CSE1500 - WEB AND DATABASE TECHNOLOGY

DB LECTURE 3

INTRODUCTION TO SQL

Alessandro Bozzon

cse1500-ewi@tudelft.nl

AT THE END OF THIS LECTURE, YOU SHOULD BE ABLE TO....

- **Describe** and **design** SQL programs for the retrieval of data from tables
- **Prototype** and **deploy** database applications using open-source database systems (e.g., PostgreSQL)

THE SQL LANGUAGE

REQUIREMENTS OF A DATABASE LANGUAGE

- **Functional requirements**
  - Create database and relation structures
  - Perform data management tasks
    - Insertion, Modification, Deletion
  - Perform simple and complex queries

- **Non-functional requirements**
  - Structure and syntax easy to learn
  - Portability across systems

THE SQL LANGUAGE

- The name is an acronym for **S**tructured **Q**uery **L**anguage
- Far richer than a query language: a **DML**, a **DDL/VDL**

- SQL is a **set-based** language
  - operators works on relations (tables)
  - results are always relations (tables)

- SQL is declarative
  - It describes **what** to do with data, not **how** to do it

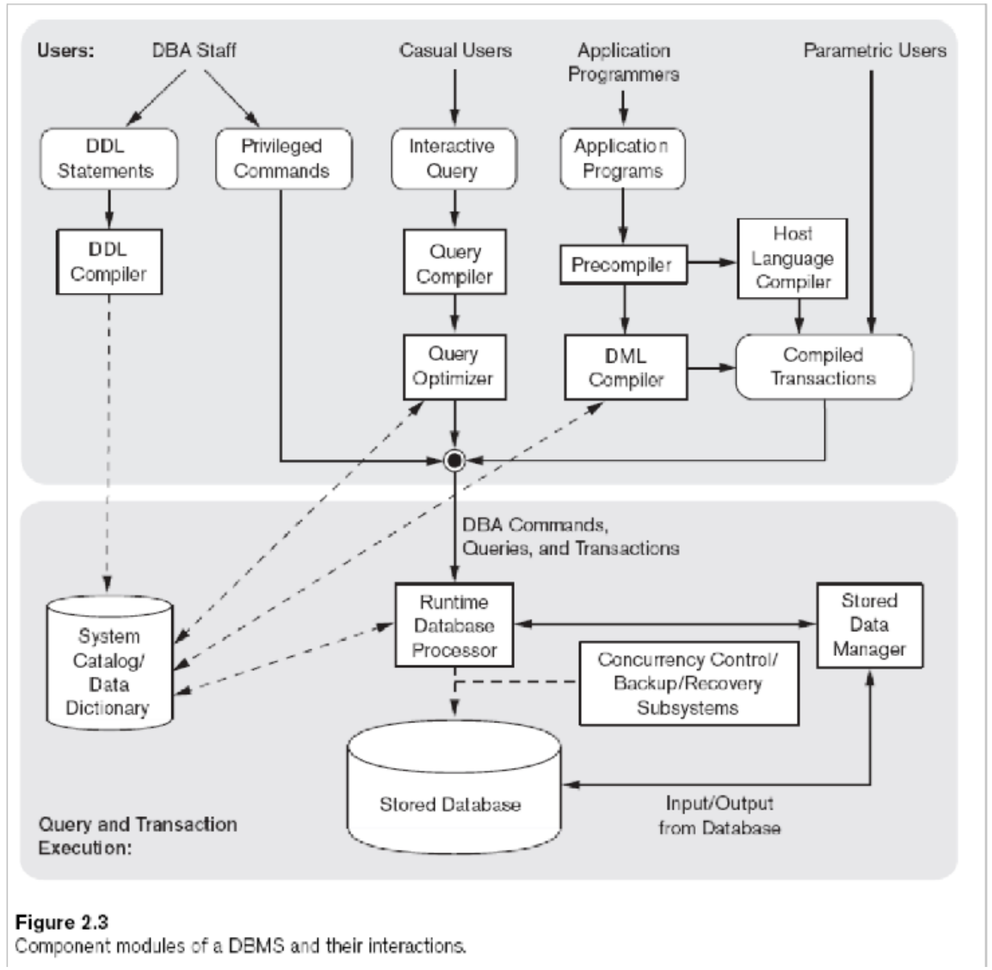


Figure 2.3
Component modules of a DBMS and their interactions.

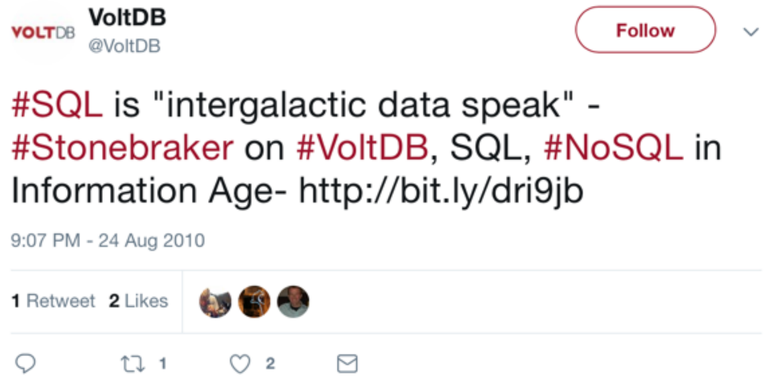
SQL IS INTERGALACTIC DATA SPEAK

- Successful, mainstream, and general purpose 4GL (fourth generation programming language) – perhaps the only one

> A brief discussion of this new class of users is in order here. There are some users whose interaction with a computer is so infrequent or unstructured that the user is unwilling to learn a query language. For these users, natural language or menu selection (3,4) seem to be the most viable alternatives. However, there is also a large class of users who, while they are not computer specialists, would be willing to learn to interact with a computer in a reasonably high-level, non-procedural query language. Examples of such users are accountants, engineers, architects, and urban planners. It is for this class of users that SEQUEL is intended. For this reason, SEQUEL emphasizes simple data structures and operations.

Chamberkin, Boyce. http://www.joakimdalby.dk/HTM/sequel.pdf

- Many standards out there:
  - ANSI SQL, SQL92 (a.k.a. SQL2), SQL99 (a.k.a. SQL3), ….
  - Vendors support various subsets (or supersets!)



Credits: Dave Roth

SQL VERSIONS

https://en.wikipedia.org/wiki/SQL

Name	Comments
SQL-86	First formalized by ANSI.
SQL-89	Minor revision that added integrity constraints, adopted as FIPS 127-1.
SQL-92	Major revision (ISO 9075), Entry Level SQL-92 adopted as FIPS 127-2.
SQL:1999	Added regular expression matching, recursive queries (e.g. transitive closure), triggers, procedural and control-of-flow statements, non-scalar types (arrays), and some object-oriented features (e.g. structured types). Embedding SQL in Java (SQL/OLB) and vice versa (SQL/JRT).
SQL:2003	Introduced XML-related features (SQL/XML), window functions, standardized sequences, and columns with auto-generated values (including identity-columns).
SQL:2006	ISO/IEC 9075-14:2006 defines ways that SQL can be used with XML. Lets integrate queries into their SQL code with XQuery
SQL:2008	Legalizes ORDER BY outside cursor definitions. Adds INSTEAD OF triggers, TRUNCATE statement, FETCH clause.
SQL:2011	Adds temporal data (PERIOD FOR). Enhancements for window functions and FETCH clause.
SQL:2016	Adds row pattern matching, polymorphic table functions, JSON.

Our Reference

Turing Complete!

Any program can be written in SQL!

- if you’re crazy enough :)

https://blog.jooq.org/2016/04/25/10-sql-tricks-that-you-didnt-think-were-possible/

DATA MANIPULATION LANGUAGE

- Instructions to retrieve information of interests
  - `SELECT`
- Instructions to modify instances of the database
  - `INSERT`: add new row(s) to a table
  - `UPDATE`: modify existing row(s) in a table
  - `DELETE`: delete existing row(s) from a table

DATA DEFINITION LANGUAGE

- Specification of the database schema: creation, modification, deletion of tables
  - `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`
- Specification of derived tables, or views
  - `CREATE VIEW`, `ALTER VIEW`, `DROP VIEW`
- Specification of indexes
  - `CREATE INDEX`, `DROP INDEX`
- Management of user access control
  - `GRANT`, `REVOKE`

EXAMPLE DATABASES

EXAMPLE DB1: EMPLOYEES

Employee					
FirstName	**Surname**	**Dept**	**Office**	**Salary**	**City**
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

Department		
DeptName	**Address**	**City**
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San Joné

EXAMPLE DB2: PRODUCTS

Supplier

CodeS	NameS	Shareholders	Office
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

Supply

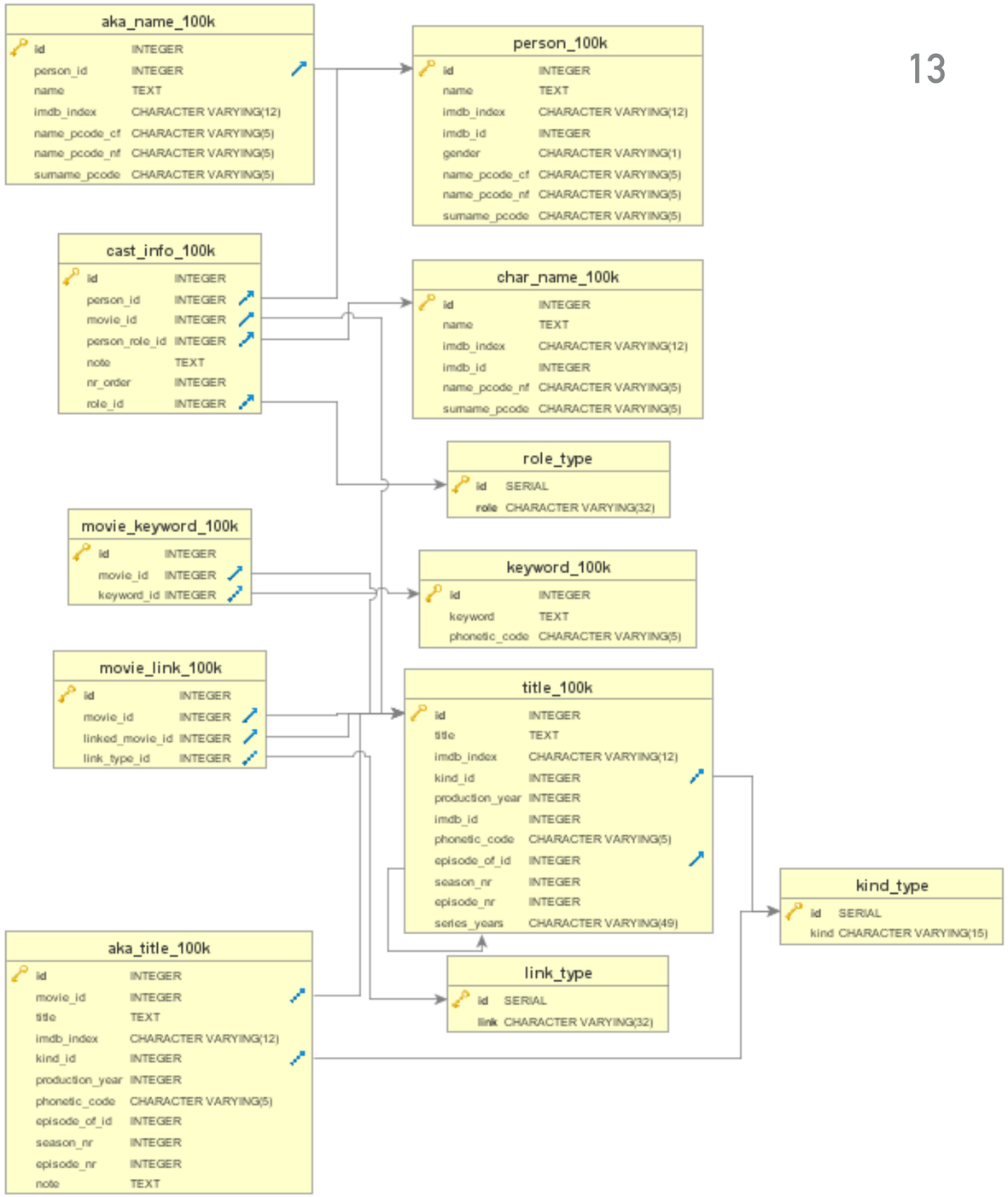
CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400

Products

CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	40	Den Haag
P6	Coat	Red	42	Amsterdam

EXAMPLE DB3: IMDB

- A subset of the schema and data from the IMDB.com website
  - Actors (person_100k), Movies (title_100k), and Actors in Movies (cast_info_100k)
  - Plus aliases, keywords, movie genres, etc.
- We will use MongoDB and Neo4J implementations of the same database (obviously, with different schemas)
- Get it (with import instructions) here
  - https://docs.google.com/document/d/1jj3cMAnk6Rc0mHkkOAIYDzYLjKisCuyj4-3KF9l-_8o



QUERYING

SQL AS A QUERY LANGUAGE

- SQL expresses queries in declarative way
  - queries specify the properties of the result, not the way to obtain it
- Queries are translated by the query optimiser into the procedural language internal to the DBMS
- The programmer should focus on readability, not on efficiency

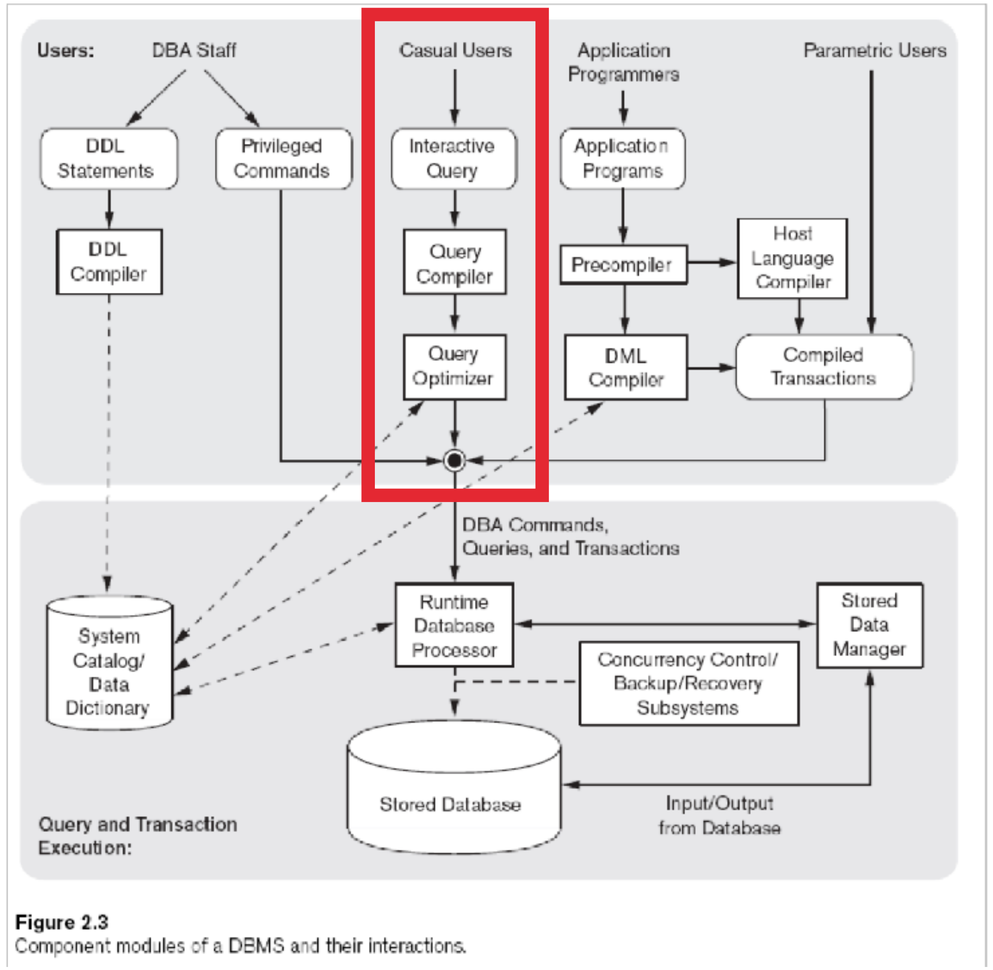


Figure 2.3
Component modules of a DBMS and their interactions.

SQL QUERIES

- Expressed by the SELECT statement

```
SELECT TargetList
FROM Table
[ WHERE Conditions ] [ ORDER BY OrderingAttributesList ]
[ GROUP BY GroupingAttributesList ] [ HAVING AggregateConditions ]
```

- The three parts of the query are usually called:
  - **Target list** or `SELECT` clause
  - `FROM` clause
  - `WHERE` clause
- The query:
  - considers the cartesian product of the tables in the `FROM` clause
  - considers only the rows that **satisfy** (evaluate to `TRUE`) the condition in the `WHERE` clause
  - for each row evaluates the attribute expressions in the `TargetList`, and returns them
  - More on `GROUP BY` and `HAVING` later

SELECT CLAUSE

SELECT QUERY FROM /1

- Find the *code* of **all** products in the DB

```
SELECT CodeP
FROM Products
```

Products				
CodeP	**NameP**	**Color**	**Size**	**Storehouse**
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	40	Den Haag
P6	Coat	Red	42	Amsterdam

CodeP
P1
P2
P3
P4
P5
P6

SELECT QUERY FROM /2

- Find the *code* and *number of shareholders* of suppliers ***located in*** "Den Haag"

```
SELECT CodeS, Shareholders
FROM Supplier
WHERE Office = "Den Haag"
```

Supplier

CodeS	NameS	Shareholders	Office
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

CodeS	Shareholders
S2	1
S3	3

Only the tuples evaluating the logical expression in the WHERE clause to TRUE are selected

* IN THE TARGET LIST

- Find all the information relating to employees **named** “Brown”

```
SELECT *
FROM Employee
WHERE Surname = “Brown”
```

Employee					
FirstName	**Surname**	**Dept**	**Office**	**Salary**	**City**
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

FirstName	**Surname**	**Dept**	**Office**	**Salary**	**City**
Mary	Brown	Administration	10	45	London
Charles	Brown	Planning	14	80	London

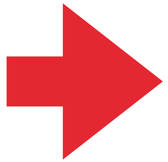
ATTRIBUTE EXPRESSIONS WITH AS/1

- The keyword AS allows the definition of an **alias**. Used in attribute expressions, it defines a new **temporary** column per the calculated expression
- Find the monthly salary of the employees **named** "White"

```
SELECT Salary / 12 AS MonthlySalary
FROM Employee
WHERE Surname = "White"
```

Employee

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse



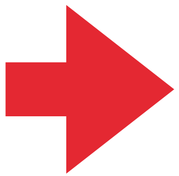
MonthlySalary
3.00

ATTRIBUTE EXPRESSIONS WITH AS/2

- Find the salaries of employees *named* "Brown", and alias it as "Remuneration"

```
SELECT Salary AS Remuneration
FROM Employee
WHERE Surname = "Brown"
```

Employee					
FirstName	**Surname**	**Dept**	**Office**	**Salary**	**City**
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse



Remuneration
45
80

DUPLICATES

- In relational algebra and calculus the results of queries **do not contain duplicates** (set semantics)
- In SQL, result sets **may have identical rows** (bag semantics)
- Duplicates **rows** can be removed using the keyword `DISTINCT`
  - This applies also with rows having multiple columns

```
SELECT City
FROM Department
```

Department

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San Joné

City
London
Toulouse
Brighton
London
San Joné

```
SELECT DISTINCT City
FROM Department
```

Department

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San Joné

City
London
Toulouse
Brighton
San Joné

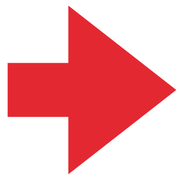
DISTINCT KEYWORD

- Find the code of the products **supplied <u>at least</u> by one supplier**

```
SELECT DISTINCT CodeP
FROM Supply
```

Supply

CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400



CodeP
P1
P2
P3
P4
P5
P6

ARE THE FOLLOWING TWO QUERIES RETURNING THE SAME NUMBER OF RESULTS?

```
SELECT DISTINCT (title, production_year)
FROM title_100k
```

```
SELECT id
FROM title_100k
```

A) **Yes**, because movies with the same name cannot be released in different years

B) **Yes**, because movies cannot have the same name

C) **No**, because movies with the same name can only be released in different years

D) **No**, because movies with the same name can be released in the same year

ARE THE FOLLOWING TWO QUERIES RETURNING THE SAME NUMBER OF RESULTS?

```
SELECT DISTINCT (title, production_year)
FROM title_100k
```

```
SELECT id
FROM title_100k
```

A) **Yes**, because movies with the same name cannot be released in different years

B) **Yes**, because movies cannot have the same name

C) **No**, because movies with the same name can only be released in different years

D) **No**, because movies with the same name can be released in the same year

Check for titles 4496363 (a movie) and 2853318 (an episode of a TV series)

WHERE CLAUSE

WHERE CLAUSE

- **Selection conditions** apply to **each single tuple** resulting from the evaluation of the `FROM` clause
- Defined as a **boolean expression of simple predicates**
- Simple predicates
  - comparison between attributes and/or constant values
  - set membership
  - textual matching
  - `NULL` values

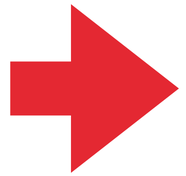
PREDICATE CONJUNCTION /1

- Find the *first names* and *surnames* of the employees **who work in office number** `20` **of the** `“Administration"` **department**

```
SELECT FirstName, Surname
FROM Employee
WHERE Office = “20” AND Dept = “Administration”
```

Employee

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse



FirstName	Surname
Gus	Green

PREDICATE CONJUNCTION /2

- Find the *first names* and *surnames* of the employees **who work in the** “Administration” **department and in the** “Production” **department**

```
SELECT FirstName, Surname
FROM Employee
WHERE Dept = “Administration” AND Dept = “Production”
```

- ???

Employee

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

- **No Results!**

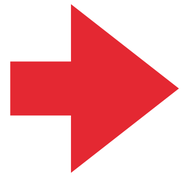
PREDICATE DISJUNCTION

- Find the *first names* and *surnames* of the employees **who work in <u>either</u> the "Administration" <u>or</u> the "Production" department**

```
SELECT FirstName, Surname
FROM Employee
WHERE Dept = "Administration" OR Dept = "Production"
```

Employee

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse



FirstName	Surname
Mary	Brown
Charles	White
Gus	Green
Pauline	Bradshaw
Alice	Jackson

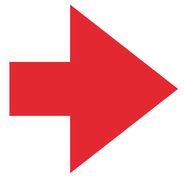
COMPLEX LOGICAL EXPRESSIONS

- Find the first names of the employees **named** “Brown” **who work in the** “Administration” **department or the** “Production” **department**

```
SELECT FirstName
FROM Employee
WHERE Surname = “Brown” AND (Dept = “Administration” OR Dept = “Production”)
```

Employee

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse



FirstName
Mary

PostgreSQL Logical Operators: https://www.postgresql.org/docs/current/functions-logical.html

OPERATOR LIKE /1

- Matching string patterns
- The character “_” is a matching term for any single character, which must be found in the specified position
- The character “%” is a matching term for any sequence of *zero* or more characters

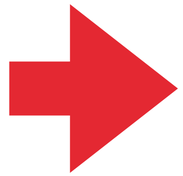
OPERATOR LIKE /2

- Find the *code* and the *name* of the products **having name starting with the letter** “S”

```
SELECT CodeP, NameP
FROM Products
WHERE NameP LIKE “S%”
```

Products

CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	40	Den Haag
P6	Coat	Red	42	Amsterdam



CodeP	NameP
P1	Sweater
P3	Shirt
P4	Shirt
P5	Skirt

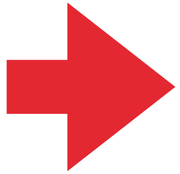
OPERATOR LIKE /3

- Find the employees with **surnames that have "r" as the second letter and that end in "n"**

```
SELECT *
FROM Employee
WHERE Surname LIKE "_r%n"
```

Employee

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse



FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Gus	Green	Administration	20	40	Oxford
Charles	Brown	Planning	14	80	London

OPERATOR LIKE /4

- Find Suppliers **having the Office address containing the string** "Den Haag"

```
WHERE Address LIKE "%Den Haag%"
```

- Find Suppliers where **the supplier code ends in** 2

```
WHERE CodeS LIKE "_2"
```

- Find Products that are in storehouses **having names that do not contain an** "e" **as second character**

```
WHERE Storehouse NOT LIKE "_e%"
```

PostgreSQL LIKE Operator Reference: https://www.postgresql.org/docs/current/functions-matching.html#FUNCTIONS-LIKE
Other PostgreSQL Pattern Matching Operators: https://www.postgresql.org/docs/current/functions-matching.html

WHICH OF THE FOLLOWING QUERIES RETURN THE SAME RESULT SET?

1

```
SELECT *
FROM char_name_100k
WHERE name = 'A'
```

2

```
SELECT *
FROM char_name_100k
WHERE name LIKE 'A'
```

3

```
SELECT *
FROM char_name_100k
WHERE name LIKE ‘A_’
```

4

```
SELECT *
FROM char_name_100k
WHERE name LIKE 'A%'
```

5

```
SELECT *
FROM char_name_100k
WHERE name LIKE 'A%_'
```

A) Only **1)** and **2)**

B) Only **3)** and **4)**

C) **1)** and **2)** , **4)** and **5)**

D) **All**

WHICH OF THE FOLLOWING QUERIES RETURN THE SAME RESULT SET?

1

```
SELECT *
FROM char_name_100k
WHERE name = 'A'
```

2

```
SELECT *
FROM char_name_100k
WHERE name LIKE 'A'
```

3

```
SELECT *
FROM char_name_100k
WHERE name LIKE ‘A_’
```

4

```
SELECT *
FROM char_name_100k
WHERE name LIKE 'A%'
```

5

```
SELECT *
FROM char_name_100k
WHERE name LIKE 'A%_'
```

A) Only **1)** and **2)**

B) Only **3)** and **4)**

C) **1)** and **2)** , **4)** and **5)**

D) **All**

DEALING WITH NULL VALUES

- NULL values may mean that:
  - a value is **unknown** (exists but it is not known)
  - a value is **not available** (exists but it is purposely withheld)
  - a value is **not applicable** (undefined for this tuple)
- Each individual NULL value is considered to be **different from every other** NULL value
- When a NULL is involved in a comparison operation, the results is considered to be UNKNOWN
- SQL uses a **three-valued logic**
  - TRUE, FALSE, and UNKNOWN
  - All logical operators evaluate to TRUE, FALSE, or UNKNOWN
  - In PostgreSQL, these are implemented as true, false, and NULL
  - Most of this is common to different SQL database servers, although some servers may return any nonzero

COMPARISONS INVOLVING NULL AND THREE-VALUED LOGIC

Table 5.1 Logical Connectives in Three-Valued Logic

(a)

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

(b)

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

(c)

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

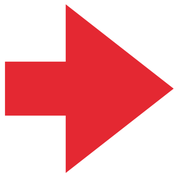
THE IS NULL OPERATOR

- AttributeName IS [NOT] NULL
- Find the *code* and the *name* of products **having no specified** Size

```
SELECT CodeP, NameP
FROM Products
WHERE Size IS NULL
```

Products

CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	NULL	Den Haag
P6	Coat	Red	42	Amsterdam



CodeP	NameP
P5	Skirt

THE IS NULL OPERATOR

- Find the *code* and the *name* of products **having size greater than** 44, **or that might have size greater than** 44

```
SELECT CodeP, NameP
FROM Products
WHERE Size > 44 OR Size IS NULL
```

Products

CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	NULL	Den Haag
P6	Coat	Red	42	Amsterdam

CodeP	NameP
P2	Jeans
P3	Shirt
P5	Skirt

ARE THE FOLLOWING TWO QUERIES RETURNING THE SAME SETS OF RESULTS?

```sql
SELECT *
FROM title_100k
WHERE season_nr IS NOT NULL
```

```sql
SELECT *
FROM title_100k
WHERE season_nr > 0
```

A) **Yes**, because no movie in the DB has `season_nr = 0`

B) **Yes**, because all movies in the DB have a `season_nr`

C) **No**, because some movies have `season_nr = NULL`

D) **No**, because the two queries test different condition

ARE THE FOLLOWING TWO QUERIES RETURNING THE SAME SETS OF RESULTS?

```sql
SELECT *
FROM title_100k
WHERE season_nr IS NOT NULL
```

```sql
SELECT *
FROM title_100k
WHERE season_nr > 0
```

A) **Yes**, because no movie in the DB has `season_nr = 0`

B) **Yes**, because all movies in the DB have a `season_nr`

C) **No**, because some movies have `season_nr = NULL`

D) **No**, because the two queries test different condition

WHICH OF THE FOLLOWING SQL QUERIES ARE GUARANTEED TO ALWAYS PRODUCE A RESULT-SET CONTAINING ALL THE TUPLES FROM THE title TABLE OF THE IMDB DATABASE?

1

```
SELECT DISTINCT (title,production_year)
FROM title_100k
WHERE id IS NOT NULL
```

2

```
SELECT DISTINCT (id,title)
FROM title_100k
WHERE id > 0
```

3

```
SELECT DISTINCT (title,production_year, season_nr)
FROM title_100k
```

4

```
SELECT *
FROM title_100k
WHERE title IS NOT NULL
```

ORDERING OF RESULTS

ORDERING

- The ORDER BY clause, at the end of the query, orders the rows of the results
- Last operator applied by the database in the query execution plan
- Syntax:

```
ORDER BY OrderingAttribute [ asc | desc  ]
           {, OrderingAttribute [ asc | desc  ]}
```

- The implicit ordering is ASC: ascending

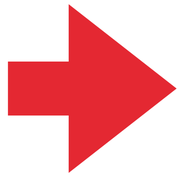
ORDER BY /1

- Find the *code*, *name* and the *size* of all the products, **in descending order of size**

```
SELECT CodeP, NameP, Size
FROM Products
ORDER BY Size DESC
```

Products

CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	40	Den Haag
P6	Coat	Red	42	Amsterdam



CodeP	NameP	Size
P2	Jeans	48
P3	Shirt	48
P4	Shirt	44
P6	Coat	42
P1	Sweater	40
P5	Skirt	40

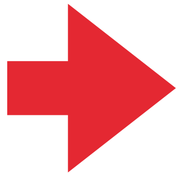
ORDER BY /2

- Find all the information about products, **in ascending order of** *name* **and descending order of** *size*

```
SELECT *
FROM Products
ORDER BY NameP, Size DESC
```

Products

CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	40	Den Haag
P6	Coat	Red	42	Amsterdam



CodeP	NameP	Color	Size	Storehouse
P6	Coat	Red	42	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	40	Den Haag
P1	Sweater	Red	40	Amsterdam

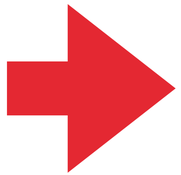
ORDER BY /3

- Find the code and the *american size* of all the products, **in ascending order of size**

```
SELECT CodeP, Size - 14 AS AmericanSize
FROM Products
ORDER BY AmericanSize
```

Products

CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	40	Den Haag
P6	Coat	Red	42	Amsterdam



CodeP	AmericanSize
P1	26
P5	26
P6	28
P4	30
P2	34
P3	34

JOIN

QUERYING MULTIPLE TABLES

- All possible tuple combinations
- What if we want to retrieve:
  - the name of all the **suppliers of product** "P2"

Supplier

CodeS	NameS	Shareholders	Office
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

Supply

CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400

Products

CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	40	Den Haag
P6	Coat	Red	42	Amsterdam

CROSS PRODUCT /1

- All possible tuple combinations
- Find the *name* of **all** the suppliers **of** product "P2"

```
SELECT NameS
FROM Supplier, Supply
```

Supplier				Supply		
CodeS	NameS	Shareholders	Office	CodeS	CodeP	Amount
S1	John	2	Amsterdam	S1	P1	300
S1	John	2	Amsterdam	S1	P2	200
S1	John	2	Amsterdam	S1	P3	400
S1	John	2	Amsterdam	S1	P4	200
S1	John	2	Amsterdam	S1	P5	100
S1	John	2	Amsterdam	S1	P6	100
S1	John	2	Amsterdam	S2	P1	300
...	...	...	...	...	...	...
S2	Victor	1	Den Haag	S1	P1	300
...	...	...	...	...	...	...
S2	Victor	1	Den Haag	S2	P1	300
...	...	...	...	...	...	...
S3	Anna	3	Den Haag	S1	P1	300
...	...	...	...	...	...	...
S3	Anna	3	Den Haag	S3	P2	200
...	...	...	...	...	...	...

CROSS PRODUCT /2

- Find the *name* of **all** the suppliers **of** product "P2"

Supplier				Supply		
CodeS	NameS	Shareholders	Office	CodeS	CodeP	Amount
S1	John	2	Amsterdam	S1	P1	300
S1	John	2	Amsterdam	S1	P2	200
S1	John	2	Amsterdam	S1	P3	400
S1	John	2	Amsterdam	S1	P4	200
S1	John	2	Amsterdam	S1	P5	100
S1	John	2	Amsterdam	S1	P6	100
S1	John	2	Amsterdam	S2	P1	300
...	...	...	...	...	...	...
S2	Victor	1	Den Haag	S1	P1	300
...	...	...	...	...	...	...
S2	Victor	1	Den Haag	S2	P1	300
...	...	...	...	...	...	...
S3	Anna	3	Den Haag	S1	P1	300
...	...	...	...	...	...	...
S3	Anna	3	Den Haag	S3	P2	200
...	...	...	...	...	...	...

What is the problem with this result set?

SIMPLE JOIN /1

```
SELECT NameS
FROM Supplier, Supply
WHERE Supplier.CodeS = Supply.CodeS
```

=

Supplier				Supply		
CodeS	NameS	Shareholders	Office	CodeS	CodeP	Amount
S1	John	2	Amsterdam	S1	P1	300
S1	John	2	Amsterdam	S1	P2	200
S1	John	2	Amsterdam	S1	P3	400
S1	John	2	Amsterdam	S1	P4	200
S1	John	2	Amsterdam	S1	P5	100
S1	John	2	Amsterdam	S1	P6	100
S1	John	2	Amsterdam	S2	P1	300
...	...	...	...	...	...	...
S2	Victor	1	Den Haag	S1	P1	300
...	...	...	...	...	...	...
S2	Victor	1	Den Haag	S2	P1	300
...	...	...	...	...	...	...
S3	Anna	3	Den Haag	S1	P1	300
...	...	...	...	...	...	...
S3	Anna	3	Den Haag	S3	P2	200
...	...	...	...	...	...	...

SIMPLE JOIN /2

- `Supplier.CodeS = Supply.CodeS` is a **JOIN CONDITION**

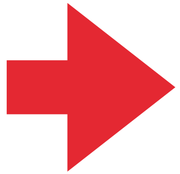
Supplier				Supply		
CodeS	NameS	Shareholders	Office	CodeS	CodeP	Amount
S1	John	2	Amsterdam	S1	P1	300
S1	John	2	Amsterdam	S1	P2	200
S1	John	2	Amsterdam	S1	P3	400
S1	John	2	Amsterdam	S1	P4	200
S1	John	2	Amsterdam	S1	P5	100
S1	John	2	Amsterdam	S1	P6	100
S2	Victor	1	Den Haag	S2	P1	300
S2	Victor	1	Den Haag	S2	P2	400
S3	Anna	3	Den Haag	S3	P2	200
S4	Angela	2	Amsterdam	S4	P3	200
S4	Angela	2	Amsterdam	S4	P4	300
S4	Angela	2	Amsterdam	S4	P5	400

OUR ORIGINAL QUERY

- Find the *name* of **all** the suppliers **of** product "P2"

```
SELECT NameS
FROM Supplier, Supply
WHERE Supplier.CodeS = Supply.CodeS AND CodeP = "P2"
```

Supplier				Supply		
CodeS	NameS	Shareholders	Office	CodeS	CodeP	Amount
S1	John	2	Amsterdam	S1	P1	300
S1	John	2	Amsterdam	S1	P2	200
S1	John	2	Amsterdam	S1	P3	400
S1	John	2	Amsterdam	S1	P4	200
S1	John	2	Amsterdam	S1	P5	100
S1	John	2	Amsterdam	S1	P6	100
S2	Victor	1	Den Haag	S2	P1	300
S2	Victor	1	Den Haag	S2	P2	400
S3	Anna	3	Den Haag	S3	P2	200
S4	Angela	2	Amsterdam	S4	P3	200
S4	Angela	2	Amsterdam	S4	P4	300
S4	Angela	2	Amsterdam	S4	P5	400



NameS
John
Victor
Anna

ANOTHER QUERY

- Find the *name* of supplier of **at least one red product**

```
SELECT DISTINCT NameS
FROM Supplier, Supply, Products
WHERE Supplier.CodeS = Supply.CodeS AND Supply.CodeP = Product.CodePCodeP
      AND Color = “Red"
```

If there are *N* tables in the `FROM` clause, at least *N – 1* `JOIN` conditions in the `WHERE` clause

Supplier

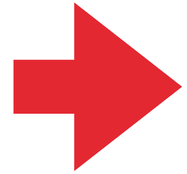
CodeS	NameS	Shareholders	Office
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

Supply

CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400

Products

CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	40	Den Haag
P6	Coat	Red	42	Amsterdam



NameS
John
Victor

USING AS KEYWORD FOR TABLES

- Find the *code* **pairs** of suppliers **having their office in the same city**
- All possible tuple combinations

```
SELECT S1.CodeS, S2.CodeS
FROM Supplier AS S1, Supplier AS S2
WHERE S1.Office = S2.Office
```

Supplier AS S1			
CodeS	**NameS**	**Shareholders**	**Office**
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

Supplier AS S2			
CodeS	**NameS**	**Shareholders**	**Office**
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

RESULT /1

- Find the *code* **pairs** of suppliers **having their office in the same city**

Supplier AS S1			
CodeS	**NameS**	**Shareholders**	**Office**
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

Supplier AS S2			
CodeS	**NameS**	**Shareholders**	**Office**
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

- **Pairs** of identical values
- **Permutations** of the same pair of values

S1.CodeS	S2.CodeS
S1	S1
S1	S4
S2	S2
S2	S3
S3	S2
S3	S3
S4	S1
S4	S4
S5	S5

RESULT /2

- Find the *code* **pairs** of suppliers **having their office in the same city**

```
SELECT S1.CodeS, S2.CodeS
FROM Supplier AS S1, Supplier AS S2
WHERE S1.Office = S2.Office AND S1.CodeS <> S2.CodeS
```

- Remove pairs with the same code value

S1.CodeS	S2.CodeS
~~S1~~	~~S1~~
S1	S4
~~S2~~	~~S2~~
S2	S3
S3	S2
~~S3~~	~~S3~~
S4	S1
~~S4~~	~~S4~~
~~S5~~	~~S5~~

RESULT/3

- Find the *code* **pairs** of suppliers **having their office in the same city**

```
SELECT S1.CodeS, S2.CodeS
FROM Supplier AS S1, Supplier AS S2
WHERE S1.Office = S2.Office AND S1.CodeS < S2.CodeS
```

- Let's keep only the right ones

S1.CodeS	S2.CodeS
~~S1~~	~~S1~~
S1	S4
~~S2~~	~~S2~~
S2	S3
~~S3~~	~~S2~~
~~S3~~	~~S3~~
~~S4~~	~~S1~~
~~S4~~	~~S4~~
~~S5~~	~~S5~~

JOINS IN SQL92

http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt

- SQL-2 introduced an alternative syntax for the representation of JOINs, representing them explicitly in the from clause:

```
SELECT TargetList
FROM Table [[AS] Alias]
   { [JoinType] JOIN Table [[AS] Alias] [ON BooleanExpression || USING JoinColumns]}
[ WHERE Conditions ]
```

- JoinType can be any of `INNER`, `RIGHT [OUTER]`, `LEFT [OUTER]` or `FULL [OUTER]`, permitting the representation of outer joins
- The keyword `NATURAL` may precede `JoinType`

JOINS IN SQL92

- `NATURAL JOIN` on two relations `R` and `S`
  - No join condition specified
  - Implicit `EQUI JOIN` condition for each pair of attribute with same name from `R` and `S`
- `INNER JOIN`
  - **Default** type of join in a joined table (equivalent to `JOIN`)
  - Must specify `JOIN` attributes
  - Tuple is included in the results only if a matching tuple exists in the other relation
- `LEFT OUTER JOIN`
  - Every tuple in **left** table must appear in result
  - If no matching tuple: values for attributes in the right table set to `NULL`
- `RIGHT OUTER JOIN`
  - Every tuple in **right** table must appear in result
  - If no matching tuple: values for attributes in the left table set to `NULL`
- `FULL OUTER JOIN`
  - If no matching tuple: values for attributes in the left and/or right tables set to `NULL`

INNER JOIN

- Find the name of supplier of **at least one red product**

```
SELECT DISTINCT NameS
FROM Products JOIN Supply USING (CodeP)
              JOIN Supplier USING (CodeS)
WHERE Color = “Red"
```

Supplier

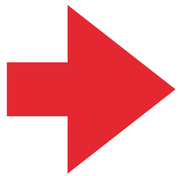
CodeS	NameS	Shareholders	Office
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

Supply

CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400

Products

CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	40	Den Haag
P6	Coat	Red	42	Amsterdam



NameS
John
Victor

- Same results as in Slide 58

LEFT OUTER JOIN

- Find the *code* and *name* of Supplier, and the *code* of the supplied Products, showing also suppliers of no products

```
SELECT Supply.CodeS, Supplier.NameS, Supply.CodeP
FROM Supplier LEFT OUTER JOIN Supply
                ON Supplier.CodeS = Supply.CodeS
```

CodeS	NameS	CodeP
S1	John	P1
S1	John	P2
S1	John	P3
S1	John	P4
S1	John	P5
S1	John	P6
S2	Victor	P1
S2	Victor	P2
S3	Anna	P2
S4	Angela	P3
S4	Angela	P4
S4	Angela	P5
S5	Paul	NULL

WRAPPING UP

TODAY WE COVERED

- SQL as a Retrieval Language
- Next Lectures
  - Advanced Retrieval Operations
  - SQL as a Schema Creation and Modification Language
  - SQL as a Data Manipulation Language

END OF LECTURE