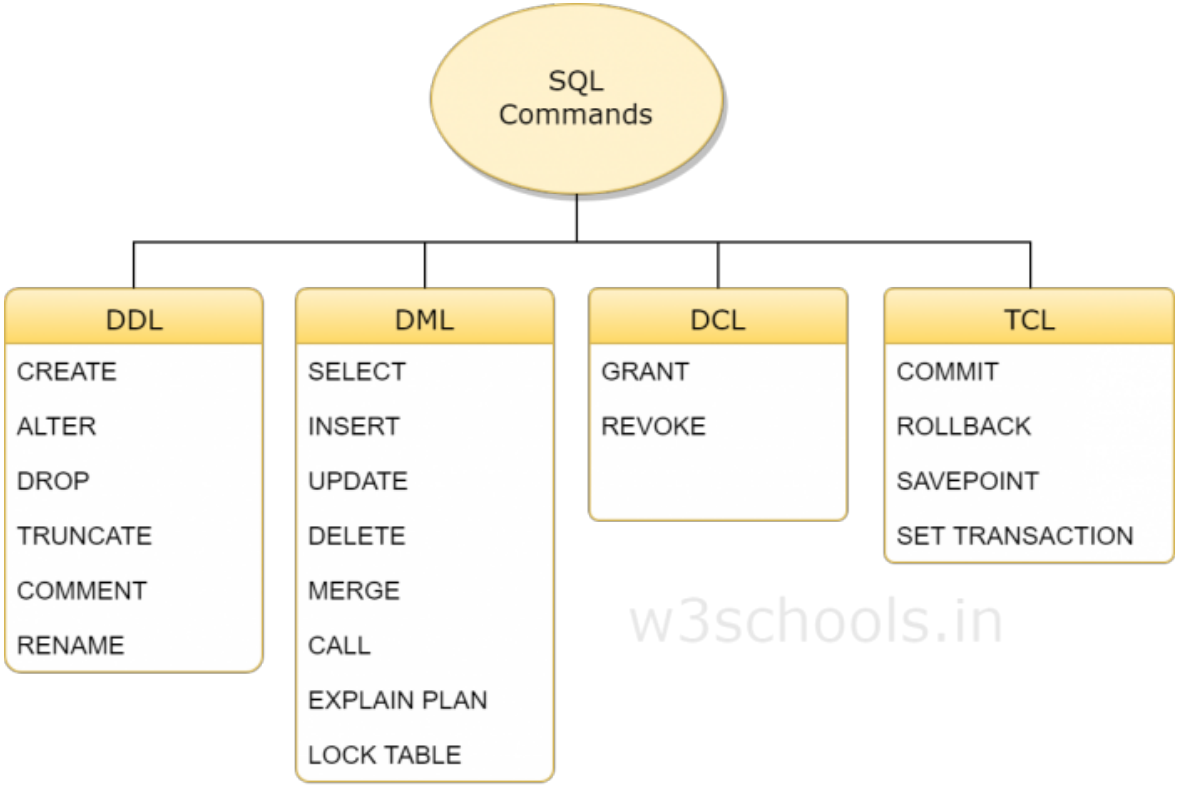
UNIVERSITÄT MANNHEIM

Database Technology
SQL Part 2

Heiko Paulheim

Looking Back

- We have seen
  - Table definition, creation, and removal
  - Reading data from tables



Outline

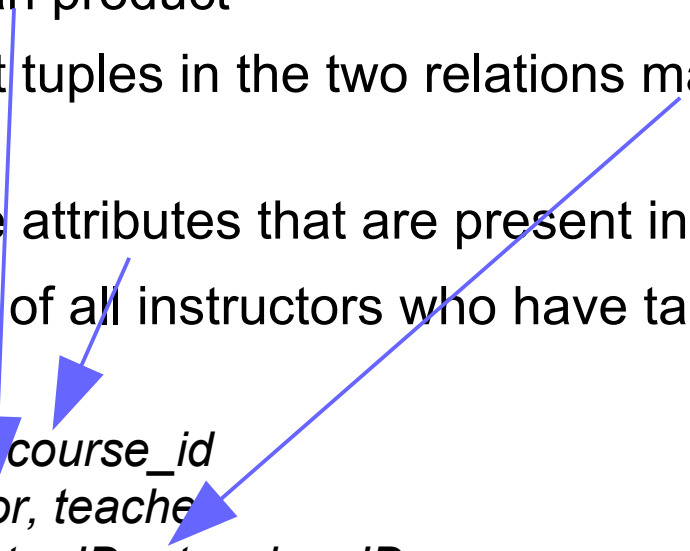
- Join Expressions
- Modifications of the database
  - Deletion of tuples from a given relation
  - Insertion of new tuples into a given relation
  - Updating of values in some tuples in a given relation
- Views
- Integrity Constraints
- SQL Data Types
- Authorization

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

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.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

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.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

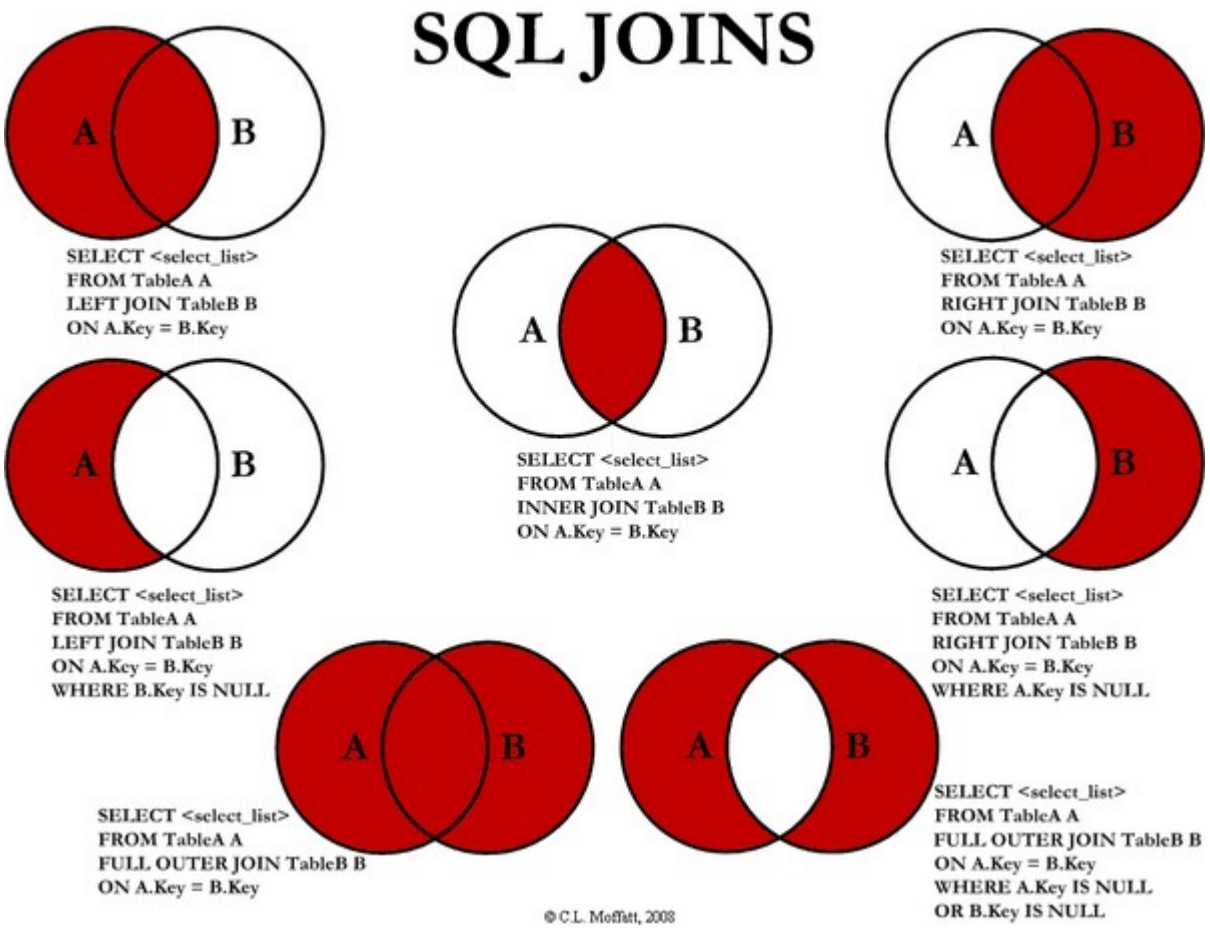
  select *C.course_id, C.title, C.credits, C.dept_name, P.prereq_id*
  from *course* **as** *C* **full outer join** *prereq* **as** *P*
  on *C.course_id = P.course_id*

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	**null**	**null**	**null**	CS-101
CS-315	Robotics	3	Comp. Sci.	**null**

Join Types at a Glance



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

Deleting from a Relation

- **Delete**
  - Remove all tuples from the *student* relation
  - **delete from** *instructor*
  - May be refined (e.g., only removing *specific* tuples)
    - **delete from** *instructor* **where ...**

Deleting from a Relation

- Delete all instructors from the Finance department

```
delete from instructor
where dept_name= 'Finance';
```

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building

```
delete from instructor
where dept name in (select dept name
                    from department
                    where building = 'Watson');
```

Deleting from a Relation

- Delete all instructors whose salary is less than the average salary of instructors

```
delete from instructor
where salary < (select avg (salary)
                from instructor);
```

- This would delete five tuples
  - But then, the average changes!

- How does the query behave if the deletion is processed one by one?

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

Deleting from a Relation

- Delete all instructors whose salary is less than the average salary of instructors

  delete from *instructor*
  where *salary* < (**select avg** (*salary*)
  from *instructor*);

- Processing this query in SQL
  - First, the **select** query is evaluated
    - i.e., the result is now treated as a constant
  - Then, the **delete** statement is executed

DELETE vs. TRUNCATE

- All records from a table can also be removed using

  truncate table *instructor;*

  Difference to

  delete from *instructor;*

  ?

- **delete** keeps the table and deletes only the data
- **truncate** drops and re-creates the table
  - much faster
  - but cannot be undone
- **delete** is DML, **truncate** is DDL
  - Different rights may be necessary (see later!)

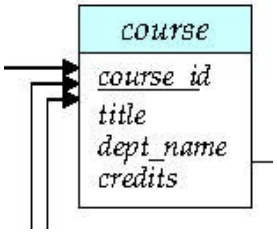
Description

`TRUNCATE TABLE` empties a table completely. It requires the `DROP` privilege (before 5.1.16, it required the `DELETE` privilege.) See `GRANT`.

Insertion into a Relation

- Add a new tuple to *course*

  insert into *course*
  values (’CS-437’, ’Database Systems’, ’Comp. Sci.’, 4);



- or equivalently

  insert into *course* (*course_id*, *title*, *dept_name*, *credits*)
  values (’CS-437’, ’Database Systems’, ’Comp. Sci.’, 4);

- Add a new tuple to *student* with *tot_creds* set to null

  insert into *student*
  values (’3003’, ’Green’, ’Finance’, *null*);

Insertion of Data from Other Tables

- Add all instructors to the *student* relation with tot_creds set to 0

  insert into *student*
  select *ID, name, dept_name, 0*
  from *instructor*

- As in the deletion example, the **select from where** statement is evaluated fully before any of its results are inserted into the relation

  Otherwise queries like

  insert into *table*1 **select** * **from** *table*1

  would cause problems

Inserting Data into Relations with Constraints

- Effect of primary key constraints:
  - **insert into** *instructor* **values** (‘10211’, ’Smith’, ’Biology’, 66000);
  - **insert into** *instructor* **values** (‘10211’, ’Einstein’, ’Physics’, 95000);
  - ...and we defined ID the primary key!

- Effect of **not null** constraints
  - **insert into** *instructor* **values** (‘10211’, **null**, ’Biology’, 66000);

- Recap: DBMS takes care of *data integrity*

Updating Data

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
- Write two **update** statements:

```
update instructor
    set salary = salary * 1.03
    where salary > 100000;
update instructor
    set salary = salary * 1.05
    where salary <= 100000;
```

- The order is important
- Can be done better using the **case** statement (next slide)

Conditional Updates with case Statement

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%

```
update instructor
        set salary = case
                when salary <= 100000 then salary * 1.05
                else salary * 1.03
                end
```

Updates with Subqueries

- Recompute and update tot_creds value for all students

```
update student S
set tot_cred = (select sum(credits)
                from takes, course
                where takes.course_id = course.course_id
                  and S.ID= takes.ID.and takes.grade <> 'F'
                  and takes.grade is not null);
```

- Sets *tot_creds* to null for students who have not taken any course
- Instead of **sum**(*credits*), use:

```
case
   when sum(credits) is not null then sum(credits)
   else 0
end
```

Views

- Recap: logical database model
  - The relations in the database and their attributes
- Views:
  - Virtual relations
  - Different from those in the database
  - But with the same data
  - ...hide data from users
- Example: instructors' names and departments without salaries, i.e.,

  select *ID*, *name*, *dept_name*
  from *instructor*

Views

- Example: some users may see employees with salaries, others only without salary

- How about two tables
  - One with salaries
  - One without salaries
- ?

Defining Views

- A view is defined using the **create view** statement:
  create view *v* **as** < query expression >
  - <query expression> is any legal SQL expression
  - the view name is represented by *v*
- Once the view has been created
  - it can be addressed as *v* as any other relations
  - it will always contain the data read by the SQL expression
    - live, not at the time of definition!

Example Views

- Instructors without their salary

  create view *faculty* **as**
  select *ID*, *name*, *dept_name*
  from *instructor*

- Using the view: find all instructors in the Biology department

  select *name*
  from *faculty*
  where *dept_name* = 'Biology';

- Create a view of department salary totals

  create view *departments_total_salary*(*dept_name*, *total_salary*)
  as
  select *dept_name*, **sum** (*salary*)
  from *instructor*
  group by *dept_name*;

Defining Views using other Views

- **create view** *physics_fall_2009* **as**
  select *course*.*course_id*, *sec_id*, *building*, *room_number*
  from *course*, *section*
  where *course*.*course_id* = *section*.*course_id*
  and *course*.*dept_name* = 'Physics'
  and *section*.*semester* = 'Fall'
  and *section*.*year* = '2009';
- **create view** *p*
  select *cour*
  from *physic*
  where *build*

```
create view physics_fall_2009_watson as
(select course_id, room_number
from (select course.course_id, building, room_number
        from course, section
        where course.course_id = section.course_id
            and course.dept_name = 'Physics'
            and section.semester = 'Fall'
            and section.year = '2009')
where building= 'Watson';
```

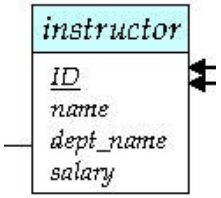
Defining Views using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
  - i.e., the *depends on* relation is transitive
- A view relation v is said to be *recursive* if it depends on itself

Updating Views

- Definition of a simple view (recap: instructors without salaries):

  create view *faculty* **as**
  select *ID*, *name*, *dept_name*
  from *instructor*



- Add a new tuple to *faculty* view which we defined earlier

  insert into *faculty* **values** ('30765', 'Green', 'Music');

- This insertion must be represented by the insertion of the tuple

  ('30765', 'Green', 'Music', null)

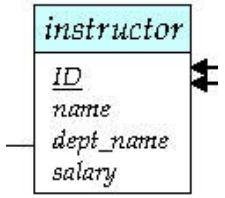
  into the *instructor* relation

This can only work if salary is *not* defined as **not null**!

Updating Views

- Consider the view

  create view *biology_faculty* **as**
  select *ID,name*
  from *faculty*
  where *dept_name* = ‘Biology’;



- and

  insert into biology_faculty
  values (43278,‘Smith’);

- Would this lead to

  insert into *instructor* **values** (43278,’Smith’,’Biology’,null);

  ?

Updating Views

- Most **where** constraints cannot be translated into a value to insert
- Consider

  where *dept_name* = ‘Biology’ **or** *dept_name* = ‘Physics’

  or

  where *salary* > 50000

- Hence, **where** clauses are typically not translated into a value

Updating Views

- Other example used before

  create view *departments_total_salary*(*dept_name*, *total_salary*)
  as
  select *dept_name*, **sum** (*salary*)
  from *instructor*
  group by *dept_name*;

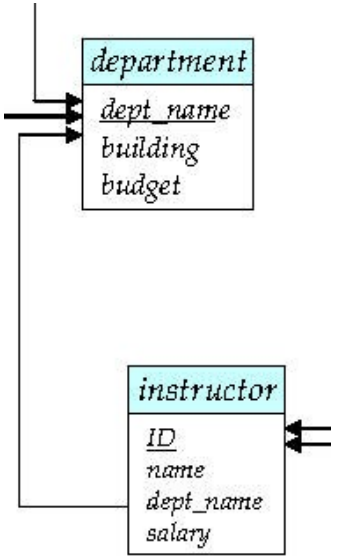
- What should happen upon

  update *departments_total_salary*
  set *total_salary* = *total_salary* * 1.05
  where *dept_name* = “Comp. Sci.”;

  ?

Updating Views

- **create view** *instructor_info* **as**
  select *ID*, *name*, *building*
  from *instructor*, *department*
  where *instructor*.*dept_name*= *department*.*dept_name*;
- **insert into** *instructor_info* **values** (’69987’, ’White’, ’Taylor’);
  - which department, if multiple departments are in Taylor?
  - what if no department is in Taylor?



Updateable Views

- A view is called *updateable* if
  - The **from** clause has only one database relation
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group** by or **having** clause

- Most DMBS only allow updates on such views!

Materialized vs. Non-Materialized Views

- Normal views are not materialized
  - When issuing a **select** against a view, the underlying data is created on the fly
  - Pro: guarantees recent and non-redundant data, saves space
  - Con: some views may be expensive to compute (e.g., extensive use of aggregates)
- **Materializing a view**: create a physical table containing all the tuples in the result of the query defining the view
  - If relations used in the query are updated, the materialized view result becomes out of date
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated

Integrity Constraints

- Data errors may occur due to, e.g.,
  - Accidental wrong entries in form fields
  - Faulty application program code
  - Deliberate attacks
- Integrity constraints
  - guard against damage to the database
  - ensuring that authorized changes to the database do not result in a loss of data consistency
- Examples
  - A checking account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $4.00 an hour
  - A customer must have a (non-null) phone number

Integrity Constraints on a Single Relation

- We have already encountered
  - **not null**
  - **primary** and **foreign key**
- We will get to know
  - **unique**
  - **check** (P), where P is a predicate

NOT NULL and UNIQUE Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**

    name **varchar**(20) **not null**
    budget **numeric**(12,2) **not null**
- **unique** (A_1, A_2, …, A_m)
  - The unique specification states that the attributes *A*1, *A*2, … *A*m form a candidate key
  - Candidate keys are permitted to be null (in contrast to primary keys)

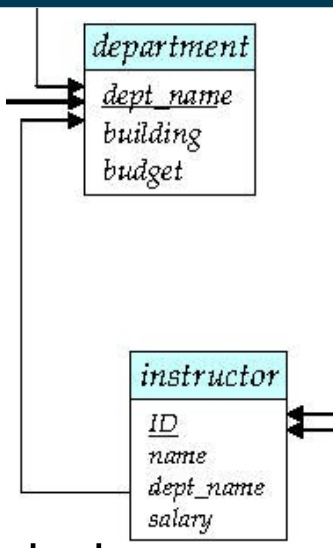
The CHECK Constraint

- **check** (P)
  - where P is a predicate

- Example: ensure that semester is either fall or spring

```
create table section (
  course_id varchar (8),
  sec_id varchar (8),
  semester varchar (6),
  year numeric (4,0),
  building varchar (15),
  room_number varchar (7),
  time slot id varchar (4),
  primary key (course_id, sec_id, semester, year),
  check (semester in ('Fall', 'Spring'))
);
```

Foreign Keys and Referential Integrity

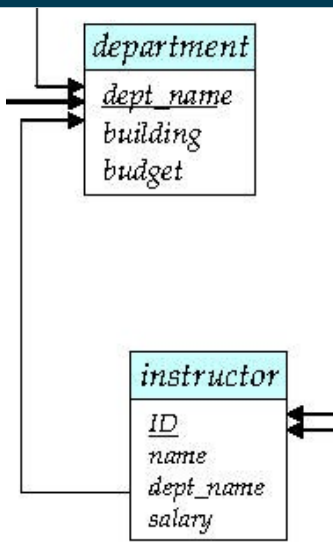


- Example:
  - instructors have a department
  - **each** department should also appear in the *department* relation
- Definition:
  - Let A be a set of attributes
  - Let R and S be two relations that contain attributes A and where A is the primary key of S
  - A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S

Cascading Actions in Referential Integrity

- Example:
  - instructors have a department
  - **each** department should also appear in the *department* relation
- How to *ensure* referential integrity?
  - i.e., what happens if a department is deleted from the *department* relation
- Possible approaches
  - Reject the deletion
  - Delete all instructors as well
  - Set the department of those instructors to **null**

default action



Cascading Actions in Referential Integrity

- Cascading updates
  - If a foreign key is changed (e.g., renaming a department)
  - ...then rename in all referring relations
- Cascading deletions
  - If a foreign key is deleted (e.g., deleting a department)
  - ...then delete all rows in referring relations
- **create table** *instructor* (

  …

  dept_name **varchar**(20),

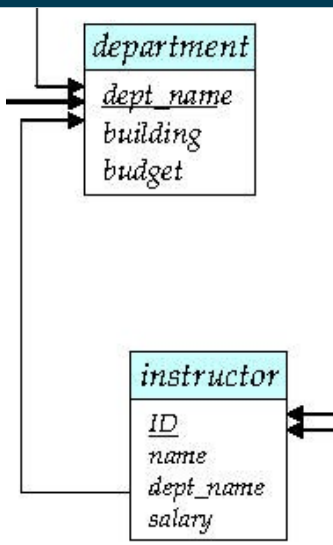
  foreign key (*dept_name*) **references** *department*

  on delete cascade

  on update cascade,

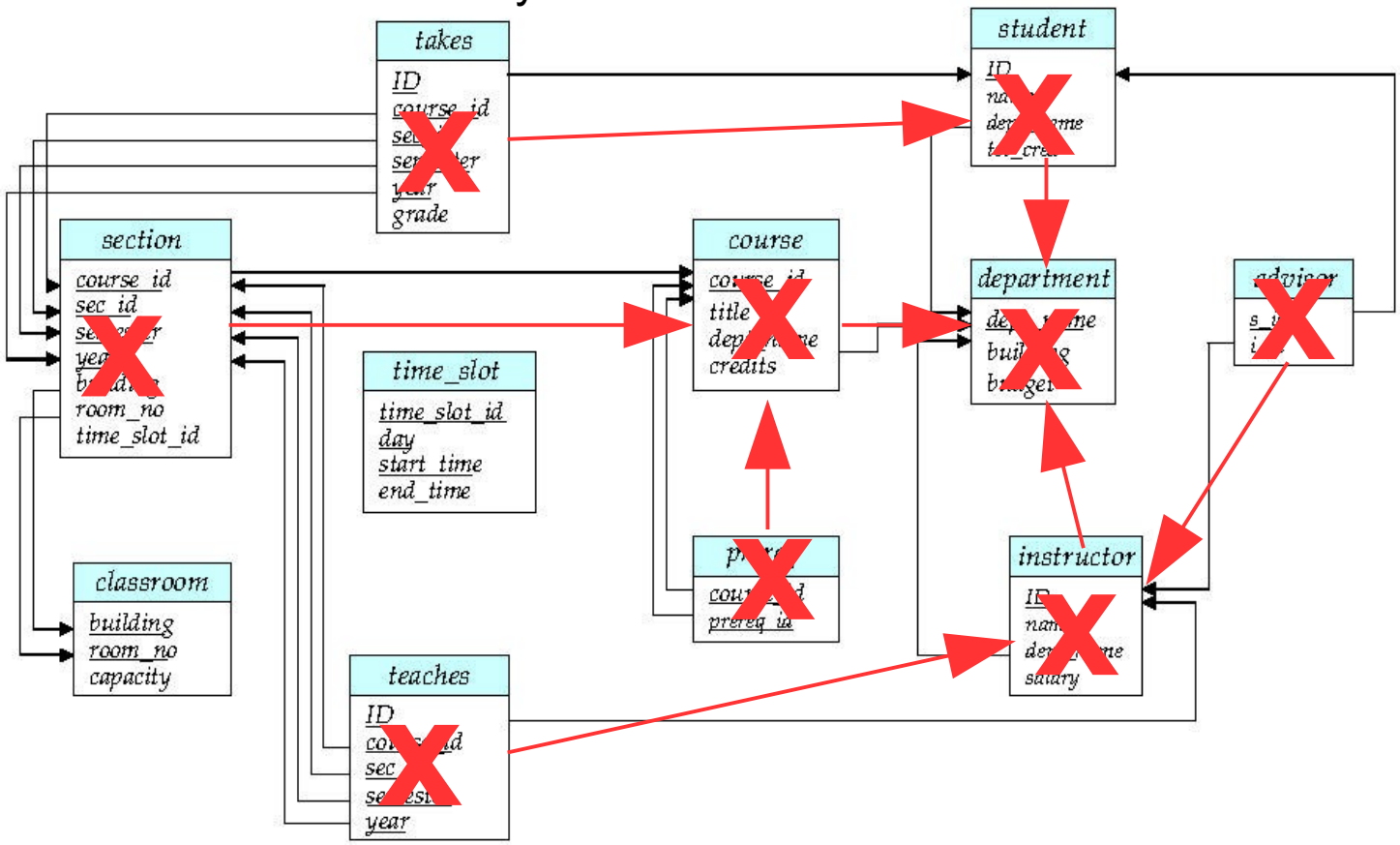
  . . .

)



Cascading Actions in Referential Integrity

- Cascading deletions may run over several tables
  - ...so we should be very careful!



Cascading Actions in Referential Integrity

- **set null** for updates
  - If a foreign key is changed (e.g., renaming a department)
  - ...then set null for all referring relations
- **set null** for deletions
  - If a foreign key is deleted (e.g., deleting a department)
  - ...then set null in referring relations
- **create table** *instructor* (

  …

  dept_name **varchar**(20),

  foreign key (*dept_name*) **references** *department*

  on delete set null,

  on update set null,

  . . .

)

Date and Time Data Types in SQL

- We have already encountered characters and numbers
- **date**: Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-7-27'
- **time**: Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30' **time** '09:00:30.75'
- **timestamp**: date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval**: period of time
  - Example: interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

Arithmetics with Dates

- Dates can be compared
  - i.e., < or >
  - e.g., select employees who started before January 1st, 2017
  - Special function: NOW() (in MariaDB; name may differ for other DBMS)
- Dates can be added to / substracted from intervals and other dates
  - e.g., select students who have been enrolled for more than five years

- Implementation not standardized
  - Details differ from DBMS to DBMS!

User Defined Types

- **create type** construct in SQL creates user-defined type

create type *Dollars* **as numeric (12,2) final**

- **create table** *department*
  (*dept_name* **varchar** (20),
  building **varchar** (15),
  budget Dollars);

required due to SQL standard; not really meaningful

User-defined Domains

- **create domain** construct creates user-defined domain types

  create domain *person_name* **char**(20) **not null**

- Types and domains are similar
  - Domains can have constraints, such as **not null**, specified on them

    create domain *degree_level* **varchar**(10)
    constraint *degree_level_test*
    check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Domain Constraints vs. Table Constraints

- Some checks may reoccur over different relations
  - e.g., degrees for students or instructors
  - e.g., salutations
  - e.g., valid ranges for ZIP codes

- Binding them to a *domain* is preferred
  - Central definition
  - Consistent usage

Large Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object -- object is a large collection of character data
- When a query returns a large object, a pointer is returned rather than the large object itself

Authorization

- Rights for accessing a database may differ
  - Only administrators may change the schema

- Rights for accessing a database can be very fine grained
  - Not everybody may see a persons' salary
  - Not everybody may alter a person's salary
  - Nobody may alter their own salary
  - Special restrictions may apply for entering salaries over a certain upper bound
  - ...

Authorization

- Forms of authorization on parts of the database:
  - **Read** - allows reading, but not modification of data
  - **Insert** - allows insertion of new data, but not modification of existing data
  - **Update** - allows modification, but not deletion of data
  - **Delete** - allows deletion of data

- Forms of authorization to modify the database schema
  - **Index** - allows creation and deletion of indices
  - **Resources** - allows creation of new relations
  - **Alteration** - allows addition or deletion of attributes in a relation
  - **Drop, Truncate** - allows deletion of relations

Authorization Specification in SQL

- The **grant** statement is used to confer authorization

  grant <privilege list>

  on <relation name or view name> **to** <user list>

- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator)

Privilege Definition in SQL

- **select**: allows read access to relation,or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 **select** authorization on the *instructor* relation:

    grant select on *instructor* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Revoking Privileges

- The **revoke** statement is used to revoke authorization.

  revoke <privilege list>

  on <relation name or view name> **from** <user list>

- Example:

  revoke select on *branch* **from** U_1, U_2, U_3

- <privilege-list> may be **all** to revoke all privileges the revokee may hold
- If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation
- All privileges that depend on the privilege being revoked are also revoked

Roles

- Databases may have many users
  - e.g., all students and employees of a university

- Managing privileges for all those individually can be difficult
  - User groups (also called: roles) are more handy
  - Example roles
    - Student
    - Instructor
    - Secretary
    - Dean
    - ...

Roles

- Creating roles and assigning them to individual users
  - **create role** instructor;
  - **grant** *instructor* **to Amit;**
- Granting privileges to roles
  - **grant select on** *takes* **to** *instructor*;

- Roles can form hierarchies
  - i.e., a role inherits from other roles

    create role *teaching_assistant*

    grant *teaching_assistant* **to** *instructor*;
  - *Instructor* inherits all privileges of *teaching_assistant*

Roles on Views

- Example: Geology department members can administrate their own staff, but not others

  create view *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');

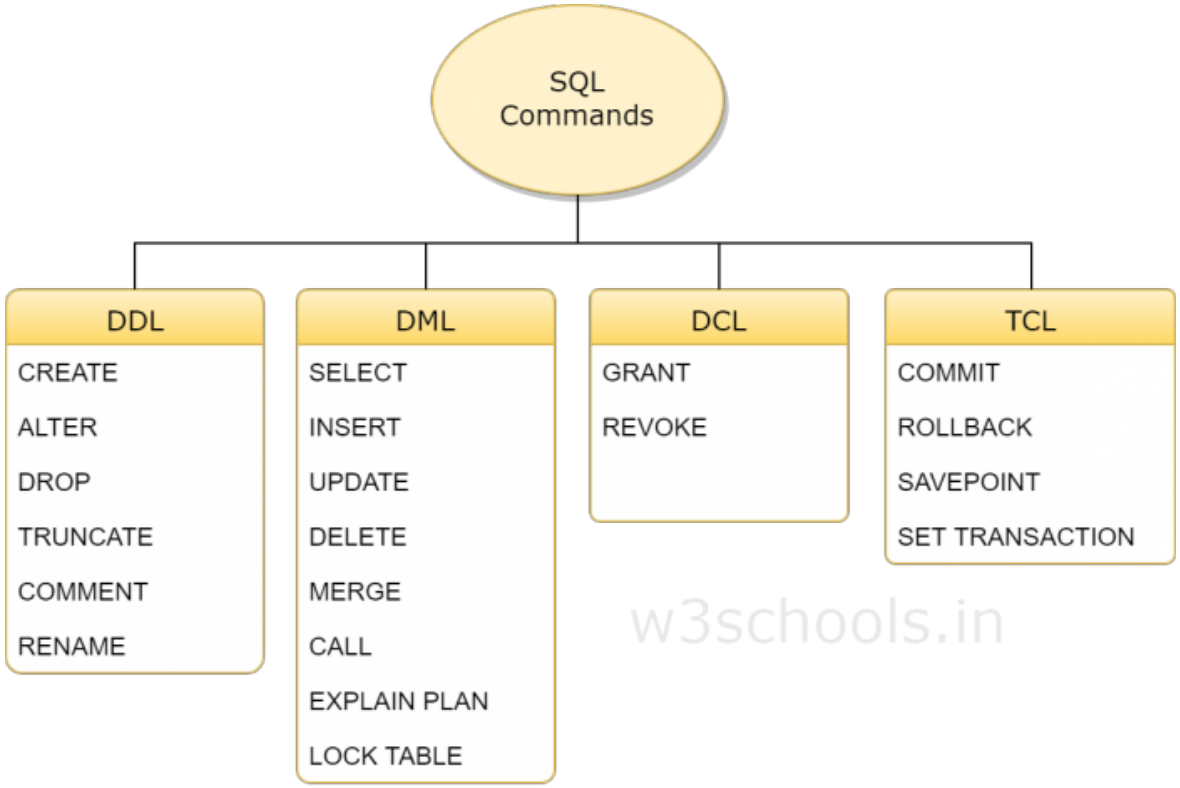
  grant select on *geo_instructor* **to** *geo_staff*

- Suppose that a *geo_staff* member issues

  select *
  from *geo_instructor*;

- What if
  - *geo_staff* does not have permissions on *instructor?*
  - creator of view did not have some permissions on *instructor?*

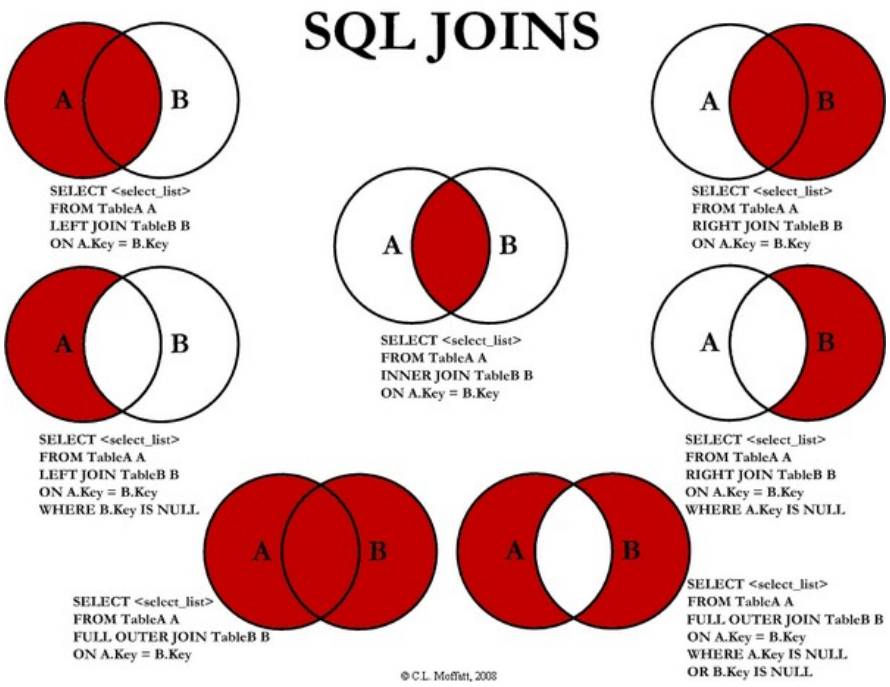
Wrap-up



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

Wrap-up

- Today, we have seen
  - More sophisticated means to read date from multiple tables
  - a.k.a. join operators



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

Wrap-up

- Today, we have seen
  - How to manipulate data in databases
  - i.e., **insert**, **update**, and **delete** statements

- Views
  - are used to provide different subsets and/or aggregations of data
  - updateable views
  - materialized views

Wrap-up

- Integrity constraints
  - unique and not null constraints
  - cascading updates and deletions

- Access rights
  - can be fine grained
  - can be bound to user groups and roles
  - roles may inherit from each other

Questions?