Database Technology
SQL Part 2

Heiko Paulheim
Looking Back

- We have seen
  - Reading data from tables
Outline

• Last week
  – Overview of The SQL Query Language
  – Basic Query Structure
  – Set Operations
  – Join Operators
  – Null Values
  – Aggregate Functions
  – Nested Subqueries

• Today
  – Data Definition
  – Data Types in SQL
  – Modifications of the database
  – Views
  – Integrity Constraints
  – Roles & Rights
SQL Data Definition Language (DDL)

- Allows the specification of information about relations, including
  - The schema for each relation
  - The domain of values associated with each attribute
  - Integrity constraints
- And as we will see later, also other information such as
  - The set of indices to be maintained for each relation
  - Security and authorization information for each relation
  - The physical storage structure of each relation on disk
Recap: Domain of an Attribute

- The set of allowed values for an attribute
  - Programmers: think *datatype*

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
</tr>
<tr>
<td>58583</td>
<td>Califieri</td>
<td>History</td>
<td>62000</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>Physics</td>
<td>87000</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>Finance</td>
<td>80000</td>
</tr>
</tbody>
</table>
Simple Domains in SQL

- **char(n).** Fixed length character string, with user-specified length \( n \).
- **varchar(n).** Variable length character strings, with user-specified maximum length \( n \).
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of \( p \) digits, with \( d \) digits to the right of decimal point. (ex., numeric\((3,1)\), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least \( n \) digits.
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 often not standardized
  – Details differ from DBMS to DBMS!
User Defined Types

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

  \[\text{create type } Dollars \text{ as numeric (12,2) final}\]

• **create table** *department*
  
  \[(\text{dept}_\text{name} \text{ varchar (20)}, \text{building} \text{ varchar (15), budget Dollars});\]

required due to SQL standard; not really meaningful
User-defined Domains

• `create domain` construct creates user-defined domain types

  ```sql
  create domain person_name char(20) not null
  ```

• Types and domains are similar
  – Domains can have constraints, such as `not null`, specified on them

  ```sql
  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
Creating Relations

• An SQL relation is defined using the **create table** command:

```
create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,
                (integrity-constraint_1),
                ...,           
                (integrity-constraint_k))
```

• *r* is the name of the relation
• each *A_i* is an attribute name in the schema of relation *r*
• *D_i* is the datatype/domain of values in the domain of attribute *A_i*

• Example:

```
create table instructor ( 
    ID char(5),
    name varchar(20),
    dept_name varchar(20),
    salary numeric(8,2))
```
Recap: Keys

- Primary keys identify a unique tuple of each possible relation $r(R)$
  - Typical examples: IDs, Social Security Number, car license plate
- Primary keys can consist of multiple attributes
  - e.g.: course ID plus semester (CS 460, FSS 2019)
  - Must be minimal – (ID, semester, instructor) would work as well
- Foreign keys refer to other tables
  - i.e., they appear in other tables as primary keys
Defining Keys

• primary key \((A_1, \ldots, A_n)\)
• foreign key \((A_m, \ldots, A_n)\) references \(r\)

Example:

```
create table instructor (  
  ID char(5),  
  name varchar(20),  
  dept_name varchar(20),  
  salary numeric(8,2),  
  primary key (ID),  
  foreign key (dept_name)  
    references department(dept_name));
```
Removing and Altering Relations

- **Removing relations**
  - drop table $r$

- **Altering**
  - alter table $r$ add $A$ $D$
    - where $A$ is the name of the attribute to be added to relation $r$, and $D$ is the domain of $A$
    - all existing tuples in the relation are assigned *null* as the value for the new attribute
  - alter table $r$ drop $A$
    - where $A$ is the name of an attribute of relation $r$
    - not supported by many databases
Back to DML...

- We have seen
  - Basic DDL: how do we define tables?
  - SELECT: how do we read from tables?
Insertion into a Relation

• Add a new tuple to course

\[
\text{insert into course values ('CS-437', 'Database Systems', 'Comp. Sci.', 4)};\
\]

• or equivalently

\[
\text{insert into course (course_id, title, dept_name, credits) values ('CS-437', 'Database Systems', 'Comp. Sci.', 4)};\
\]

• Add a new tuple to student with \(\text{tot_creds}\) set to null

\[
\text{insert into student values ('3003', 'Green', 'Finance', null)};\
\]
Insertion of Data from Other Tables

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

  ```sql
  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

  ```sql
  insert into table1 select * from table1
  ```

  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*
Caveats with NOT NULL Constraints

• Rationale:
  – Each course takes place at a specific room and time slot
  – We’ll create a **not null** constraint on those fields
  – *Note*: no online courses here

• Use case:
  – First: enter all courses in the system
  – Second: run clever time and room allocation algorithm
    • Which will then fill all the buildings and time slots
Caveats with NOT NULL Constraints (ctd.)

• Example: every employee needs a substitute
  – create table employee ( 
    ID varchar(5),
    name varchar(20) not null,
    substitute varchar(5) not null,
    primary key (ID),
    foreign key (substitute) references employee(ID));

• What do you think?
Updating Data

• Example: update the salary of a single person
  
  ```sql
  update employee
  set salary = 80000
  where person_id = 43743
  ```

• Example: update all salaries by 5%
  
  ```sql
  update employee
  set salary = salary * 1.05
  ```

• Example: moving all people from a department to a new building
  
  ```sql
  update employee
  set building = 'Taylor'
  where dept_name = 'Biology'
  ```

• Anatomy of an update query
  
  – set defines which updates to carry out
  
  – where defines which records to update (omitted = all records)
Updating Data

- Cut salaries above 100,000 by 5%, below 100,000 by 3%
- Write two `update` statements:
  
  ```
  update instructor 
  set salary = salary * 0.95 
  where salary > 100000;
  
  update instructor 
  set salary = salary * 0.97 
  where salary <= 100000;
  ```

- Should rather be done using the `case` statement (next slide)

thought experiment: Tom’s salary is 102,000
Conditional Updates with case Statement

• Cut salaries above 100,000 by 5%, below 100,000 by 3%

  update instructor
    set salary = case
      when salary > 100000 then salary * 0.95
      else salary * 0.97
    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 or passed any course

- Instead of `sum(credits)`, use:
  ```
  case
      when sum(credits) is not null then sum(credits)
      else 0
  end
  ```
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
  
  ```sql
  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
  
  ```sql
  delete from instructor
  where dept_name in (select dept_name
                        from department
                        where building = 'Watson');
  ```

  where clause may contain everything also usable for `select`
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?
Deleting from a Relation

• Delete all instructors whose salary is less than the average salary of instructors
  
  ```sql
  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!)
Back to DML...

- We have seen
  - Basic DDL: how do we define tables?
  - SELECT: how do we read from tables?
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.,
  \( \text{select} \ ID, \ name, \ dept\_name \)
  \( \text{from} \ instructor \)
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:
  
  ```sql
  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

```sql
create view faculty as
select ID, name, dept_name
from instructor
```

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

```sql
select name
from faculty
where dept_name = 'Biology';
```

• Create a view of department salary totals

```sql
create view departments_total_salary(dept_name, total_salary) as
select dept_name, sum(salary)
from instructor
group by dept_name;
```
Updating Views

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

  ```sql
  create view faculty as
  select ID, name, dept_name
  from instructor
  ```

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

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

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

  ```sql
  ('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

```sql
create view biology_faculty as
select ID, name
from faculty
where dept_name = 'Biology';
```

• and

```sql
insert into biology_faculty
values (43278, 'Smith');
```

• Would this lead to

```sql
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

  ```sql
  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

  ```sql
  update departments_total_salary
  set total_salary = total_salary * 1.05
  where dept_name = "Comp. Sci.";
  ```
Updating Views

- **create view** `instructor_info` **as**
  ```sql
  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 — default action
  – Delete all instructors as well
  – Set the department of those instructors to null
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,  
  ...  
  )
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
  
  \[\text{grant } <\text{privilege list}> \]
  
  \[\text{on } <\text{relation name or view name}> \text{ to } <\text{user list}>\]

- **<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:
    ```sql
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 U₁, U₂, U₃
  ```

• `<privilege-list>` may be `all` to revoke all privileges the revokee may hold

• If `<user 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
Revoking Privileges

• Scenario 1:
  – **grant** select on instructor to john, mary
  – **revoke** select on instructor from john
    → Mary retains right

• Scenario 2:
  – **grant** select on instructor to public
  – **grant all on** instructor to john
  – **revoke all on** instructor from public
    → John retains right, since he has been granted the right explicitly
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: Example

- Employee
  - grant select on department
  - grant employee to teaching_assistant
  - grant employee to admin_staff

- Teaching Assistant
  - grant teaching_assistant to instructor

- Admin Staff
  - grant all on takes

- Instructor
  - grant all on section
  - grant read on takes
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
  – 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?