKER	AVGSCORE
Chris	3
Risa	3.67

HAVING

RATES (DRINKER, COFFEE, SCORE)

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



```
CREATE VIEW COFFEE_AVG_RATING AS
   SELECT r.COFFEE, AVG (r.SCORE) AS AVG_RATING
   FROM RATES r
   GROUP BY r.COFFEE

SELECT a.COFFEE
FROM COFFEE_AVG_RATING a
WHERE a.AVG_RATING = (SELECT MAX(a.AVG_RATING)
                      FROM COFFEE_AVG_RATING a)
```

What problems here?

HAVING

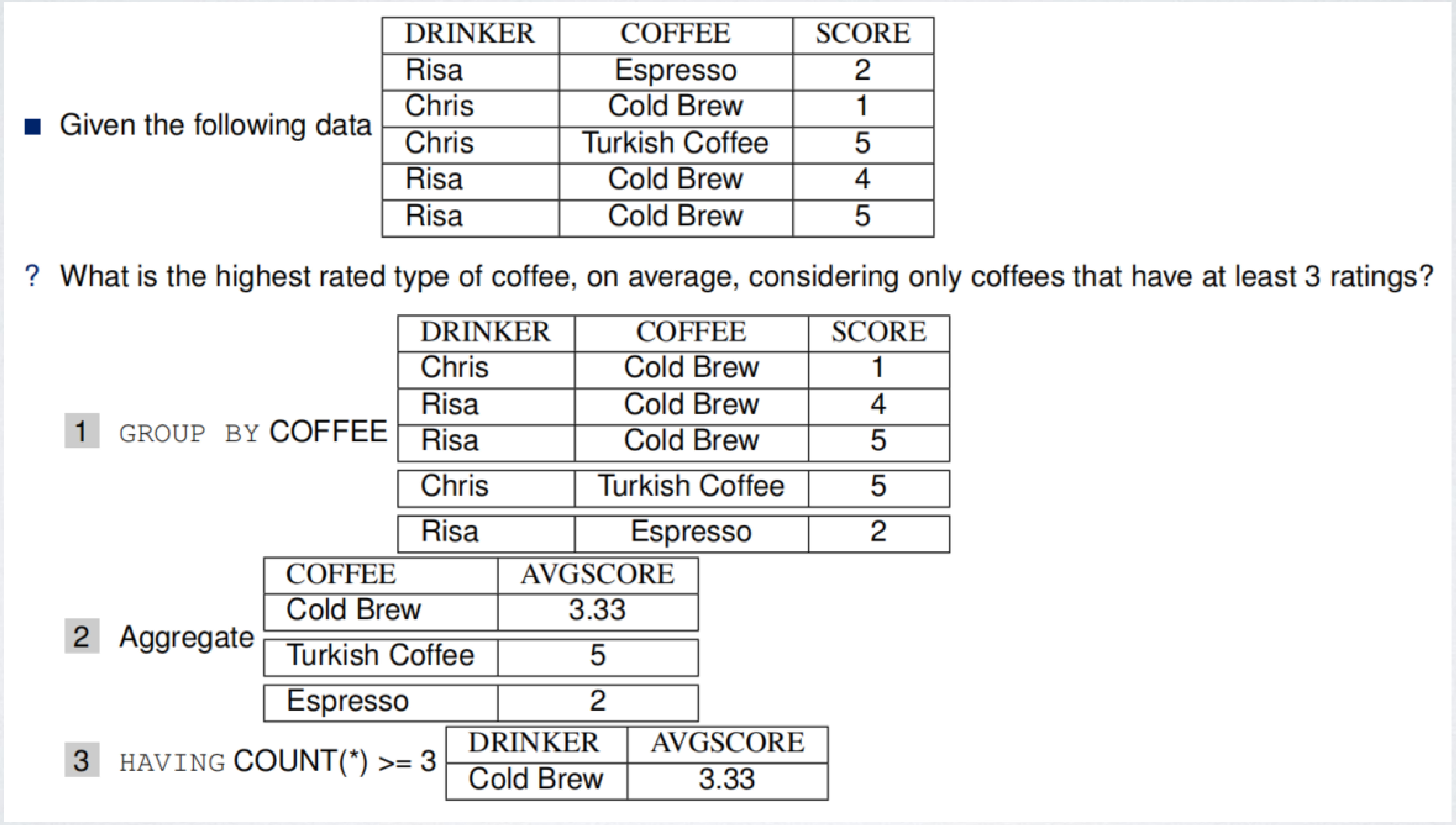
RATES (DRINKER, COFFEE, SCORE)

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

Change COFFEE_AVG_RATING to:

```
CREATE VIEW COFFEE_AVG_RATING AS
SELECT r.COFFEE, AVG(r.SCORE) AS AVG_RATING
FROM RATES r
GROUP BY COFFEE
HAVING COUNT(*) >= 3
```

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

Aggregate Functions & NULL

What about NULL?

- COUNT(1) or COUNT(*) will count _____ row
- COUNT(<attribute>) will count _____ values
- AVG, MIN, MAX, etc. ignore NULL values
- GROUP BY includes a row for NULL

Subquery in FROM Revisited

RATES (DRINKER, COFFEE, SCORE)

What is the highest rated coffee, on average?

```
CREATE VIEW COFFEE_AVG_RATING AS
SELECT r.COFFEE, AVG (r.SCORE) AS AVG_RATING
FROM RATES r
GROUP BY r.COFFEE

SELECT a.COFFEE
FROM COFFEE_AVG_RATING a
WHERE a.AVG_RATING = (SELECT MAX(a.AVG_RATING)
                      FROM COFFEE_AVG_RATING a)
```

TOP K/ LIMIT K/ ORDER BY

RATES (DRINKER, COFFEE, SCORE)

What is the highest rated coffee, on average?

```
CREATE VIEW COFFEE_AVG_RATING AS
SELECT r.COFFEE, AVG (r.SCORE) AS AVG_RATING
FROM RATES r
GROUP BY r.COFFEE

SELECT a.COFFEE
FROM COFFEE_AVG_RATING a
ORDER BY a.AVG_RATING DESC LIMIT 1;
```

TOP K/ LIMIT K/ ORDER BY

ORDER BY

- Can choose ASC or DESC
- Finally: note that ORDER BY can be used without LIMIT

FUNCTIONS

- Functions are used in SQL to perform business calculations
- Can be used in select statement
- Different Database Management Systems support different build-in function
  - MySQL: https://dev.mysql.com/doc/refman/8.0/en/func-op-summary-ref.html
  - Oracle: https://docs.oracle.com/javadb/10.8.3.0/ref/rrefsqlj29026.html
  - SQL Server: https://docs.microsoft.com/en-us/sql/t-sql/functions/functions?view=sql-server-ver15
  - PostgreSQL: https://www.postgresql.org/docs/9.2/functions.html

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';
```

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 Proc’s
    - 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 (DOES NOT SUPPORT 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.

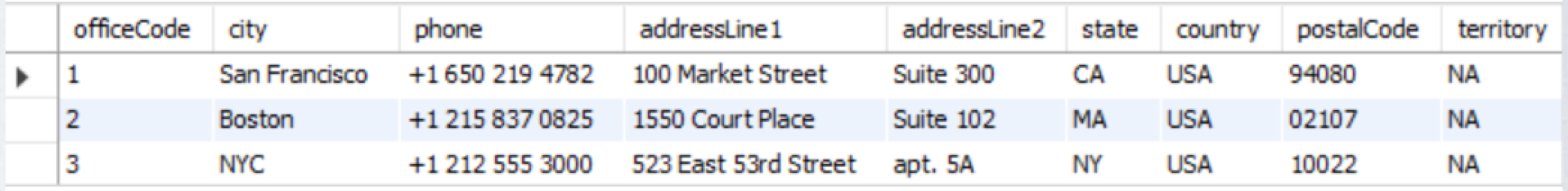
UD STORED PROCEDURE

```sql
DELIMITER //

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

DELIMITER ;
```

```sql
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

```sql
CALL GetOfficeByCountry();
```

Will result in Error

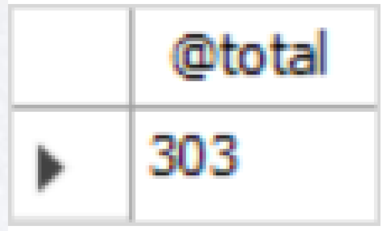
UD STORED PROCEDURE

```sql
DELIMITER $$

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

DELIMITER ;
```

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



	@total
▶	303

QUERY EXECUTION

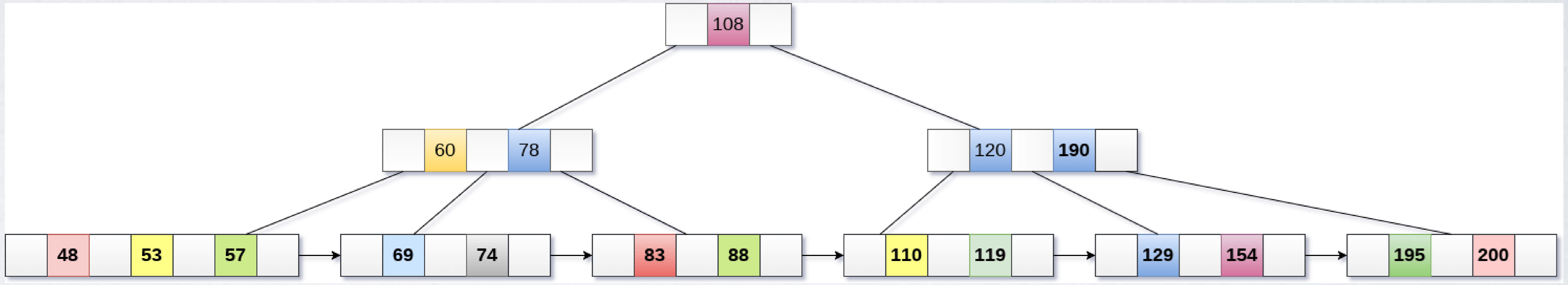
- Step 1:
  - Parser check query syntax
  - Break query to token --> (intermediate files)
- Step 2:
  - Query Optimizer creates best possible execution plan based on current resource utilization
- Step 3:
  - DB engine --> Run the query

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

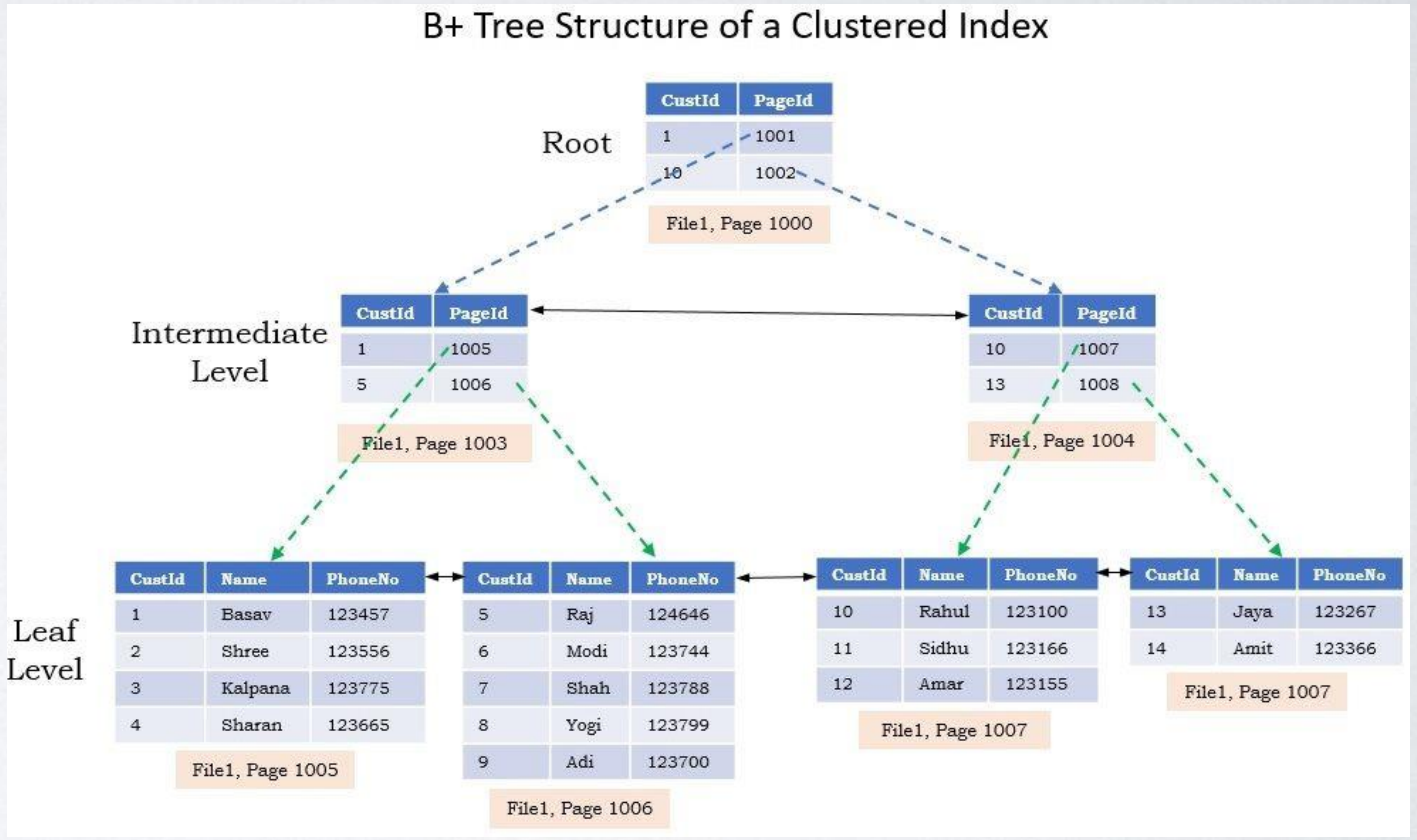
- Composed of 3 main levels
  - Root Level
  - Intermediate Level
  - Leaf Page Level
- Each Node is about 8KB in size
  - 8060B for data
  - 132B for pointers
  - 8192B in Total



- 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



CustId	PageId
1	1001
10	1002

File1, Page 1000

CustId	PageId
1	1005
5	1006

File1, Page 1003

CustId	PageId
10	1007
13	1008

File1, Page 1004

CustId	Name	PhoneNo
1	Basav	123457
2	Shree	123556
3	Kalpana	123775
4	Sharan	123665

File1, Page 1005

CustId	Name	PhoneNo
5	Raj	124646
6	Modi	123744
7	Shah	123788
8	Yogi	123799
9	Adi	123700

File1, Page 1006

CustId	Name	PhoneNo
10	Rahul	123100
11	Sidhu	123166
12	Amar	123155

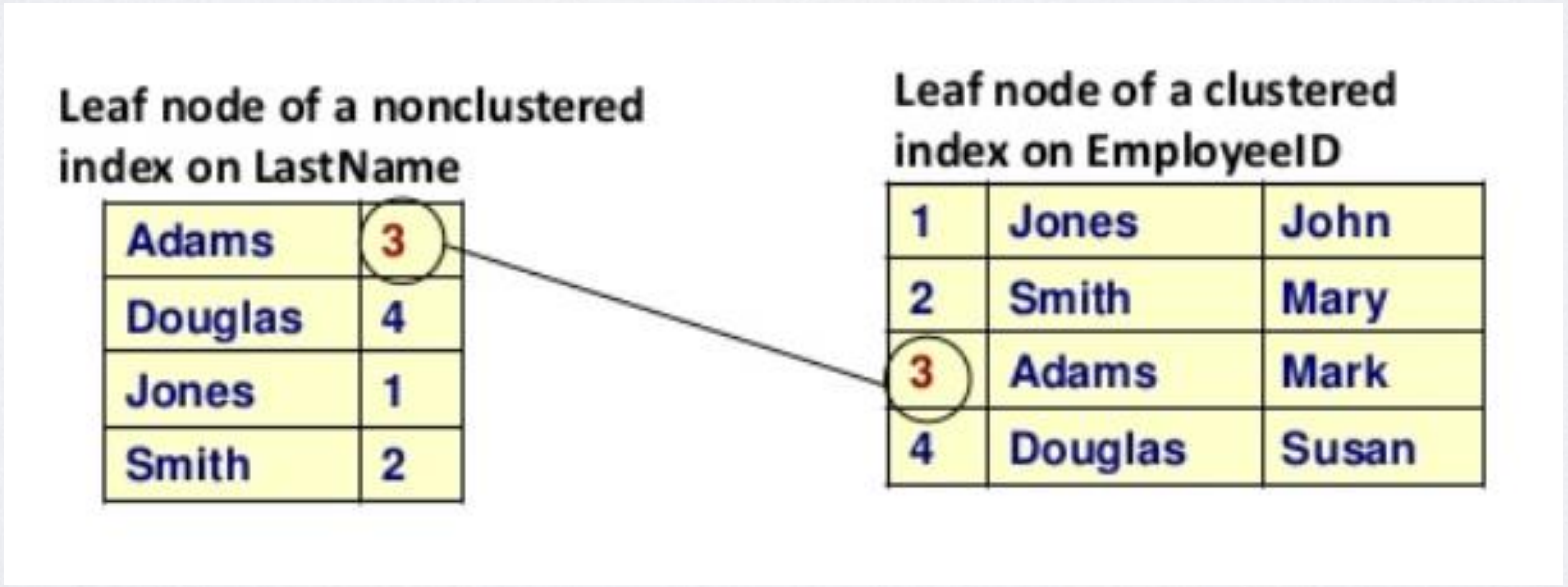
File1, Page 1007

CustId	Name	PhoneNo
13	Jaya	123267
14	Amit	123366

File1, Page 1007

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



Leaf node of a nonclustered index on LastName

Adams	3
Douglas	4
Jones	1
Smith	2

Leaf node of a clustered index on EmployeeID

1	Jones	John
2	Smith	Mary
3	Adams	Mark
4	Douglas	Susan

INDEX

- How does Index improve the performance?
  - Find Index vs. Table Scan
- Will Index always improve the performance?
  - Maintain the index

Any Questions?