

# Database Technology SQL Part 1

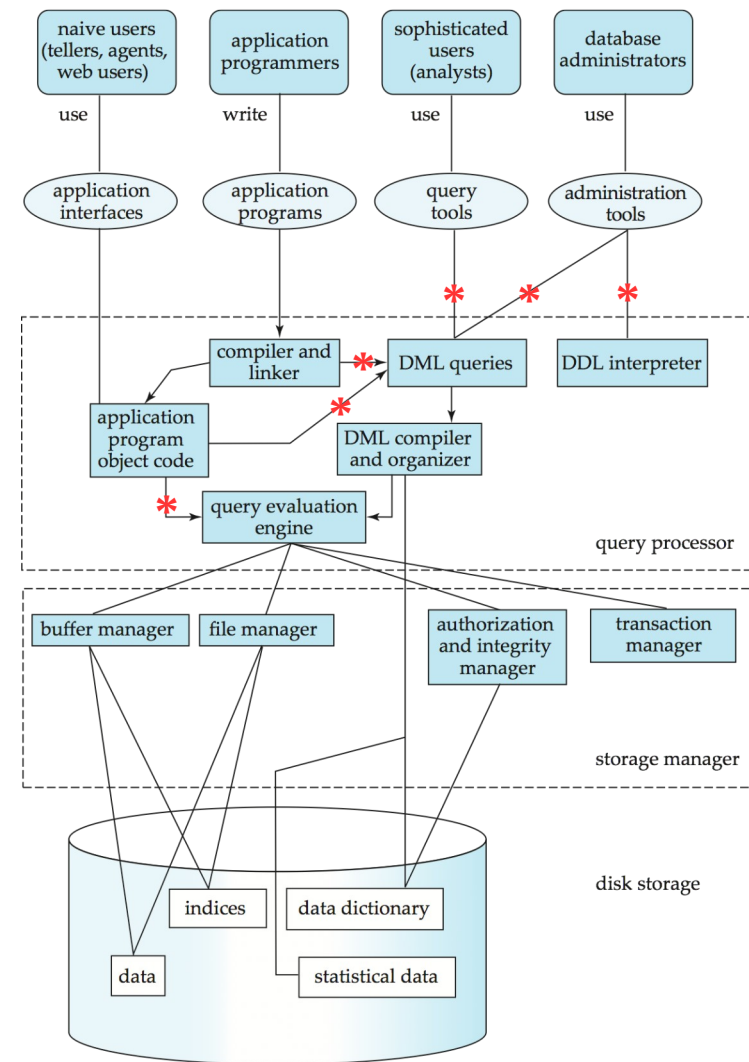


# Outline

- 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



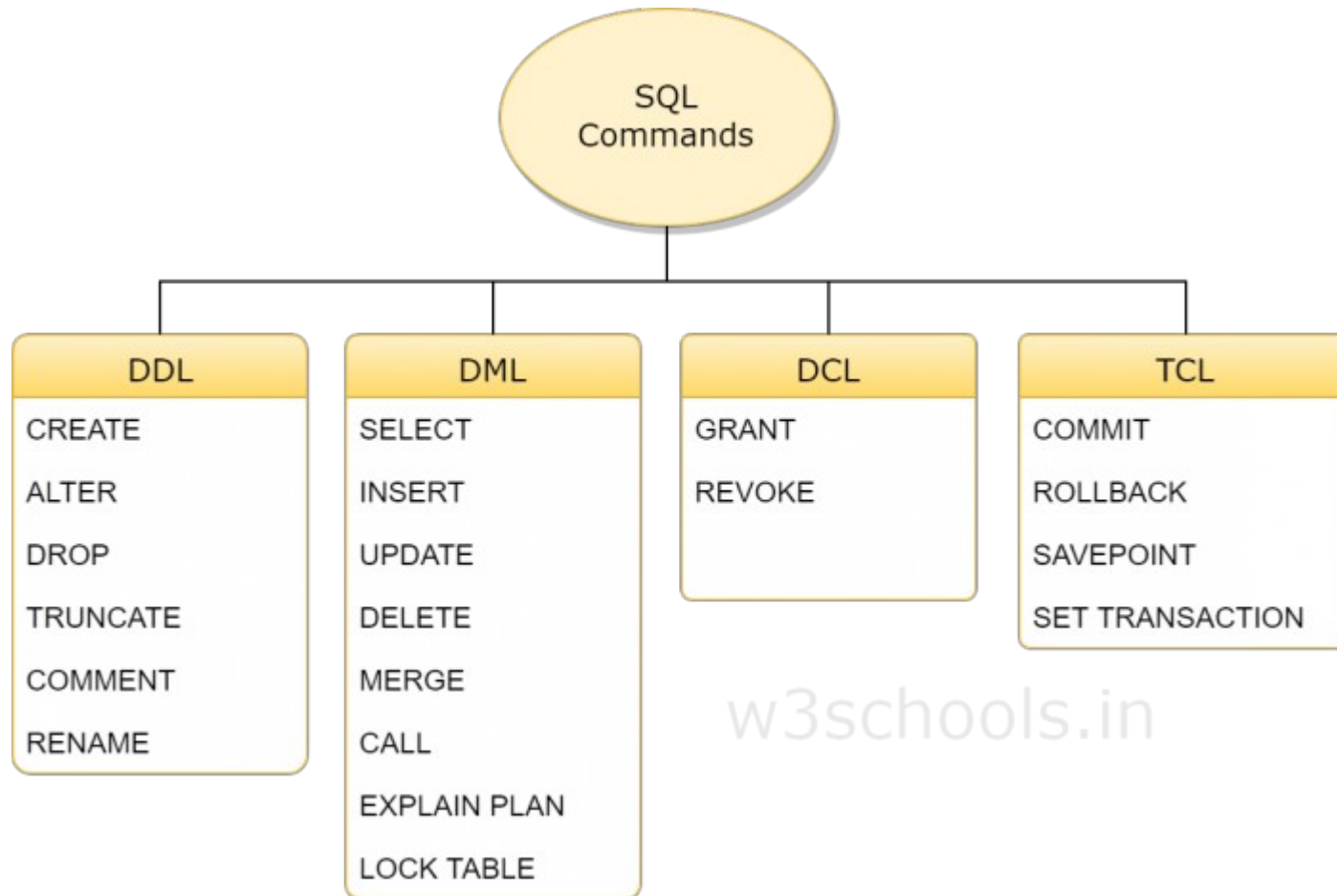
# History

- 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
- 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!

Naming became  
Y2K compliant! ;-)



# 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:
  - $\Pi_{\text{name}}(\textit{instructor})$

# A Note on Case Sensitivity

- 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:

$$\rho_{\text{payment} \leftarrow \text{salary}} (\Pi_{\text{name, salary}} (\text{instructor}))$$



# The Select Clause

- 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'

FOO
-----

**select** '437' **as** *FOO*

437
-----

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

**select** *name*, 'Instructor' **as** *role* **from** *instructor*

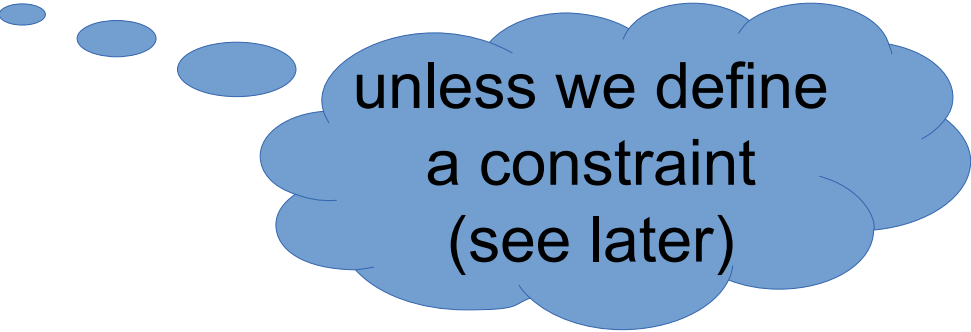
**union**

**select** *name*, 'Student' **as** *role* **from** *student*

name	role
Smith	Instructor
Einstein	Instructor
...	...
Johnson	Student
...	...

# Duplicates

- 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*



# 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
- In relational algebra: combination of selection and projection

```
select name  
from instructor  
where dept_name = 'Comp. Sci.'
```

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

# 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
```

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

- 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

- 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
- 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:
  - $\rho_{instructor.ID \leftarrow ID}(instructor) \times \rho_{teaches.ID \leftarrow ID}(teaches)$



but is that useful?

# Cartesian Product

instructor				teaches																											
ID	name	dept_name	salary	ID	course_id	sec_id	semester	year																							
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009																							
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010																							
15151	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2009																							
22222									12121	FIN-201	1	Spring	2010																		
32343														15151	MU-199	1	Spring	2010													
33456																			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:

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

# Cartesian Product

<i>instructor</i>				<i>teaches</i>				
<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010
15151								2009
22222								2010
32343								2010
33456								2009
<i>Inst.ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>teaches.ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
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
<del>10101</del>	<del>Srinivasan</del>	<del>Comp. Sci.</del>	<del>65000</del>	<del>12121</del>	<del>FIN-201</del>	<del>1</del>	<del>Spring</del>	<del>2010</del>
<del>10101</del>	<del>Srinivasan</del>	<del>Comp. Sci.</del>	<del>65000</del>	<del>15151</del>	<del>MU-199</del>	<del>1</del>	<del>Spring</del>	<del>2010</del>
<del>10101</del>	<del>Srinivasan</del>	<del>Comp. Sci.</del>	<del>65000</del>	<del>22222</del>	<del>PHY 101</del>	<del>1</del>	<del>Fall</del>	<del>2009</del>
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
<del>12121</del>	<del>Wu</del>	<del>Finance</del>	<del>90000</del>	<del>10101</del>	<del>CS-101</del>	<del>1</del>	<del>Fall</del>	<del>2009</del>
<del>12121</del>	<del>Wu</del>	<del>Finance</del>	<del>90000</del>	<del>10101</del>	<del>CS-315</del>	<del>1</del>	<del>Spring</del>	<del>2010</del>
<del>12121</del>	<del>Wu</del>	<del>Finance</del>	<del>90000</del>	<del>10101</del>	<del>CS-347</del>	<del>1</del>	<del>Fall</del>	<del>2009</del>
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2010
<del>12121</del>	<del>Wu</del>	<del>Finance</del>	<del>90000</del>	<del>15151</del>	<del>MU-199</del>	<del>1</del>	<del>Spring</del>	<del>2010</del>
<del>12121</del>	<del>Wu</del>	<del>Finance</del>	<del>90000</del>	<del>22222</del>	<del>PHY 101</del>	<del>1</del>	<del>Fall</del>	<del>2009</del>
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

# 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'
```

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

# Cartesian Product

*instructor*

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151			
22222			
32343			
33456			

*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	Finance	90000	10101	CS-347	1	Fall	2009
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2010
12121	Wu	Finance	90000	15151	MU 199	1	Spring	2010
12121	Wu	Finance	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.'
```

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

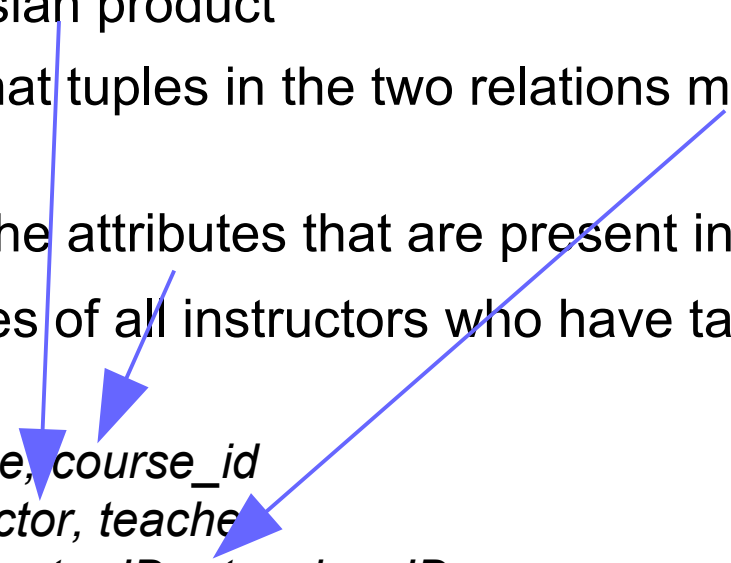
- What happens if we omit the **distinct** here?

# Join Operations

- **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

- 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

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

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

<i>course_id</i>	<i>prereq_id</i>
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.prereq_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
- **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

for the moment:  
keyword for “a blank cell”

<i>Join types</i>	<i>Join Conditions</i>
<b>inner join</b> <b>left outer join</b> <b>right outer join</b> <b>full outer join</b>	<b>natural</b> <b>on</b> <predicate> <b>using</b> ( $A_1, A_1, \dots, A_n$ )

# 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.prereq_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

- 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
```

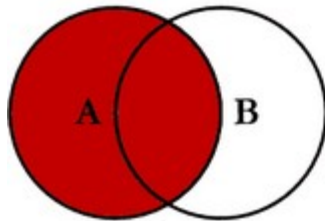
course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

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

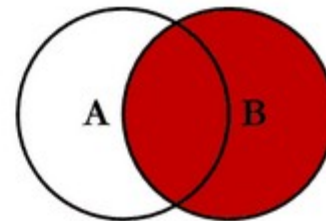
C.course_id	C.title	C.credits	C.dept_name	P.prereq_id
BIO-301	Genetics	4	Biology	BIO-101
CS-190	Game Design	4	Comp. Sci.	CS-101
CS-347	<b>null</b>	<b>null</b>	<b>null</b>	CS-101
CS-315	Robotics	3	Comp. Sci.	<b>null</b>

# Join Types at a Glance

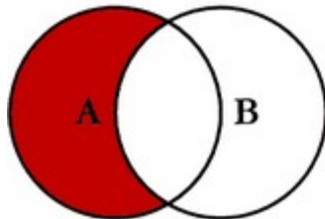
## SQL JOINS



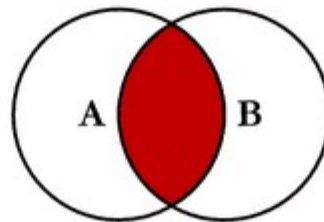
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key
```



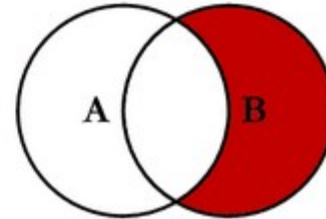
```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key
```



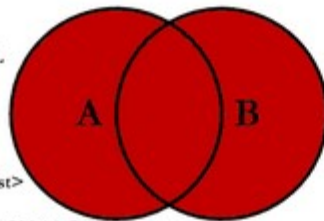
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key  
WHERE B.Key IS NULL
```



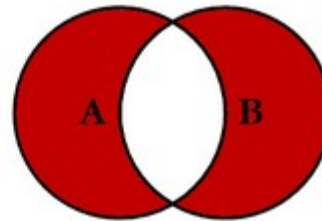
```
SELECT <select_list>  
FROM TableA A  
INNER JOIN TableB B  
ON A.Key = B.Key
```



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



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



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

© C.L. Moffatt, 2008

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

# Ordering Results

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

# Set Operations

- 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)**

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

- Alternative solution

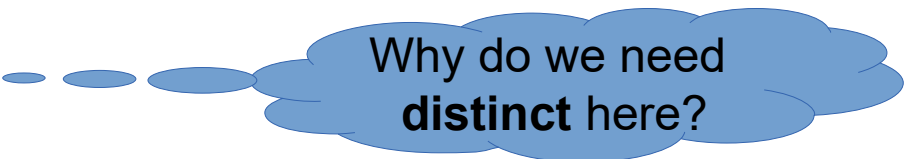
**(select course\_id from section where**

**((sem = 'HWS' and year = 2017) or (sem = 'FSS' and year = 2018))**

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

# Aggregate Functions – Examples

- 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*;

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
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

<i>dept_name</i>	<i>avg_salary</i>
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

# Aggregate Functions with Group By

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list

*/\* erroneous query \*/*

```
select dept_name, ID, avg (salary)
from instructor
group by dept_name;
```

why?

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

- 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;`
- 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**



performance!



# NULL Values

- *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:  $5 + \text{null}$  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

- 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:  $(unknown \text{ or } true) = true$ ,  
 $(unknown \text{ or } false) = unknown$   
 $(unknown \text{ or } unknown) = unknown$
  - AND:  $(true \text{ and } unknown) = unknown$ ,  
 $(false \text{ and } unknown) = false$ ,  
 $(unknown \text{ and } unknown) = unknown$
  - NOT:  $(\text{not } unknown) = unknown$
- “*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.	null
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	null
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	null
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	null

# 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  $A_1, A_2, \dots, A_n$   
from  $r_1, r_2, \dots, r_m$   
where  $P$ 
```

as follows:

- $A_i$  can be replaced by 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:

$B \text{ <operation> (subquery)}$

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');
```

- 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

- 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

- $F <\text{comp}> \text{some } r \Leftrightarrow \exists t \in r \text{ such that } (F <\text{comp}> t)$

Where  $<\text{comp}>$  can be:  $<$ ,  $\leq$ ,  $>$ ,  $=$ ,  $\neq$

$(5 < \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{true}$  (read:  $5 <$  some tuple in the relation)

$(5 < \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{false}$

$(5 = \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$

$(5 \neq \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true (since } 0 \neq 5)$

$(= \text{some}) \equiv \text{in}$

However,  $(\neq \text{some}) \not\equiv \text{not in}$

# Definition: Comparisons with ALL

- $F <\text{comp}> \mathbf{all} \ r \Leftrightarrow \forall t \in r \ (F <\text{comp}> t)$

$$(5 < \mathbf{all} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{false}$$

$$(5 < \mathbf{all} \begin{array}{|c|} \hline 6 \\ \hline 10 \\ \hline \end{array}) = \text{true}$$

$$(5 = \mathbf{all} \begin{array}{|c|} \hline 4 \\ \hline 5 \\ \hline \end{array}) = \text{false}$$

$$(5 \neq \mathbf{all} \begin{array}{|c|} \hline 4 \\ \hline 6 \\ \hline \end{array}) = \text{true (since } 5 \neq 4 \text{ and } 5 \neq 6)$$

$(\neq \mathbf{all}) \equiv \mathbf{not\ in}$

However,  $(= \mathbf{all}) \not\equiv \mathbf{in}$

# 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  $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using **= all** and its variants

# Test for Duplicate Tuples

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

```
select T.course_id
from course as T
where unique (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;
```

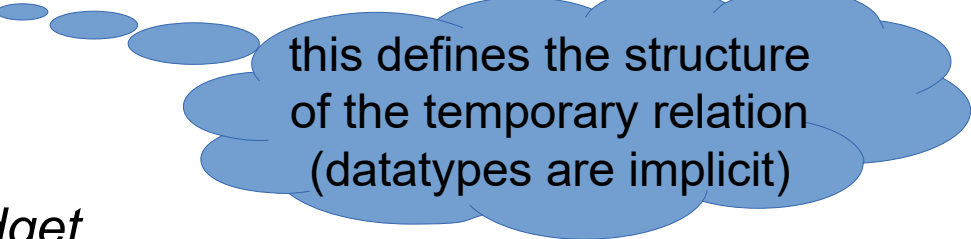
- Note that we do not need to use the **having** clause
  - why?



# Creating Temporary Relations Using WITH

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

**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;
```

# 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

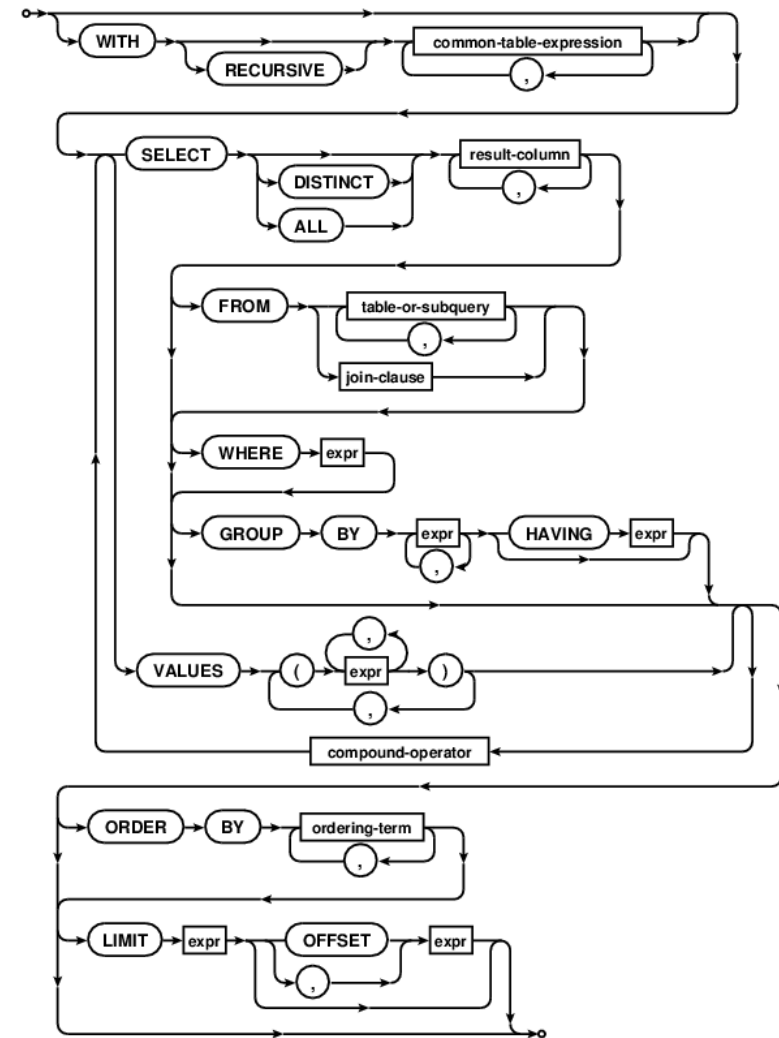
- SELECT queries are the most often used part of SQL
- Their basic structure is simple, but subqueries are a powerful means to make them quite expressive

**select**  $A_1, A_2, \dots, A_n$   
**from**  $r_1, r_2, \dots, r_m$   
**where**  $P$

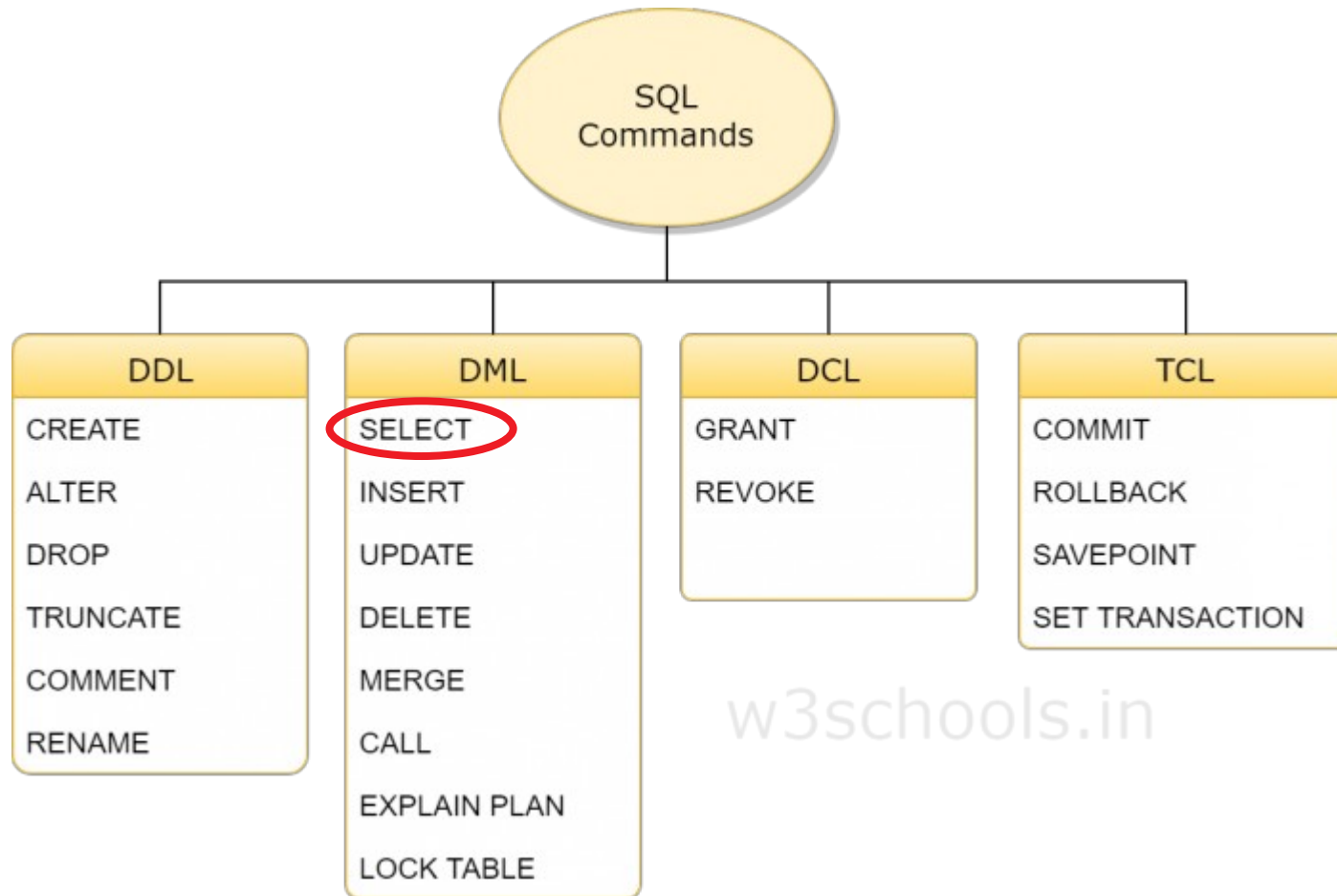
- Subqueries in **select** part ( $A_1, A_2, \dots, A_n$ )
  - Scalar subqueries (single values, like aggregates)
- Subqueries in **from** part ( $r_1, r_2, \dots, r_m$ )
  - Temporary relations (can also be defined using **with**)
- Subqueries in **where** part ( $P$ )
  - Set comparisons, empty sets, 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



# Recap: The Big Picture



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

# Summary and Take Aways

- SQL is a standardized 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?

