L06: SQL 3

DSAN 6300/PPOL 6810: Relational Databases and SQL Programming

Irina Vayndiner

October 2 and 5, 2023

GEORGETOWN UNIVERSITY

Logistics

- Last Class was recorded
- Important Dates
  - HW2 - due **Mon, 10/16** (grace period till Wed 10/17)
    - Answers will be available as soon as **all** students submit!
  - Quiz 2 – due 10/10
  - Mid-term is on **Mon,10/23 and Thu 10/19 (no other dates!)**
    - Closed books
    - Format: similar to Lab06, HW2, will explain shortly
  - Project will be assigned on the week of 11/7, due 12/6 (no late submissions!)
  - **Test: Tue 12/5 (all sections)**
- Starting today: ALL delayed assignments must be emailed ONLY to
    - Ram (DSAN)
    - Peijin (PPOL)
    - I need to be CC'ed

No other Tas can accept your work for grading

Agenda for today's class

- Today:
  - Lecture: SQL 3
  - Lab: SQL exercises
    - No need to disable the safe mode as slide says

- Format of submitting SQL results
  - 1 file with sql + number of rows returned (.sql)
    - Problem formulation is commented out
    - Solution
    - Number of rows commented out
  - N files corresponding to N questions
    - E.g. Output_question2_last_name_first_name
      - No file needed if 0 rows returned

Outline: Proceeding with SQL

Last Class

- Nested Subqueries in WHERE, FROM and SELECT clauses
- Modification of the Database (insert, update, delete)
- Join Expressions including outer joins
- Views

- **This Class**
  - Transactions
  - Use of Integrity Constraints in SQL
  - SQL Data Types and Schemas
  - Index Definition in SQL
  - Authorization
  - Accessing SQL From a Programming Language
  - Functions and Procedures
  - Triggers

Transactions

- Consider a banking application where we need to transfer money from one bank account to another in the same bank
- To do so, we need to update two account balances, subtracting the amount transferred from one, and adding it to the other.
- If the system crashes after subtracting the amount from the first account but before adding it to the second account, the bank balances will be inconsistent.
- A similar problem occurs if the second account is credited before subtracting the amount from the first account and the system crashes just after crediting the amount.
- To handle this scenarios a notion of a **transaction** was introduced

Transactions (continued)

Transaction Begins

```
UPDATE savings_accounts
     SET balance = balance - 500
     WHERE account = 3209;
```

Decrement Savings Account

```
UPDATE checking_accounts
     SET balance = balance + 500
     WHERE account = 3208;
```

Increment Checking Account

```
INSERT INTO journal VALUES
     (journal_seq.NEXTVAL, '1B'
     3209, 3208, 500);
```

Record in Transaction Journal

```
COMMIT WORK;
```

End Transaction

Transaction Ends

ref: docs.oracle.com

Transactions (continued)

- A **transaction** consists of a sequence of queries and/or update (UPDATE, DELETE, or INSERT) statements
- Transactions are **atomic**
  - Either fully executed or rolled back as if it never occurred
  - It is a “unit” of work
- The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed.
- The transaction must end with one of the following statements:
  - **Commit:** The updates performed by the transaction become permanent in the database.
  - **Rollback:** All the updates performed by the SQL statements in the transaction are undone.
- If an error occurs, DBMS rolls back a transaction
- In many SQL implementations, each SQL statement is taken to be a transaction and it gets committed as soon as it is executed.
  - Such automatic commit of individual SQL statements must be turned off if a transaction consisting of multiple SQL statements needs to be executed.

Transactions in MySQL example

```
START TRANSACTION [transaction_characteristic]

transaction_characteristic: {

            | READ WRITE
            | READ ONLY
}
BEGIN [WORK]
If errors: ROLLBACK [WORK]
COMMIT [WORK]
```

ref: docs.oracle.com

Integrity Constraints

- **Integrity constraints** guard against accidental damage to the database, by ensuring that changes to the database do not result in a loss of data consistency.
- Examples of integrity constraints are:
  - An instructor name cannot be null.
  - No two instructors can have the same instructor ID.
  - Every department name in the course relation must have a matching department name in the department relation.
  - The budget of a department must be greater than $0.00.
- Integrity constraints are usually identified as part of the database schema design process, and declared as part of the create table command used to create relations.
- Integrity constraints can also be <u>added to an existing relation</u> by using the command

  alter table *table-name* **add** *constraint*

- It is possible to assign a name to integrity constraints.
  - Such names are useful if we want to drop a constraint that was defined previously.

Constraints on a Single Relation

- The **create table** command often includes integrity-constraint statements
- There are a number of constraints that can be included in the create table command. The allowed integrity constraints include
  - **not null**
  - **primary key**
  - **unique**
  - **check** (P), where P is a predicate

not null Constraints

- As you remember, the null value is a member of all domains, and as a result it is a legal value for every attribute in SQL by default.
  - For certain attributes, however, null values may be inappropriate.
- Consider a tuple in the student relation where name is null; it does not contain useful information.
- In cases such as this, we wish to forbid null values, and we can do so by restricting the domain of the attributes name to exclude null values, by declaring it as follows:

  student_name **varchar**(20) **not null**
- The **not null** constraint prohibits the insertion of a null value for the attribute, and is an example of a ***domain constraint.***
- Any database modification that would cause a null to be inserted in an attribute declared to be not null <u>generates an error</u>

Unique Constraints

- The **unique** specification says that attributes $A_1, A_2, \ldots, A_m$ form a candidate key; that is, no two tuples in the relation can be equal on all the listed attributes.
- **unique** ($A_1, A_2, \ldots, A_m$)
  - Candidate keys are permitted to be null (in contrast to primary keys).
- Example

```
create table Persons (
    ID int not null,
    LastName varchar(255) not null,
    FirstName varchar(255),
    Age int,
    unique (ID)
);
```

- Note**:** Unique **operator** is different from this unique **constraint**
  - Reminder: Returns TRUE if a result set of a subquery does not have any duplicates and is used mainly in where clause

Check clause

- The **check** (P) clause in the relation declaration specifies a predicate P that must be satisfied by every tuple in a relation.
- Example: ensure that semester is one of: Fall, Winter, Spring or Summer

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),
check (*semester* **in** ('Fall', 'Winter', 'Spring', 'Summer')))

- Can *semester* have a null value in above statement?
  - Yes! A check clause is satisfied if it is not false, so clauses that evaluate to **unknown** are not violations.
    - Unless semester declared **not null**
- A check clause may appear on its own, as shown above, or as part of the declaration of an attribute.

Referential Integrity (reminder)

- Often we want to ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If "Biology" is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for "Biology".
- Reminder:
  - 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.

Referential Integrity (continued)

- Foreign *keys can be* specified as part of the SQL
- **create table** statement

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

- By default, a foreign key references the primary-key attributes of the referenced table.
- SQL allows a list of attributes of the referenced relation to be specified explicitly.

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

- The specified list of attributes that are referenced must, however, be declared either a **primary key** constraint, or a **unique** constraint.

Cascading Actions in Referential Integrity

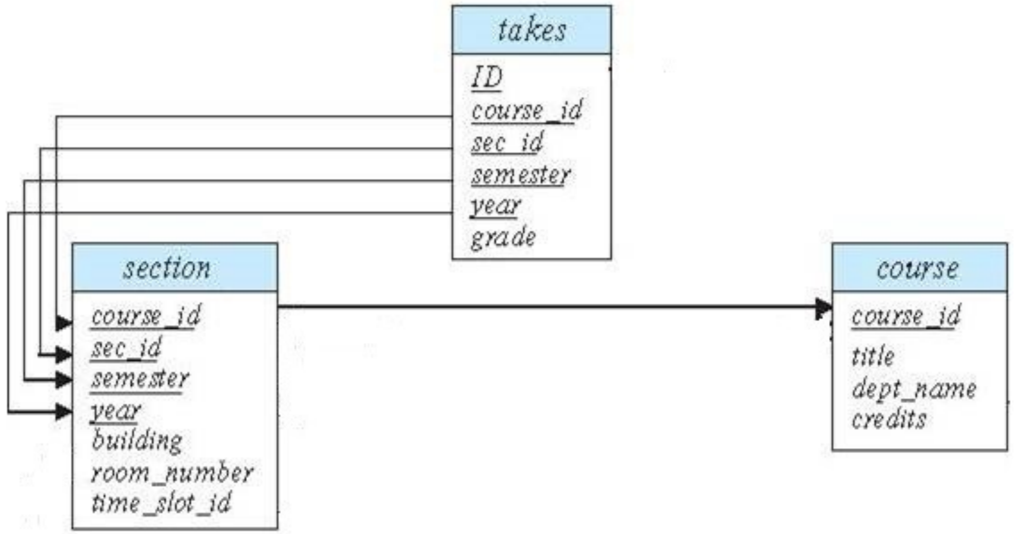
- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- An alternative, in case of delete or update is to cascade

create table *course* (
    (…
    dept_name **varchar**(20),
    foreign key (*dept_name*) **references** *department*
        on delete cascade
        on update cascade,
    . . .)

- In case department table is deleted, courses in that department will be deleted
- Instead of cascade we can use, for example:
  - **On delete set null**
  - **On update set default**

Integrity Constraints During Transactions

- Question: When does the DBMS verify whether an integrity constraint is violated?
- Approach 1: After a single database modification (insert, update or delete statement) - **immediate mode**
  - In this case the order of operations is important



  - Insert order: course->section->takes
  - Delete order (opposite direction): takes->section->course
- Side note: Can we join takes with course, w/o including section? Yes!
  - Course_id is FK from takes to section, and from section to course => FK from takes to course

Integrity Constraints During Transactions

- Question: When does the DBMS verify whether an integrity constraint is violated?
- Approach 2: At the end of a transaction - **deferred mode**

  begin transaction

  Db modifications

  end transaction

  - In this case modifications can be done in any order, as long as in the end all constraints are satisfied
- Note:
  - Not all DBMSs have that functionality
  - Sometimes database do not allow (or RESTRICT) deletes or updates for the parent table

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- The following constraints, can be expressed using assertions:
  - For each tuple in the *student* relation, the value of the attribute *tot_cred* must equal the sum of credits of courses that the student has completed successfully.
  - An instructor cannot teach in two different classrooms in a semester in the same time slot
- An assertion in SQL takes the form:

  create assertion <assertion-name> **check** (<predicate>);
- Currently NOT supported by many DBMS
  - Supported by Oracle
- Example of assertion of only one CEO of a Company
  - create assertion AT_MOST_ONE_CEO as CHECK
    ((select count(*) from EMPLOYEE where e.JOB = CEO') <= 1)

Built-in Data Types in SQL (as we saw earlier)

The SQL standard supports a variety of built-in types, including

- **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)
- **float(n).** Floating point number, with user-specified precision of at least *n* digits.
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.

Large-Object Types

- Large objects (photos, videos, etc.) can be 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

Built-in Data Types in SQL (continued)

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** ‘2020-7-27'
- **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** ‘2022-7-27 09:00:30.75'
- Many useful operations to deal with <u>dates and time</u>
  - Subtracting a date/time/timestamp value from another gives an interval value
    - **extract** (*field* from d), where *field* can be year, month, day, hour, minute etc.
  - Date/time functions, e.g. **current_date**
- Syntax **varies** greatly from one RDBMS to another

Built-in Data Types: MySQL

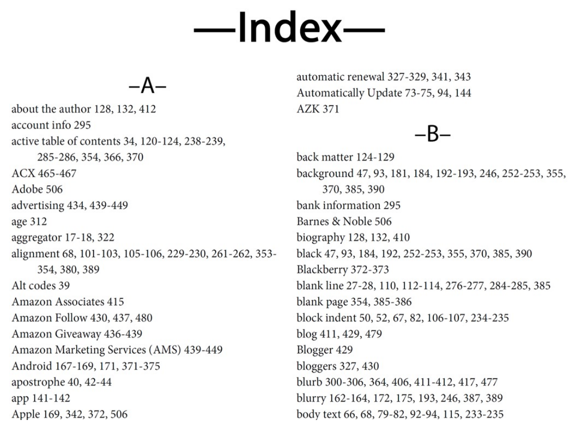
MySQL DATA TYPES

DATE TYPE	SPEC	DATA TYPE	SPEC
CHAR	String (0 - 255)	INT	Integer (-2147483648 to 214748-3647)
VARCHAR	String (0 - 255)	BIGINT	Integer (-9223372036854775808 to 9223372036854775807)
TINYTEXT	String (0 - 255)	FLOAT	Decimal (precise to 23 digits)
TEXT	String (0 - 65535)	DOUBLE	Decimal (24 to 53 digits)
BLOB	String (0 - 65535)	DECIMAL	"DOUBLE" stored as string
MEDIUMTEXT	String (0 - 16777215)	DATE	YYYY-MM-DD
MEDIUMBLOB	String (0 - 16777215)	DATETIME	YYYY-MM-DD HH:MM:SS
LONGTEXT	String (0 - 4294967295)	TIMESTAMP	YYYYMMDDHHMMSS
LONGBLOB	String (0 - 4294967295)	TIME	HH:MM:SS
TINYINT	Integer (-128 to 127)	ENUM	One of preset options
SMALLINT	Integer (-32768 to 32767)	SET	Selection of preset options
MEDIUMINT	Integer (-8388608 to 8388607)	BOOLEAN	TINYINT(1)

Index

Index creation

- Many queries reference only a small proportion of the records in a table.
- Ex. "Find all instructors with the last name SMITH"
- It is inefficient for the system to read every record to find a record with particular value
- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute <u>efficiently</u>, without scanning through all the tuples of the relation.



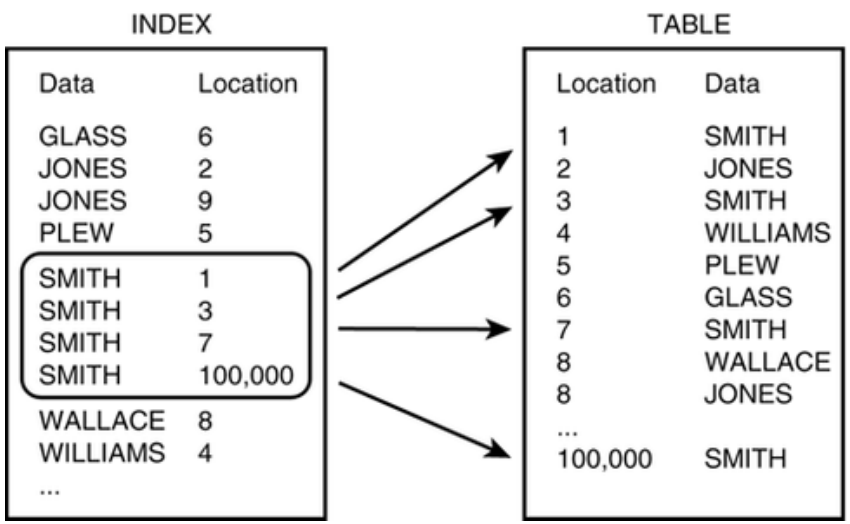
—Index—

–A–

about the author 128, 132, 412
account info 295
active table of contents 34, 120-124, 238-239, 285-286, 354, 366, 370
ACX 465-467
Adobe 506
advertising 434, 439-449
age 312
aggregator 17-18, 322
alignment 68, 101-103, 105-106, 229-230, 261-262, 353-354, 380, 389
Alt codes 39
Amazon Associates 415
Amazon Follow 430, 437, 480
Amazon Giveaway 436-439
Amazon Marketing Services (AMS) 439-449
Android 167-169, 171, 371-375
apostrophe 40, 42-44
app 141-142
Apple 169, 342, 372, 506
automatic renewal 327-329, 341, 343
Automatically Update 73-75, 94, 144
AZK 371

–B–

back matter 124-129
background 47, 93, 181, 184, 192-193, 246, 252-253, 355, 370, 385, 390
bank information 295
Barnes & Noble 506
biography 128, 132, 410
black 47, 93, 184, 192, 252-253, 355, 370, 385, 390
Blackberry 372-373
blank line 27-28, 110, 112-114, 276-277, 284-285, 385
blank page 354, 385-386
block indent 50, 52, 67, 82, 106-107, 234-235
blog 411, 429, 479
Blogger 429
bloggers 327, 430
blurb 300-306, 364, 406, 411-412, 417, 477
blurry 162-164, 172, 175, 193, 246, 387, 389
body text 66, 68, 79-82, 92-94, 115, 233-235



Index Creation Example

- Create an index with the **create index** command

  create index <name> **on** <relation-name> (attribute_list);

- Example for *Relation: student*(*ID* **varchar** (5), *name* **varchar** (20),*dept_name* **varchar** (20))

  create index *studentID_index* **on** *student*(*ID*)

- The query:

  select *
  from *student*
  where *ID* = '12345'

  can be executed by using the index to find the required record, without looking at all records of *student*

- Typically, you create indexes for a table at the *time of table creation*
  - Can add later for performance tuning (more on that later)
- When you create a table with a primary key or unique key, MySQL automatically creates a special index named **Primary.**

Accessing SQL from a Programming Language - Motivation

A database programmer must have access to a general-purpose programming language for example, for these reasons:

- Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language.
- Non-declarative actions -- such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface (GUI) -- cannot be done from within SQL.
- Often, data from a database is an input to a business or analytical process that is implemented using a programming language like Python or Java
  - Training a predictive model
  - Creating a business report

Methods to Access SQL from a Programming Language

There are two approaches to accessing SQL from a general-purpose programming language

- **Dynamic SQL**. A general-purpose program can connect to and communicate with a database server using a collection of functions (SQL API)
  - Construct an SQL query as a character string at runtime
  - Submit the query
  - Retrieve the result into program variables a tuple at a time
  - SQL statements are constructed *at runtime*; for example, the application may allow users to enter their own queries
- **Embedded SQL**. Provides means by which a program can interact with a database server.
  - The SQL statements are translated *at compile time* into function calls.
  - At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities.
  - Not that popular these days
  - SQL statements in an application do not change at runtime and, therefore, need to be hard-coded into the application.

JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports a metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
- Model for communicating with the database:
  - Open a connection
  - Create a "statement" object
  - Send queries (using the statement object) for execution, and fetch the results
  - Provide an exception mechanism to handle errors

- Database vendors provide client libraries (JDBC drivers) for their DBMS
- JDBC Basics Tutorial
  - https://docs.oracle.com/javase/tutorial/jdbc/index.html

ODBC

- Open DataBase Connectivity (ODBC)
  - Conceptually similar to JDBC
- ODBC is a standard for application program (e.g. C++, Python) to communicate with a database server.
- It is an Application program interface (API) that allows to
  - Open a connection with a database
  - Send queries and updates
  - Get back the results.
- Many applications such as GUI, spreadsheets, etc. can use ODBC

Functions and (Stored) Procedures

Functions and Procedures

- Functions and procedures allow "business logic" to be stored in the database and executed from SQL statements.
- These can be defined either by the procedural component of SQL or by an external programming language such as Python, Java, or C++.
- The syntax presented further here is defined by the SQL standard.
  - Many databases implement nonstandard versions of this syntax.

Declaring SQL Functions examples

- Define a function that, given the name of a department, returns (count of) the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
        returns integer
   begin
        declare d_count integer;
            select count (* ) into d_count
            from instructor
            where instructor.dept_name = dept_name
        return d_count;
   end
```

- The function *dept*_count can now be used to find the department names and budget of all departments, say, with more that 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

SQL (Stored) Procedures

- The *dept_count* function could instead be written as a **procedure**
- **create procedure** *dept_count_proc*

```
                    (in dept_name varchar(20), out d_count integer)
 begin
      select count(*) into d_count
      from instructor
      where instructor.dept_name = dept_count_proc.dept_name
  end
```

- The keywords **in** and **out** are parameters that are expected to have values assigned to them and parameters whose values are set in the procedure in order to return results.
- Procedures can be invoked either from an SQL procedure or from programming language, using the **call** statement.

```
declare d_count integer;
call dept_count_proc( 'Physics', d_count);
```

SQL Stored Procedures (continued)

- A **procedure** (often called a stored procedure) is a subroutine like a subprogram in a regular computing language, stored in database.
  - A procedure has a name, a parameter list, and SQL statement(s).
- Procedures and functions can be invoked also from dynamic SQL
  - A **stored procedure** will accept input parameters so that a single **procedure** can be used over the network by several clients using different input data
- **Stored procedures** differ from ordinary SQL statements and from batches of SQL statements in that they are precompiled.
  - Subsequently, the procedure is executed according to the **stored** plan.
  - Since most of the query processing work has already been performed, stored procedures usually execute much faster.

Table Functions

- The SQL standard supports functions that can return tables as results; such functions are called **table functions**
- Example: Return all instructors in a given department

  Define: **create function** *instructor_of* (*dept_name* **char**(20))

  returns table (*ID* **varchar**(5), *name* **varchar**(20))

  return table
  (**select** *ID, name*
  from *instructor*
  where *instructor.dept_name = instructor_of.dept_name*)

- Usage

  select *
  from table (*instructor_of* ('Music'))

- Does not work in MySQL; works in PostgreSQL

Language Constructs for Procedures & Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
  - Warning: most database systems implement their own variant of the standard syntax below.
- Compound statement: **begin** … **end**
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements

Language Constructs: while-do and repeat-until

- **while** search condition **do**

  sequence of statements

  end while

- **repeat**

  sequence of statements

  until boolean expression

  end repeat

Language Constructs: For loop

- **for** loop
  - Permits iteration over all results of a query
- Example: Find the *total* budget of all departments that have budget > $10,000

declare *n* **integer default** 0;
for *r* **as**
    select *budget* **from** *department*
    where *budget* > 10000
do
        set *n* = *n* + *r.budget*
end for

Language Constructs – if-then-else

- Conditional statements (**if-then-else**)

  if *boolean expression*
  then *statement or compound statement*
  elseif *boolean expression*
  then *statement or compound statement*
  else *statement or compound statement*
  end if

External Language Routines

- SQL allows us to define functions in a programming language such as Java, or C++, etc.
  - DBMS do not support procedural extensions in a **standard** way
  - Can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL, can be executed by these functions.
- Allows so called in-database processing (more on that later)
  - No need to download model input data from the database
  - Allows to use parallel processing capabilities of the DBMS engine
    - Ex.: calculating risk scores with a predictive model trained outside of the database and implemented as Java User-Defined Function (UDF)

Triggers

Triggers

- A **database trigger** is procedural code that is **automatically** executed in response to certain events on a particular table or view in a database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed: Inserts, updates, or deletes rows in the associated table.
    - For example, rows can be inserted by INSERT (or LOAD DATA) statements, and an insert trigger activates for each inserted row
  - Specify the actions to be taken when the trigger executes.
  - A trigger can be set to activate either before or after the trigger event
    - For example, a trigger can activate <u>before</u> each row that is inserted into a table, or <u>after</u> each row that is updated.
  - Triggers are created using the CREATE TRIGGER statement, and dropped using DROP TRIGGER.

When to Use Triggers

- To implement integrity constraints that cannot be specified using the constraint mechanism of SQL.
- Alerting humans or for starting certain tasks automatically when certain conditions are met.
- The triggers are often used for maintaining the integrity of the information on the database.
- Example: whenever a student takes a course we want to automatically update his/her total credits. In this example,
  - Whenever a tuple is inserted into the *takes* relation
    - update the tuple in the *student* relation by adding *credits* to *student.tot_cred*

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - For example, **after update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - **referencing old row as** name_old_row**:** for deletes and updates
  - **referencing new row as** name_new_row**:** for inserts and updates
- Triggers activated before an event can serve as extra constraints. For example, this trigger checks for the attempts to set *grade* value to *'blank'* and substitutes it for **null** just before the update:

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
     when (nrow.grade = ' ')
      begin atomic
        set nrow.grade = null;
 end;
```

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows
- Example:
  - A row-level trigger is activated for each row that is inserted, updated, or deleted.
    - If a table has 100 rows inserted, updated, or deleted, the trigger is automatically invoked 100 times for the 100 rows affected.
    - A statement-level trigger is executed once for each transaction regardless of how many rows are inserted, updated, or deleted.
- MySQL supports only row-level triggers.

When *Not* To Use Triggers

- Triggers were used historically for tasks such as:
  - Maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these tasks now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
- Stored procedures can be used instead of triggers in many cases to:
  - Define procedures to update fields
  - Carry out actions as part of the procedure that performs update instead of through a trigger

Risks when using Triggers

- Risk of unintended execution of triggers, for example, when
  - Loading data from a backup copy
  - Replicating updates at a remote site
  - To mitigate: Trigger execution can be disabled before such actions.
- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
    - Or, trigger itself has an error
  - Cascading execution
    - At times when SQL statement of a trigger can fire other triggers. This results in **cascading triggers**.
      - Oracle allows around 32 cascading triggers. Cascading triggers can cause result in abnormal behavior of the application.

Database Authorization

- **Authorization** is the process where the **database** manager gets information about the authenticated user (after a successful login).
- Part of that information is determining which **database** operations the user can perform and which data objects a user can access.
- We may assign a user several forms of authorizations on parts of the database. Usually done by a Database Administrator (DBA)
  - **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.
- Each of these types of authorizations is called a **privilege**.
  - DBA may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
- To create a role we use:

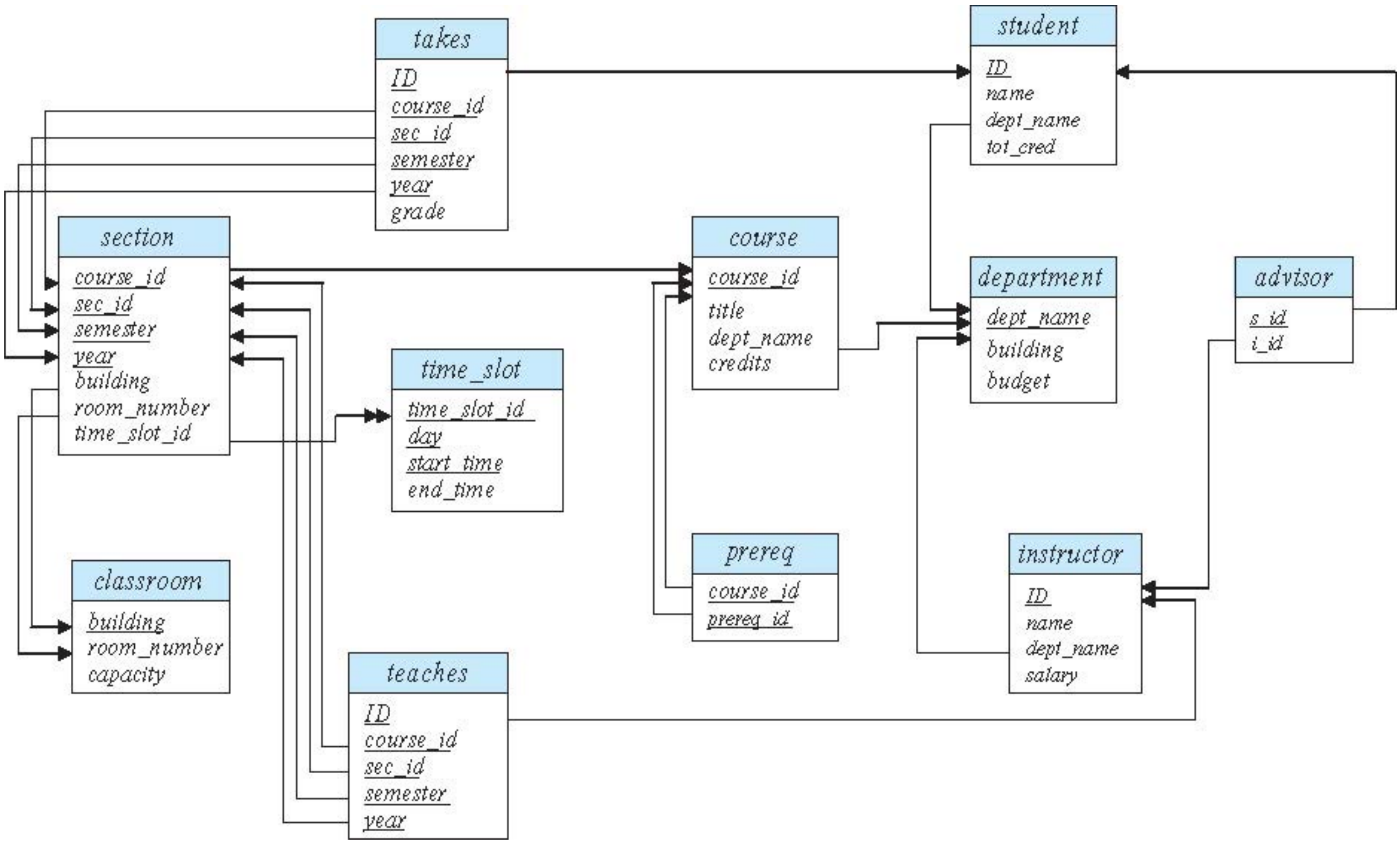
  create a role <name>
- Example:
  - **create role** instructor
- Once a role is created we can assign “users” to the role using:
  - **grant** <role> **to** <users>

Roles Example

- **create role** instructor;
- **grant** *instructor* **to** Amit**;**
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching_assistant*
  - **grant** *teaching_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

Lab 06: Submit ALL problems, verify deletes/inserts/updates

Schema Diagram for University Database



SQL – Problem 1

Find the highest salary of any instructor.

Solution steps (reminder)
What data attributes we need in the result set?
What data attributes are used in the condition?
In what tables do we have this information, what are the names of the attributes?
If there is more than one table, how we join the tables?
What other condition should be satisfied?

SQL – Answer to Problem 1

Find the highest salary of any instructor.

```
select max(salary) from instructor;
```

SQL – Problem 2

Find instructors earning the highest salary (there may be more than one with the same salary).

Solution steps:
What data attributes we need in the result set?
What data attributes are used in the condition?
In what tables do we have this information, what are the names of the attributes?
If there is more than one table, how we join the tables?
What other condition should be satisfied?

Hint: you may want to use a query of Problem 2 as a subquery in where clause here

Answer to Problem 2

Find instructors earning the highest salary (there may be more than one with the same salary).

select ID, name
from instructor
where salary =
    (**select** max(salary) **from** instructor)

SQL – Problem 3

Find the enrollment (=number of students) of each section of each course that was offered in Fall 2017.

Solution steps

What data attributes we need in the result set?

What data attributes are used in the condition?

In what tables do we have this information, what are the names of the attributes?

If there is more than one table, how we join the tables?

What other condition should be satisfied?

SQL – Answer to Problem 3

Find the enrollment of each section of each course that was offered in Fall 2017. Three options:

1) without group by with subquery

```
select course_id, sec_id,
    (select count(ID)
        from takes
        where takes.year = section.year and
        takes.semester = section.semester and
        takes.course_id = section.course_id and
        takes.sec_id = section.sec_id)
    as enrollment
from section
where semester = 'Fall' and year = 2017;
```

SQL – Answer to Problem 3 (continued)

2) With group by and join

```
select takes.course_id, takes.sec_id, count(ID) as enrollment
    from section, takes
    where takes.year = section.year and
        takes.semester = section.semester and
        takes.course_id = section.course_id and
        takes.sec_id = section.sec_id and
        takes.semester = 'Fall' and
        takes.year = 2017
    group by takes.course_id, takes.sec_id;
```

3) with group by w/o join (since it happened that takes has info about section!)

```
select course_id,  sec_id ,count(ID) as enrollment
from takes
where semester = 'Fall' and year = 2017
group by course_id, sec_id ;
```

SQL – Problem 4

Outer join problem

Display a list of all instructors, showing each instructor's ID and the number of sections taught.
Make sure to show the number of sections as 0 for instructors who have not taught any section.

Your query should use ***an outer join, and should not use subqueries.***

SQL – Answer to Problem 4

Display a list of all instructors, showing each instructor's ID and the number of sections taught.
Make sure to show the number of sections as 0 for instructors who have not taught any section.
Your query should use an outer join, and should not use subqueries.

select ID, **count(**sec_id**) as** number_of_sections
from instructor **natural left outer join** teaches
group by ID

SQL Problem 5

inserts, **deletes**, or **updates** in SQL

Add a Finance department student with the last name 'Green', and student ID = 3003

Answer to SQL Problem 5

inserts, **deletes**, or **updates** in SQL

Add a Finance department student with the last name Green, and student ID=3003

```
insert into student
    values ('3003', 'Green', 'Finance', null);

# to verify
select * from student where id=3003;
```

Disable Safe Mode

inserts, **deletes**, or **updates** in SQL

Disable Safe Updates

MySQLWorkbench=>Preferences=> SQL Editor=>Safe Updates (uncheck)

SQL Problem 6

inserts, **deletes**, or **updates** in SQL

Change the last name of the student ID=3003 from Green to Brown

Answer to SQL Problem 6

inserts, **deletes**, or **updates** in SQL

Change the last name of the student ID=3003 from Green to Brown

```
update student set name='Brown' where id=3003;

# to verify
select * from student where id=3003;
```

SQL Problem 7

inserts, **deletes**, or **updates** in SQL

Delete the student with ID=3003

Answer to SQL Problem 7

inserts, **deletes**, or **updates** in SQL

Delete the student with ID=3003

```
delete from  student where id=3003;

# to verify
select * from student where id=3003;
```

SQL – Problem 8

inserts, **deletes**, or **updates** in SQL

Increase the salary of each instructor in the Comp. Sci. department by 10%.

SQL – Answer to Problem 8

inserts, **deletes**, or **updates** in SQL

Increase the salary of each instructor in the Comp. Sci. department by 10%.

update *instructor* **set** salary = salary * 1.10
where dept name = ‘Comp. Si.’

SQL- Problem 9

Delete all courses that have never been offered (i.e. do not occur in the *section* relation).

SQL – Answer to Problem 9

Delete all courses that have never been offered (i.e. do not occur in the *section* relation).

delete from *course*
where *course_id* **not in** (**select** *course_id* **from** *section*)