SQL Part 1

CS460 Database Technology

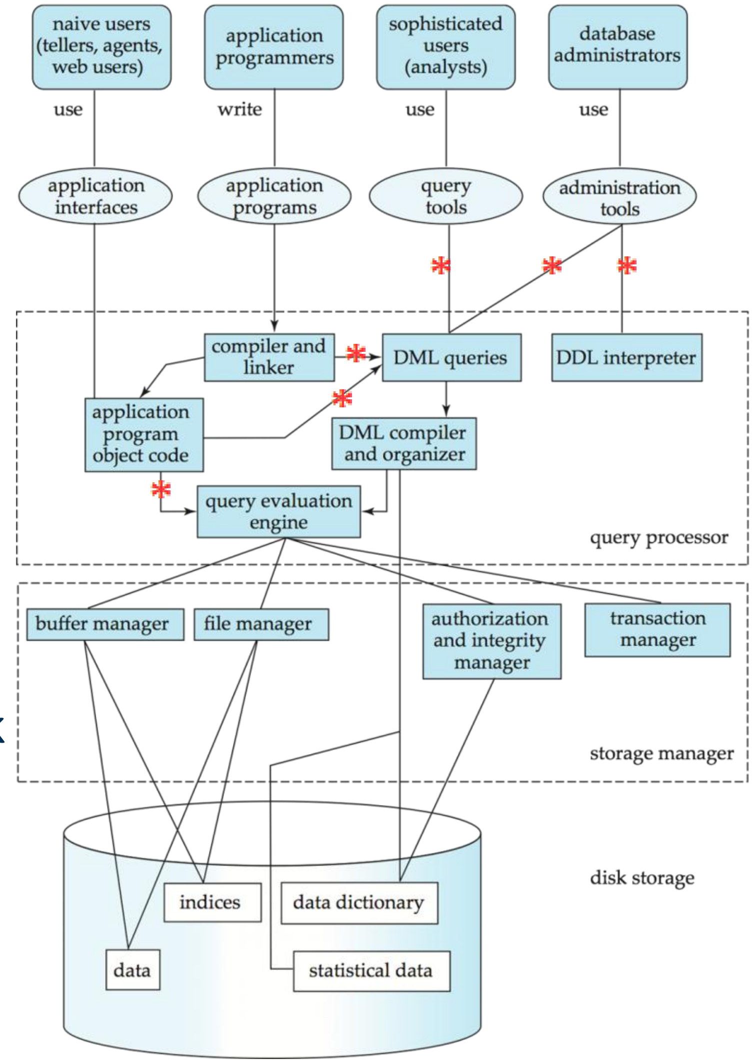
Outline

- **Today**
  - Overview of The SQL Query Language
  - Basic Query Structure
  - Set Operations
  - Join Operators
  - Null Values
  - Aggregate Functions
  - Nested Subqueries

- **Next week**
  - Data Definition
  - Data Types in SQL
  - Modifications of the database
  - Views
  - Integrity Constraints
  - Roles & Rights

Recap: Database Systems

- Users and applications interact with databases
  - By issuing *queries*
  - Data definition (DDL):
    defining, altering, deleting tables
  - Data manipulation (DML):
    reading from & writing to tables
- SQL is both a DDL and a DML
  - The language that most DBMS speak



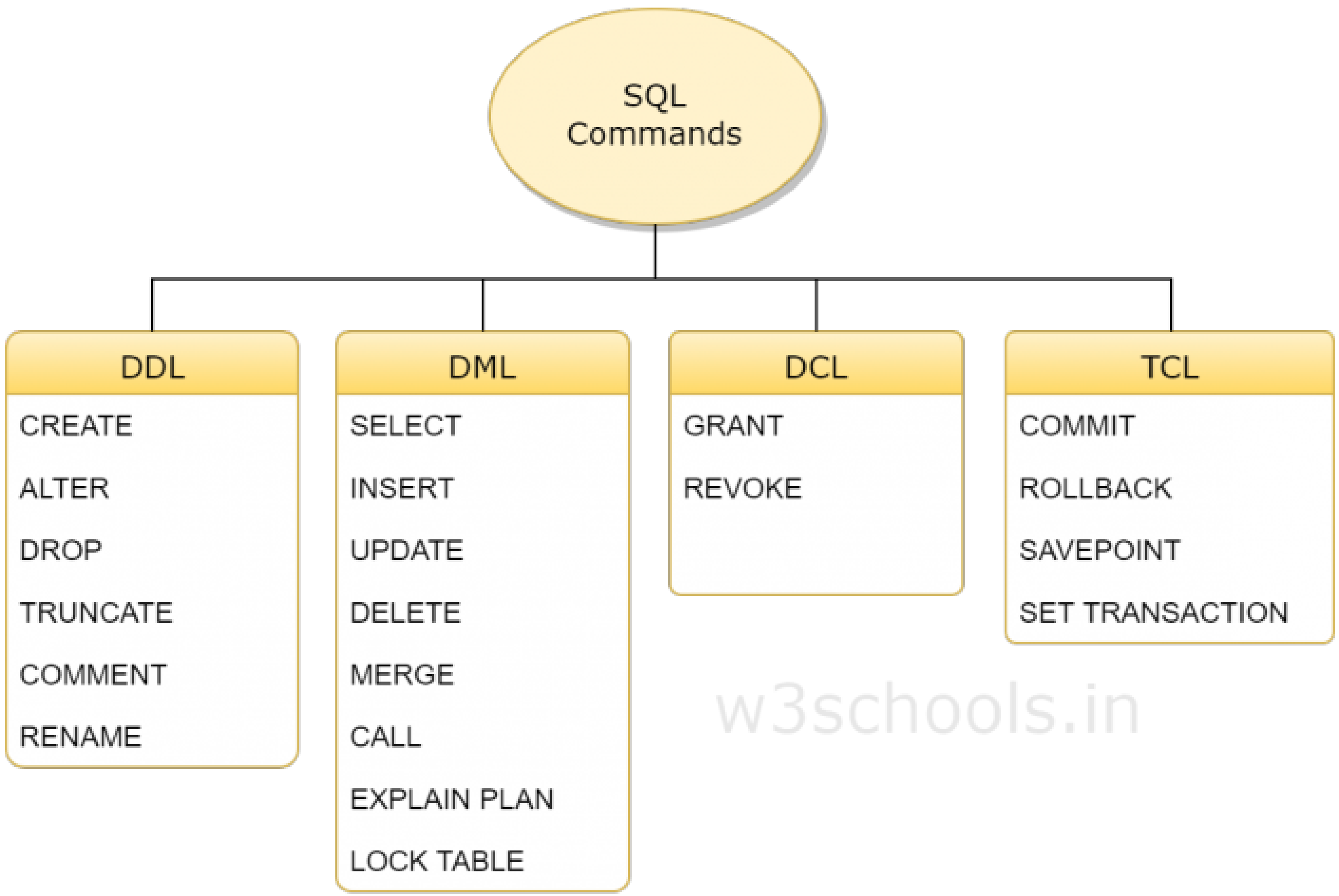
History

- IBM SEQUEL language developed as part of System R project at the IBM San Jose Research Laboratory
  - *Structured English QUEry Language*
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999
  - SQL:2003

Naming became Y2K compliant! ;-)

- Commercial + free systems offer most, if not all, SQL-92 features
  - plus varying feature sets from later standards and special proprietary features
  - Not all examples here may work on your particular system!

Parts of SQL: The Big Picture



Source: https://www.w3schools.in/mysql/ddl-dml-dcl/

Reading Data

- The **select** clause lists the attributes desired in the result of a query
- Example: find the names of all instructors:

  SELECT *name*
  FROM *instructor*

- In relational algebra:
  - $\prod_{\text{name}} (instructor)$

A Note on Case Sensitivity

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- SQL is completely case insensitive
  - **select = SELECT = SeLeCt**
- Also for names of relations and attributes
  - instructor = Instructor = INSTRUCTOR
  - name = NAME = nAmE
- Each relation / attribute can only exist once
  - Hence, two relations named *instructor* and *Instructor* would not be feasible
- Case sensitivity does *not* apply to values!
  - i.e., “Einstein” and “einstein” are different values!

Renaming Columns in a Select

- Columns can be renamed during selection

  SELECT *name, salary* **as** *payment*

  FROM *instructor*

- In relational algebra
  - a composition of projection and renaming:
  - $\rho_{payment \leftarrow salary} (\Pi_{name,salary} (instructor))$

The Select Clause

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- An asterisk in the select clause denotes "all attributes"

  SELECT *

  FROM *instructor*

- An attribute can be a literal with no **FROM** clause, possibly renamed

  SELECT '437'

  SELECT '437' **AS** *FOO*

FOO
437

- An attribute can be a literal with **FROM** clause

  SELECT *name*, 'Instructor' **AS** *role* **FROM** *instructor*

  UNION

  SELECT *name*, 'Student' **AS** *role* **FROM** *student*

name	**role**
Smith	Instructor
Einstein	Instructor
…	…
Johnson	Student
…	…

Duplicates

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Difference to relational algebra
  - Sets do not contain duplicates!
- SQL allows duplicates in relations as well as in query results

unless we define a constraint (see later)

- To force the elimination of duplicates, insert the keyword **DISTINCT** after SELECT.
- Find the department names of all instructors, and remove duplicates

  SELECT DISTINCT *dept_name*

  FROM *instructor*

Arithmetics in the Selection

- The **SELECT** clause can contain arithmetic expressions involving the operation, +, –, ∗, and /, and operating on constants or attributes of tuples
  - Here, we leave relational algebra!

- The query

    SELECT *ID, name, salary/12*

    FROM *instructor*

  would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12

- Combined with renaming:
  - **SELECT** *ID, name, salary/12* **AS** *monthly_salary*
    FROM *instructor*

Reading Parts of a Relation

- So far, we have always read an entire relation
- Usually, we are interested only in a small portion
- The **WHERE** clause restricts which parts of the table to read
- To find all instructors in Comp. Sci. dept

  SELECT *name*
  FROM *instructor*
  WHERE *dept_name* = 'Comp. Sci.'

- In relational algebra: combination of selection and projection

$$\pi_{\text{name}}(\sigma_{\text{dept\_name = 'Comp. Sci.'}}(\textit{instructor}))$$

Reading Parts of a Relation

- Comparison results can be combined using the logical connectives **AND, OR,** and **NOT**

  SELECT *name*

  FROM *instructor*

  WHERE *dept_name* = 'Comp. Sci.' **AND** *salary* > 90000

$$\pi_{name}(\sigma_{dept\_name = \text{'Comp. Sci.'} \wedge salary>90000}(instructor))$$

- Can be combined with results of arithmetic expressions

  SELECT *name, salary/12* **AS** *monthly_salary*
  FROM *instructor*
  WHERE *dept_name* = 'Comp. Sci.' **AND** *monthly_salary* > 7500

Searching in Texts

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- So far, we have handled exact equality in selections
- Sometimes, we want to search differently
  - All books that contain “database”
  - All authors starting with “S”
  - …
- In SQL: comparing with **LIKE** and two special characters:
  - _ = any arbitrary character
  - % = any number of arbitrary characters
  - masking with backslash
  - **SELECT … WHERE** *title* **LIKE** ‘%database%’
  - **SELECT** … **WHERE** *author* **LIKE** ‘S%’
  - **SELECT** … **WHERE** *amount* **LIKE** *‘100\%’*

most SQL engines don’t check types

Reading Data from Multiple Tables

- Example: find all instructors and the courses they teach
- **SELECT * FROM** *instructor, teaches*
  - this generates the *cartesian product*, i.e., instructor x teaches
  - result: generates every possible instructor – teaches pair, with all attributes from both relations

but is that useful?

- Common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name
  - e.g., *instructor.ID, teaches.ID*
- Relational algebra notation:
  - $\rho_{instructor.ID \leftarrow ID}(instructor) \times \rho_{teaches.ID \leftarrow ID}(teaches)$

Cartesian Product

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151			
22222			
32343			

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
				2009
				2010
				2010
				2009

Inst.ID	name	dept_name	salary	teaches.ID	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2009
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010
12121	Wu	Pinance	90000	10101	CS-347	1	Fall	2009
12121	Wu	Pinance	90000	12121	FIN-201	1	Spring	2010
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2010
12121	Wu	Pinance	90000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Cartesian Products with Selection

- Find the names of all instructors who have taught some course and the course_id

  SELECT *name, course_id*

  FROM *instructor, teaches*

  WHERE *instructor.ID = teaches.ID*

- Relational algebra:

$$\pi_{name,course\_id}(\sigma_{instructor.ID=teaches.ID}(\rho_{instructor.ID \leftarrow ID}((\text{instructor}) \text{ x } \rho_{teaches.ID \leftarrow ID}(\text{teaches}))))$$

Cartesian Product

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151			
22222			
32343			

teaches

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
				2009
				2010
				2010
				2009

Inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2009
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~	~~12121~~	~~FIN-201~~	~~1~~	~~Spring~~	~~2010~~
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~	~~15151~~	~~MU-199~~	~~1~~	~~Spring~~	~~2010~~
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~	~~22222~~	~~PHY-101~~	~~1~~	~~Fall~~	~~2009~~
…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…
~~12121~~	~~Wu~~	~~Finance~~	~~90000~~	~~10101~~	~~CS-101~~	~~1~~	~~Fall~~	~~2009~~
~~12121~~	~~Wu~~	~~Finance~~	~~90000~~	~~10101~~	~~CS-315~~	~~1~~	~~Spring~~	~~2010~~
~~12121~~	~~Wu~~	~~Pinance~~	~~90000~~	~~10101~~	~~CS-347~~	~~1~~	~~Fall~~	~~2009~~
12121	Wu	Pinance	90000	12121	FIN-201	1	Spring	2010
~~12121~~	~~Wu~~	~~Finance~~	~~90000~~	~~15151~~	~~MU-199~~	~~1~~	~~Spring~~	~~2010~~
~~12121~~	~~Wu~~	~~Pinance~~	~~90000~~	~~22222~~	~~PHY-101~~	~~1~~	~~Fall~~	~~2009~~
…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…

Cartesian Products with Selection

- Find the names of all instructors in the Finance department who have taught some course, together with the course_id

  SELECT *name, course_id*

  FROM *instructor , teaches*

  WHERE *instructor.ID = teaches.ID* **AND** *instructor. dept_name* = 'Finance'

$\pi_{name,course\_id}(\sigma_{instructor.ID=teaches.ID \land dept\_name='Finance'}(\rho_{instructor.ID \leftarrow ID}(instructor) \times \rho_{teaches.ID \leftarrow ID}(\text{teaches})))$

Cartesian Product

University of Mannheim
Data and Web Science Group

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151			
22222			
32343			

teaches

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
				2009
				2010
				2010
				2009

Inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~	~~10101~~	~~CS-101~~	~~1~~	~~Fall~~	~~2009~~
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~	~~10101~~	~~CS-315~~	~~1~~	~~Spring~~	~~2010~~
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~	~~10101~~	~~CS-347~~	~~1~~	~~Fall~~	~~2009~~
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~	~~12121~~	~~FIN-201~~	~~1~~	~~Spring~~	~~2010~~
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~	~~15151~~	~~MU-199~~	~~1~~	~~Spring~~	~~2010~~
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~	~~22222~~	~~PHY-101~~	~~1~~	~~Fall~~	~~2009~~
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
~~12121~~	~~Wu~~	~~Finance~~	~~90000~~	~~10101~~	~~CS-101~~	~~1~~	~~Fall~~	~~2009~~
~~12121~~	~~Wu~~	~~Finance~~	~~90000~~	~~10101~~	~~CS-315~~	~~1~~	~~Spring~~	~~2010~~
~~12121~~	~~Wu~~	~~Pinance~~	~~90000~~	~~10101~~	~~CS-347~~	~~1~~	~~Fall~~	~~2009~~
12121	Wu	Pinance	90000	12121	FIN-201	1	Spring	2010
~~12121~~	~~Wu~~	~~Finance~~	~~90000~~	~~15151~~	~~MU-199~~	~~1~~	~~Spring~~	~~2010~~
~~12121~~	~~Wu~~	~~Pinance~~	~~90000~~	~~22222~~	~~PHY-101~~	~~1~~	~~Fall~~	~~2009~~
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Cartesian Product of a Table with Itself

- Find the names of all instructors who have a higher salary than **some** instructor in 'Comp. Sci'.
  - We need the same table twice
  - So, we have to use it under different names
  - **SELECT DISTINCT** *T.name*
    FROM *instructor* **AS** *T, instructor* **AS** *S*
    WHERE *T.salary > S.salary* **AND** *S.dept_name = 'Comp. Sci.'*

$$\pi_{T,name}(\sigma_{T.salary>S.salary \wedge S.dept\_name='Comp. Sci.'}(\rho_T(\text{instructor}) \text{ x } \rho_S(\text{instructor})))$$

- What happens if we omit the **distinct** here?

Join Operations

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- **Join operations**
  - take two relations
  - return as new relation as their result
- A join operation
  - is a Cartesian product
  - requires that tuples in the two relations match (under some condition)
  - specifies the attributes that are present in the result of the join

- The join operations are typically used as subquery expressions in the **FROM** clause

Join Operations

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Recap: We have already seen a form of joins:
- A join operation
  - is a Cartesian product
  - requires that tuples in the two relations match (under some condition)
  - specifies the attributes that are present in the result of the join

- Find the names of all instructors who have taught some course and the course_id

  SELECT *name, course_id*
  FROM *instructor, teaches*
  WHERE *instructor.ID = teaches.ID*

Outer Joins

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Consider the two relations below
- Desired:
  - List all courses with their prerequisites
  - Note: course CS-315 has no prerequisites

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Outer Joins

- List all courses with their prerequisites

  SELECT *C.course_id, C.title, C.credits, C.dept_name, P.course_id*

  FROM *course* **AS** *C, prereq* **AS** *P*

  WHERE *C.course_id = P.course_id*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

C.course_id	**C.title**	**C.credits**	**C.dept_name**	**P.course_id**
BIO-301	Genetics	4	Biology	BIO-101
CS-190	Game Design	4	Comp. Sci.	CS-101

Outer Joins

- List all courses with their prerequisites

SELECT *C.course_id, C.title, C.credits, C.dept_name, P.prereq_id*

FROM *course* **AS** *C* **LEFT OUTER JOIN** *prereq* **AS** *P* **ON** *C.course_id = P.course_id*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

C.course_id	C.title	C.credits	C.dept_name	P.course_id
BIO-301	Genetics	4	Biology	BIO-101
CS-190	Game Design	4	Comp. Sci.	CS-101
CS-315	Robotics	3	Comp. Sci.	**NULL**

Join Operations

- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated
  - **INNER JOIN**: ignore
  - **OUTER JOIN**: fill with **NULL** values

for the moment:
keyword for "a blank cell"

- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join
  - explicit: **ON** clause
  - implicit: **NATURAL** keyword

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_1, \ldots, A_n)$

Outer Joins

- List all courses with their prerequisites

  SELECT *C.course_id, C.title, C.credits, C.dept_name, P.prereq_id*

  FROM *course* **AS** *C* **RIGHT OUTER JOIN** *prereq* **AS** *P* **on** *C.course_id = P.course_id*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

C.course_id	**C.title**	**C.credits**	**C.dept_name**	**P.course_id**
BIO-301	Genetics	4	Biology	BIO-101
CS-190	Game Design	4	Comp. Sci.	CS-101
CS-347	**NULL**	**NULL**	**NULL**	CS-101

Outer Joins

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- List all courses with their prerequisites

  SELECT *C.course_id, C.title, C.credits, C.dept_name, P.prereq_id*

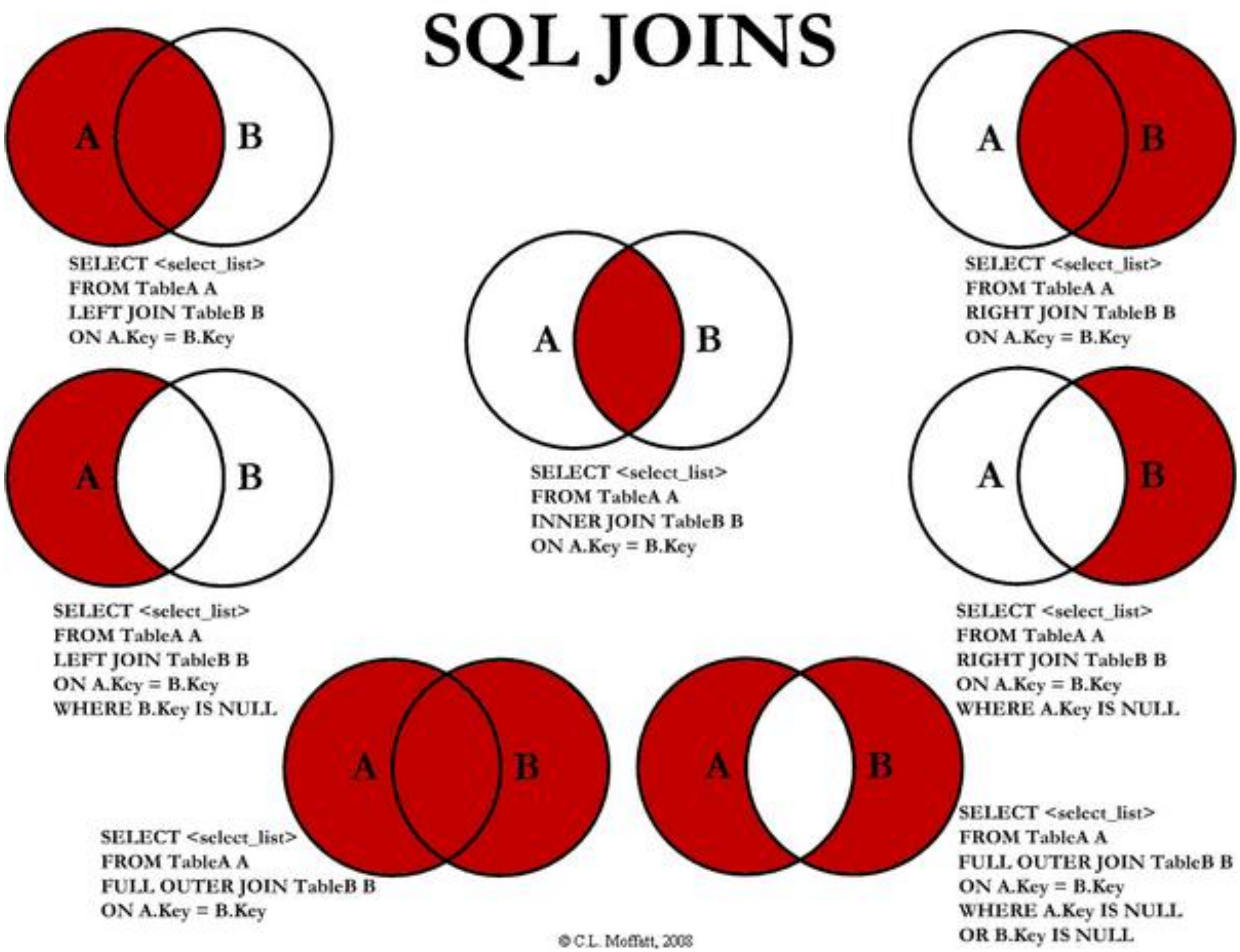
  FROM *course* **AS** *C* **FULL OUTER JOIN** *prereq* **AS** *P* **ON** *C.course_id = P.course_id*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

C.course_id	**C.title**	**C.credits**	**C.dept_name**	**P.course_id**
BIO-301	Genetics	4	Biology	BIO-101
CS-190	Game Design	4	Comp. Sci.	CS-101
CS-347	**NULL**	**NULL**	**NULL**	CS-101
CS-315	Robotics	3	Comp. Sci.	**NULL**

Join Types at a Glance



Source: https://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins

Ordering Results

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Recap: Relational Algebra works on sets
  - i.e., it does not have orderings

- For database applications, ordering is often useful, e.g.,
  - list students ordered by names

```
SELECT   id,name
FROM     student
ORDER BY name
```

  - list instructors ordered by department first, then by name

```
SELECT   id,name,dept_name
FROM     instructor
ORDER BY dept_name, name
```

Limiting Results

- Find the three lecturers with the highest salaries

  SELECT *id,name,salary*
  FROM *instructor*
  ORDER BY *salary* **DESC**
  LIMIT 3;

- *Note:* the **DESC** keyword creates a descending ordering
- **ASC** also exists and creates an ascending ordering
  – also the default when not specifiying the direction

Paging with LIMIT and OFFSET

- Applications, e.g., Web applications, often offer a *paged* view
- Example:
  - Display student list on pages of 100 students
  - with navigation (next page, previous page)

    SELECT *id,name*

    FROM *student*

    ORDER BY *name*

    LIMIT 100

    OFFSET 100;
- **OFFSET** 100 means: skip the first 100 entries
  - i.e., this query would create the second page
- *Note:* offset should only be used with **order by**
  - otherwise, the results are not deterministic

Set Operations

- All courses that are offered in HWS 2017 ***and*** FSS 2018

  (**SELECT** *course_id* **FROM** *section* **WHERE** *sem* = 'HWS' **AND** *year* = 2017)
  INTERSECT
  (**SELECT** *course_id* **FROM** *section* **WHERE** *sem* = 'FSS' **AND** *year* = 2018)

  $\pi_{course\_id}(\sigma_{sem='HWS' \wedge year=2017}(\text{section})) \cap \pi_{course\_id}(\sigma_{sem='FSS' \wedge year=2018}(\text{section}))$

- All courses that are offered in HWS 2017 ***but not** in* FSS 2018

  (**SELECT** *course_id* **FROM** *section* **WHERE** *sem* = 'HWS' **AND** *year* = 2017)
  EXCEPT
  (**SELECT** *course_id* **FROM** *section* **WHERE** *sem* = 'FSS' **AND** *year* = 2018)

  $\pi_{course\_id}(\sigma_{sem='HWS' \wedge year=2017}(\text{section})) - \pi_{course\_id}(\sigma_{sem='FSS' \wedge year=2018}(\text{section}))$

Set Operations

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- All courses that are offered in HWS 2017 ***or*** FSS 2018

  (**SELECT** *course_id* **FROM** *section* **WHERE** *sem* = 'HWS' **AND** *year* = 2017)
  UNION
  (**SELECT** *course_id* **FROM** *section* **WHERE** *sem* = 'FSS' **AND** *year* = 2018)

  $\pi_{course\_id}(\sigma_{sem='HWS' \wedge year=2017}(\text{section})) \cup \pi_{course\_id}(\sigma_{sem='FSS' \wedge year=2018}(\text{section}))$

- Alternative solution

  SELECT course_id
  FROM section
  WHERE (*sem* = 'HWS' **AND** *year* = 2017) **OR** (*sem* = 'FSS' **AND** *year* = 2018)

  $\pi_{course\_id}(\sigma_{(sem='HWS' \wedge year=2017) \vee (sem='FSS' \wedge year=2018))}(\text{section}))$

Aggregate Functions – Examples

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Find the average salary of instructors in the Computer Science department
  - **SELECT AVG** (*salary*)
    FROM *instructor*
    WHERE *dept_name*= ’Comp. Sci.’
- Find the number of tuples in the *course* relation
  - **SELECT COUNT** (*)
    FROM *course*
- Find the total number of instructors who teach a course in the Spring 2010 semester
  - **SELECT COUNT** (**DISTINCT** *ID*)
    FROM *teaches*
    WHERE *semester* = ’Spring’ **AND** *year* = 2010

Why do we need **distinct** here?

Aggregate Functions with Group By

- Find the average salary of instructors in each department

SELECT *dept_name*, **AVG** (*salary*) **AS** *avg_salary*
FROM *instructor*
GROUP BY *dept_name*

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	*salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Aggregate Functions with Group By

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Attributes in **SELECT** clause outside of aggregate functions must appear in **GROUP BY** list

why?

/* erroneous query */
SELECT *dept_name*, *ID*, **AVG** (*salary*) **AS** *avg_salary*
FROM *instructor*
GROUP BY *dept_name*;

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	*Avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Conditions on Aggregate Values

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Find the names and average salaries of all departments whose average salary is greater than 42000
  - **SELECT** *dept_name*, **AVG** (*salary*) **AS** *avg_salary*
    FROM *instructor*
    GROUP BY *dept_name*
    WHERE *avg_salary* > 42000;
- Problem:
  - Aggregation is performed *after* selection and projection
  - Hence, the variable *avg_salary* is not available when the **where** clause is evaluated
- → The above query will not work

Conditions on Aggregate Values

- Find the names and average salaries of all departments whose average salary is greater than 42000
  - **SELECT** *dept_name*, **AVG** (*salary*) **AS** *avg_salary*
    FROM *instructor*
    GROUP BY *dept_name*
    HAVING *avg_salary* > 42000;

performance!

- The **having** clause is evaluated *after* the aggregation
- Hence, it is different from the **where** clause

- Rule of thumb
  - Conditions on aggregate values can only be defined using **having**

NULL Values

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- *null* signifies an unknown value or that a value does not exist
- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
  - can be forbidden by a **not null** constraint
  - keys can never be null!
- The result of any arithmetic expression involving *null* is *null*
- Example: 5 + *null* returns null
- The predicate **is null** can be used to check for null values
- Example: Find all instructors whose salary is null

SELECT *name*
FROM *instructor*
WHERE *salary* **IS NULL**

NULL Values and Three Valued Logic

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Three values – *true*, *false*, *unknown*
- Any comparison with *null* returns *unknown*
  - Example*: 5 < null   or   null <> null   or   null = null*
- Three-valued logic using the value *unknown*:
  - OR: (*unknown* **OR** *true*)   *= true,*
    (*unknown* **OR** *false*) *= unknown*
    (*unknown* ***OR*** *unknown) = unknown*
  - AND: *(true* **AND** *unknown) = unknown,*
    (false **AND** *unknown) = false,*
    (unknown **AND** *unknown) = unknown*
  - NOT*: (***NOT** *unknown) = unknown*
- "*P* **IS UNKNOWN**" evaluates to true if predicate *P* evaluates to *unknown*
- Result of **WHERE** clause predicate is treated as *false*
  if it evaluates to *unknown*

Aggregates and NULL Values

- Total all salaries

  SELECT SUM (*salary*)

  FROM *instructor*

  – Above statement ignores null amounts
  – Result is *null* if there is no non-null amount
- All aggregate operations except **COUNT(*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
  – count returns 0
  – all other aggregates return null

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

Subqueries

- SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **SELECT-FROM-WHERE** expression that is nested within another query
- The nesting can be done in the following SQL query

  SELECT $A_1, A_2, ..., A_n$
  FROM $r_1, r_2, ..., r_m$
  WHERE P

  as follows:
- A_i can be replaced be a subquery that generates a single value
- r_i can be replaced by any valid subquery
- P can be replaced with an expression of the form:

  B <operation> (subquery)

Where *B* is an attribute and <operation> to be defined later

Subqueries in the WHERE Clause

- A common use of subqueries is to perform tests:
  - for set membership
  - for set comparisons
  - for set cardinality

Test for Set Membership

- Find courses offered this term by lectures from the biology department

```
SELECT DISTINCT course_id
FROM teaches
WHERE semester = 'Spring' AND year= 2022 AND ID IN (
    SELECT ID
    FROM instructor
    WHERE dept_name = 'Biology'
)
```

Test for Set Membership

- Find courses offered this term before 9 am or after 5 pm

  SELECT DISTINCT *course_id*

  FROM *section*

  WHERE *semester* = ’Spring’ **AND** *year*= 2022 **AND** time_slot_id **NOT IN** (

      SELECT *time_slot_id*

      FROM *time_slot*

      WHERE *end_time* >= 9 **AND** *start_time* <= 17

)

Test for Set Membership

- Find the total number of (distinct) courses offered by instructors in the biology department

```
SELECT COUNT(DISTINCT course_id)
FROM teaches
WHERE semester = 'Spring' AND year= 2022 AND ID IN (
    SELECT ID
    FROM instructor
    WHERE dept_name = 'Biology'
)
```

- Note: in all of those cases, other (sometimes simpler) solutions are possible
  - In SQL, there are often different ways to solve a problem
  - A question of personal taste
  - But also: a question of performance...

Test for Set Membership

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Find the total number of (distinct) courses offered by instructors in the biology department

```
SELECT COUNT(DISTINCT course_id)
FROM teaches
WHERE semester = 'Spring' AND year= 2022 AND ID IN (
    SELECT ID
    FROM instructor
    WHERE dept_name = 'Biology'
)
```

creates a temporary table

vs.

```
SELECT COUNT (DISTINCT course_id)
FROM teaches, instructor
WHERE teaches.ID = instructor.ID AND instructor.department = 'Biology'
```

computes cartesian product

Set Comparison with SOME

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department

  SELECT DISTINCT *T.name*
  FROM *instructor* **AS** *T*, *instructor* **AS** *S*
  WHERE *T.salary* > *S.salary* **AND** *S.dept name* = 'Biology'

- Same query using > **SOME** clause

  SELECT *name*
  FROM *instructor*
  WHERE *salary* > **SOME** (**SELECT** *salary*
  FROM *instructor*
  WHERE *dept name* = 'Biology')

Set Comparison with ALL

- Find names of instructors with salary greater than that of all instructors in the Biology department

SELECT *name*
FROM *instructor*
WHERE *salary* > **ALL** (**SELECT** *salary*
FROM *instructor*
WHERE *dept name* = ’Biology’)

- Note: we could also achieve this with MIN and MAX aggregates in the subqueries

Definition: Comparisons with SOME

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- F <comp> **SOME** $r \Leftrightarrow \exists\, t \in r$ such that (F <comp> t)
  Where <comp> can be: $<, \leq, >, =, \neq$

(5 < **SOME**

0
5
6

) = true

(read: 5 < some tuple in the relation)

(5 < **SOME**

0
5

) = false

(5 = **SOME**

0
5

) = true

(5 $\neq$ **SOME**

0
5

) = true (since 0 $\neq$ 5)

(= **SOME**) $\equiv$ **IN**

However, ($\neq$ **SOME**) $\not\equiv$ **not in**

Definition: Comparisons with ALL

- F <comp> **ALL** *r* ⇔ ∀ *t* ∈ *r* (F <comp> *t*)

(5 < **ALL**

0
5
6

) = false

(5 < **ALL**

6
10

) = true

(5 = **ALL**

4
5

) = false

(5 ≠ **ALL**

4
6

) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **ALL**) ≠ **NOT IN**

However, (= **ALL**) ≠ **IN**

Existential Quantification in Subqueries

- Select all courses offered this year which are taken by at least one student
  - **SELECT** *course_id*
    FROM *section*
    WHERE *semester* = 'Spring' **AND** *year* = 2022 **AND EXISTS** (
    SELECT *
    FROM *takes*
    WHERE *takes.course_id* = *section.course_id*
    AND *takes.sec_id* = *section.sec_id*
    AND *takes.semester* = *section.semester*)
- The **EXISTS** construct returns the value **true** if the result of the subquery is not empty
  - **EXISTS** $r \Leftrightarrow r \neq \emptyset$
  - **NOT EXISTS** $r \Leftrightarrow r = \emptyset$

Subqueries with NOT EXISTS

- Find all students who have taken all courses offered in the Biology department

```
SELECT DISTINCT S.ID, S.name
FROM student AS S
WHERE NOT EXISTS (   (SELECT course_id
                      FROM course
                      WHERE dept_name = 'Biology')
                    EXCEPT
                     (SELECT T.course_id
                      FROM takes AS T
                      WHERE S.ID = T.ID))
```

  - First nested query lists all courses offered in Biology
  - Second nested query lists all courses a particular student took
- Note that $X - Y = \emptyset \iff X \subseteq Y$
- Note: Cannot write this query using **= all** and its variants

Test for Duplicate Tuples

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Find all courses that were offered at most once in 2009

  SELECT *T.course_id*
  FROM *course* **AS** *T*
  WHERE UNQIUE (SELECT *R.course_id*
  FROM *section* **as** *R*
  WHERE *T.course_id*= *R.course_id* ***AND***
  R.year = 2009)

- The **unique** construct evaluates to “true” if a given subquery contains no duplicates
- With **not unique**, we could query for courses that were offered more than once

Subqueries in the FROM Clause

- So far, we have considered subqueries in the **where** clause
- Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

  SELECT *dept_name*, *avg_salary*
  FROM (
      SELECT *dept_name*, **AVG** (*salary*) **AS** *avg_salary*
      FROM *instructor*
      GROUP BY *dept_name*
)
  WHERE *avg_salary* > 42000;

- Note that we do not need to use the **having** clause
  - why?

Creating Temporary Relations Using WITH

UNIVERSITY OF MANNHEIM
Data and Web Science Group

- Find all departments with the maximum budget

```
WITH max_budget (value) AS (
        SELECT MAX(budget)
        FROM department
)
SELECT department.name
FROM department, max_budget
WHERE department.budget = max_budget.value
```

this defines the structure of the temporary relation (datatypes are implicit)

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs

Creating Temporary Relations Using WITH

- A more complex example involving two temporary relations:

WITH *dept _total* (*dept_name*, *value*) **AS** (
  SELECT *dept_name*, **SUM**(*salary*)
  FROM *instructor*
  GROUP BY *dept_name*
),
dept_total_avg(*value*) **as** (
  SELECT AVG(*value*)
  FROM *dept_total*
)
select *dept_name*
from *dept_total*, *dept_total_avg*
where *dept_total.value* > *dept_total_avg.value*

Find all departments where the total salary is greater than the average of the total salary at all departments

Scalar Subqueries in the SELECT Part

- List all departments along with the number of instructors in each department

  SELECT *dept_name*, (

      SELECT COUNT(*)

      FROM *instructor*

      WHERE *department.dept_name* = *instructor.dept_name*

)**AS** *num_instructors*

  FROM *department*;

- Scalar subqueries return a single result
  - More specifically: a single *tuple*
- Runtime error if subquery returns more than one result tuple

Summary of Subqueries

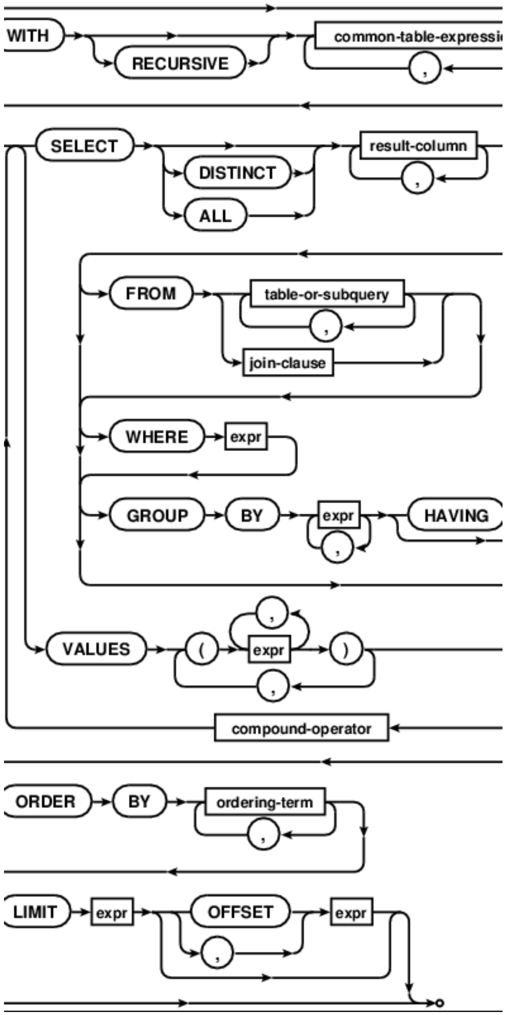
- SELECT queries are the most often used part of SQL
- Their basic structure is simple, but subqueries are a powerful means to make them quite expressive

  SELECT $A_1, A_2, ..., A_n$
  FROM $r_1, r_2, ..., r_m$
  WHERE P

- Subqueries in **SELECT** part ($A_1, A_2, ..., A_n$)
  - Scalar subqueries (single values, like aggregates)
- Subqueries in **FROM** part ($r_1, r_2, ..., r_m$)
  - Temporary relations (can also be defined using **WITH**)
- Subqueries in **WHERE** part (P)
  - Set comparisons, empty sets, test for duplicates
  - Universal and existential quantification

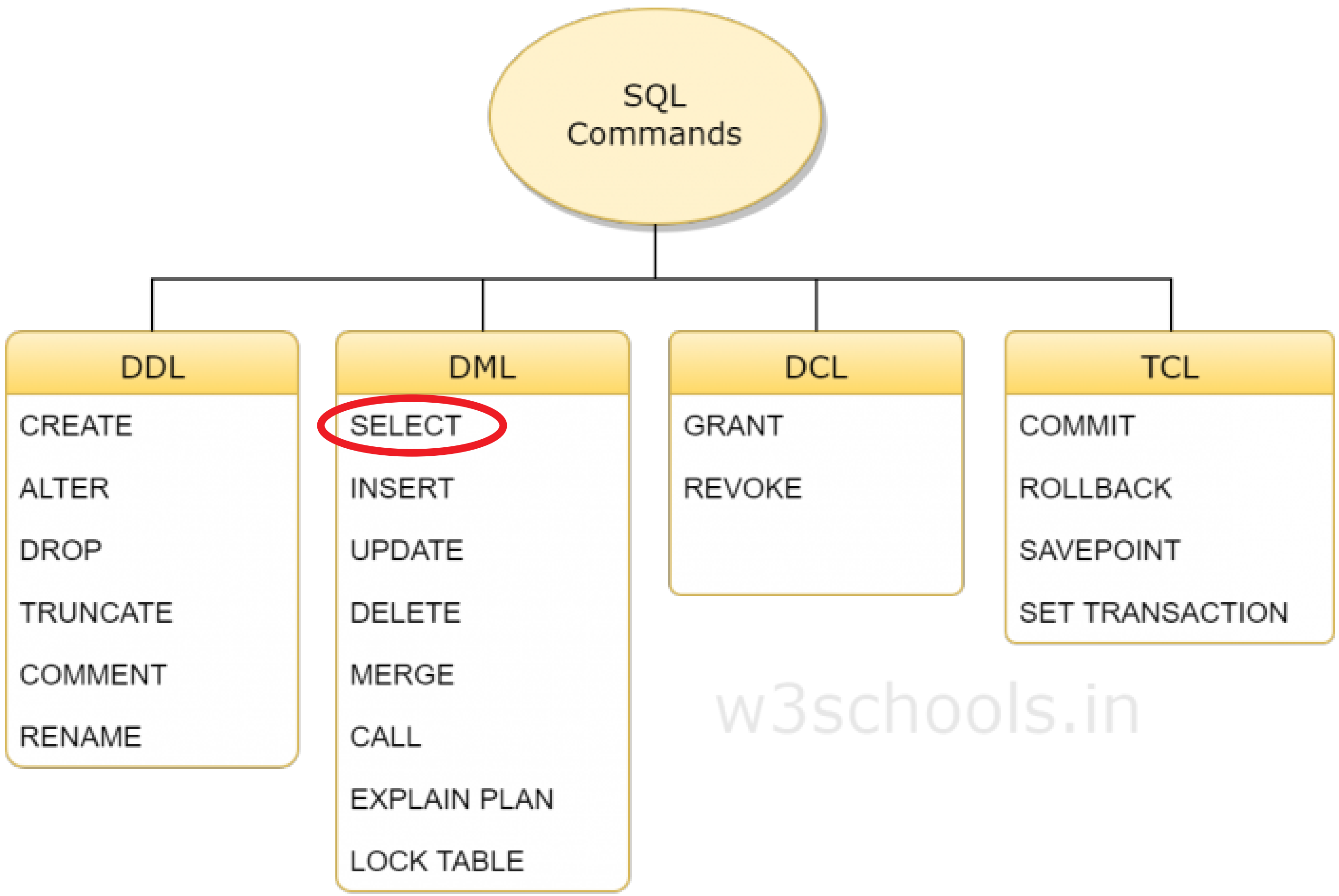
Summary: SQL SELECT at a Glance

- The tool support of SQL varies
- what we have covered here is standard SQL
  - Supported by *most* tools



Parts of SQL: The Big Picture

UNIVERSITY OF MANNHEIM
Data and Web Science Group



Source: https://www.w3schools.in/mysql/ddl-dml-dcl/

Summary and Take Aways

- SQL is a standarized language for relational databases
  - DML: Data Manipulation Language
- DML
  - Read data from tables using SELECT
- Coming Up:
  - Writing data to tables
  - Creating and changing tables
  - Rights & Roles
  - ...

Questions?