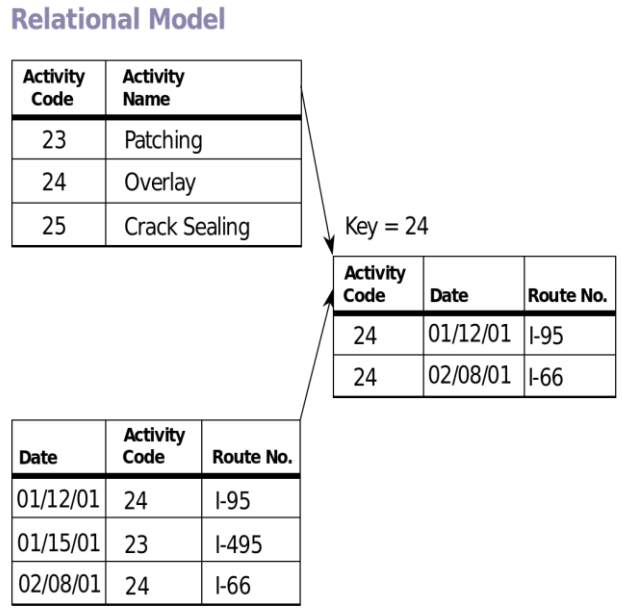
DB Management Systems: SQL - MySQL

Joel Klein – jdk514@gwmail.gwu.edu

Introduction to MySQL

MySQL

- MySQL is a RDBMS (Relational Database Management System)
  - Database means that it is designed for the storage of information
  - Relational means that it conforms to the relational model



Source: https://en.wikipedia.org/wiki/File:Relational_Model.svg

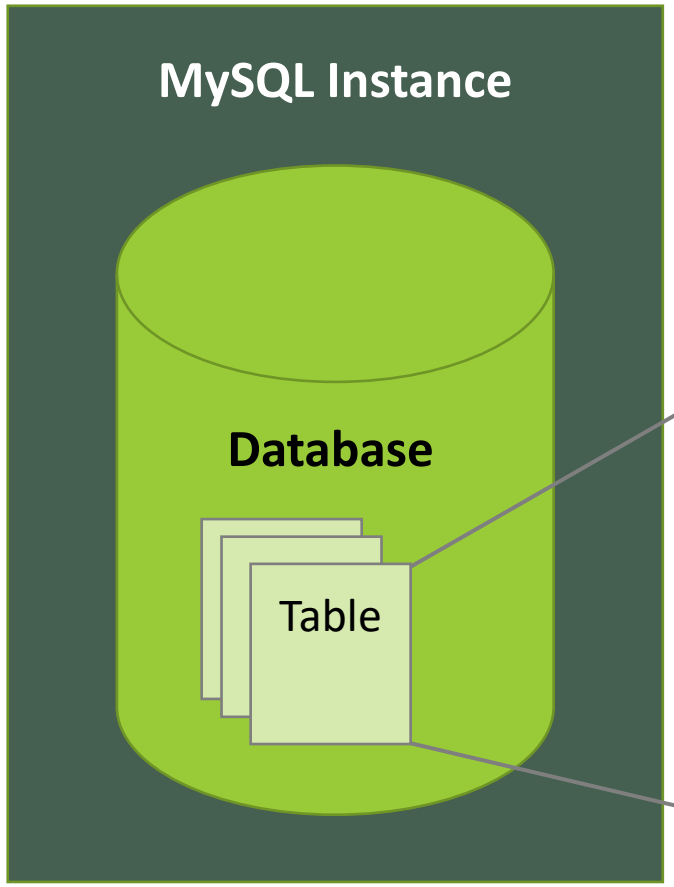
- The key design being that we are working with tabular data with interconnecting relationships
  - *Think PANDAS DataFrame!*

Terminology

- Instance: Installation or running application of the RDBMS
- Database: Collection of Tables, there can be multiple Databases on a single Instance
- Table: Matrix of data, there can be multiple Tables in a single Database
- Column: Features/data-types found within a Table
- Row (record/tuple): Group of Column-values that define a datapoint
- Primary Key: Column(s) that define a unique record in a Table
- Foreign Key: Column whose set of possible values is defined by a column within another Table
- Index: Column(s) in a Table that will be optimized for Queries
- Query: A command to extract information from a Database

Standard SQL Data Types

- Numeric
  - INT – Integer ranging from -2147483648 to 2147483647
  - BIGINT – Integer ranging from -9223372036854775808 to 9223372036854775807
  - FLOAT(M,D) – Floating point number (M: Display Length, D: # of Decimals)
  - DOUBLE(M,D) – Double precision floating point
- Datetimes
  - DATE – YYYY-MM-DD
  - DATETIME – YYYY-MM-DD HH:MM:SS (TIMESTAMP is same w/out symbols)
  - TIME – HH:MM:SS
- String
  - CHAR(M) – Fixed-length string of size M
  - VARCHAR(M) – Variable sized string of max-size M
  - BLOB/TEXT – Manner to store large objects (BLOB is for Binary objects, TEXT for text)
- NOTE: Additional types are supported (JSON), but they may differ between databases and/or have different functionality.

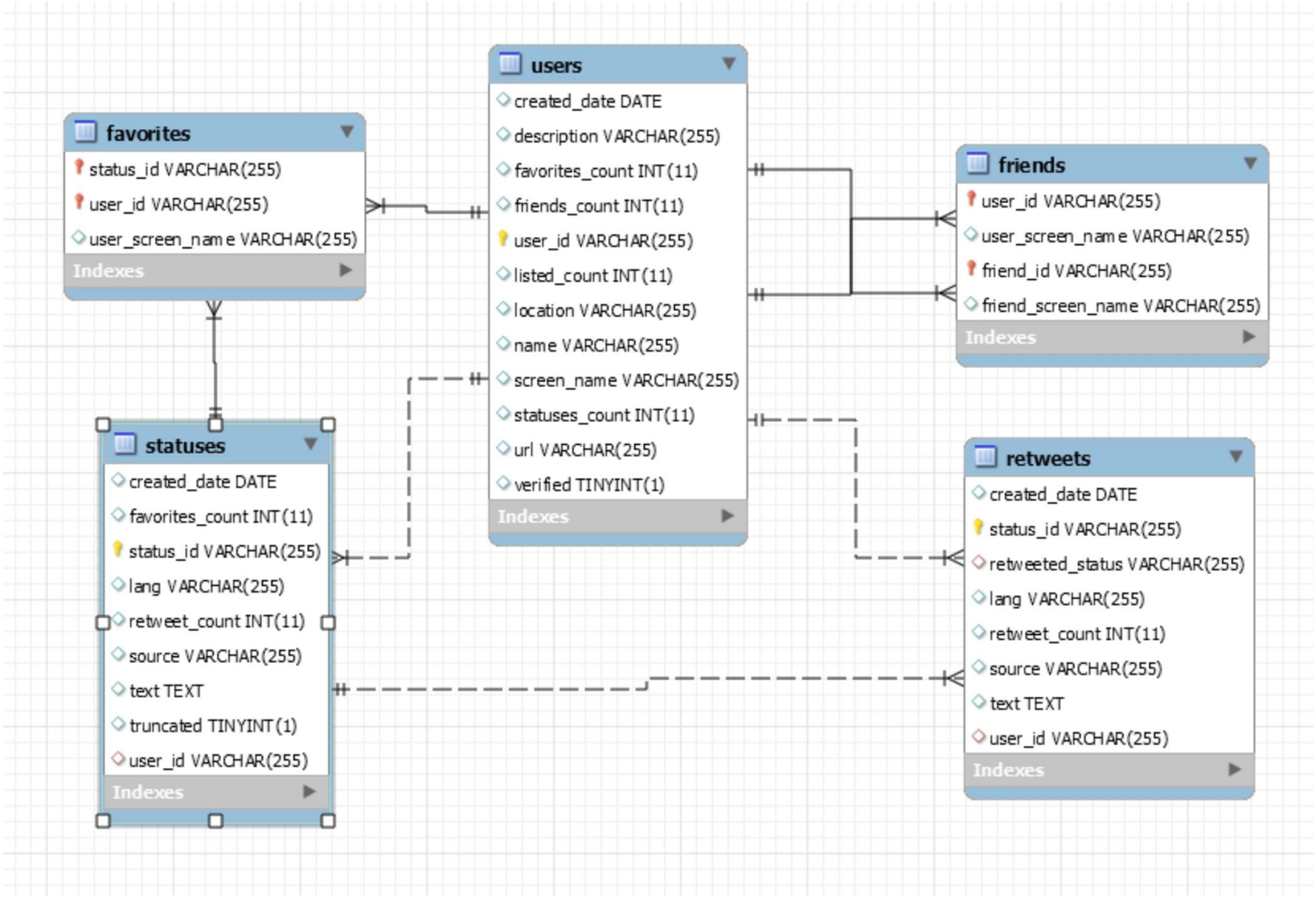


Primary Key Column

Account ID	Account Type	Date Created	Balance
123213	Checking	01/03/1986	1,003.23
124312	Savings	11/13/1999	5,452.32
312343	Investment	01/07/2003	12,321.01
142323	Checking	01/23/1996	42.34
432423	Checking	05/05/2016	243.00
INT	VARCHAR	DATE	DOUBLE

Record

Our Entity-Relationship Model



Types of relationships

- One-to-One (1-1):
  - This means that each entity in one table is linked to one entity in another table
- One-to-Many (1-m):
  - This means that each entity in one table is linked to one or more entities in another table
- Many-to-Many (m-m):
  - This means that one or more entities are linked to one or more entities in another table
- **In Class:**
  - Identify each type of relationship in our ERM Model (previous slide)

SQL

So How Do We Access Our Data?

- SQL (Structured Query Language): A Database specific language designed for managing data in RDBMS
  - While SQL has a overall generic structure, it does change from Database to Database (MySQL vs MSSQL vs Oracle vs etc.)

- We can run SQL commands directly off the database (command-line), from a GUI (MySQL workbench), or programmatically (python, java, etc.)

SQL - SELECT

Selecting Information

- To get information out of our database we need to **select** the data
  - “SELECT col1, col2, etc. FROM table”
    - Ex. “SELECT text FROM statuses;”
    - Ex. “SELECT text FROM EMSE6992.statuses”
  - We can also select all columns using the “*” symbol
    - Ex. “SELECT * FROM statuses”

- Lets run this in the workbench!

SQL - WHERE

Filtering the Information

- Obviously there will be times **where** we want to subselect data (we may be looking for something specific).
  - “SELECT * FROM statuses WHERE [NOT] condition1”
    - Conditions are usually focused around column values
    - Ex. “SELECT * FROM statuses WHERE location='Washington, DC'”

- We can combine conditions using logical statements **AND/OR**
  - Ex. “SELECT * FROM users
    WHERE location='Washington, DC' AND verified=1”

WHERE – Data Types

- WHERE conditional comparators/operators are type dependent
  - String: =, !=, or LIKE (talk about shortly)
  - Numeric: =, !=, >, <, >=, or <=
  - Datetime: =, !=, >, <, >=, or <=
    - For DATETIME's we can simply provide a DATE
    - DATE looks like DATE("YYYY-MM-DD")

- **In Class:**
  - Write a query to find all of the statuses made on the January 1st, 2019

WHERE - LIKE

- The **LIKE** clause is a special comparator in SQL that enables partial string matching
  - Using the LIKE clause with a string containing a ‘%’, the ‘%’ is treated as a wild character
    - This means ***‘%{word}%’*** is effectively the REGEX ***‘.*{word}.*’***
  - This means that we can find columns where only a certain key phrase exists

- Ex.
  - “SELECT * FROM statuses WHERE text LIKE '%AI%';”

- **In Class:**
  - Find all retweets that contain an @ symbol

WHERE - NULL

- Like many programming languages, SQL databases also have a means to indicate that no information is stored (versus an attentionally blank value) – **NULL**
  - This is necessary as every record must have a value for every column

- Likewise, sometimes we need to evaluate if there are/are not any NULL's in a column
  - "SELECT cols FROM tbl WHERE col IS [NOT] NULL"

SELECT cont.

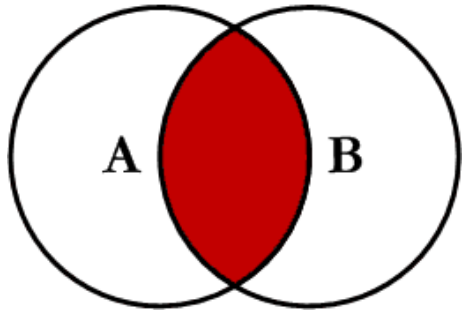
- Run the following query in MySQL Workbench:
  - “SELECT text, name, description FROM statuses, users;”

- **In Class:**
  - What is this query returning?
  - Why can this be dangerous?
  - How could we fix this?

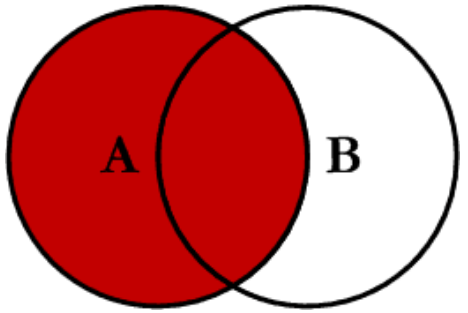
SQL - JOINS

Joining Tables

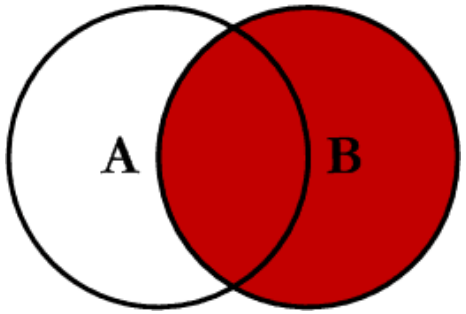
- What we achieved in the previous slide is called a **table join**
  - This is the process where we join together data in our database to get a more comprehensive view of our data
- Joins come in a number of different forms:
  - Inner Join
  - Right Join
  - Left Join



Inner Join



Right Join



Left Join

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

Proper Table Join

- The first example of a Join that we did is not advised
  - It is not supported in all SQL databases
  - It can lead to confusion and ambiguity in larger queries

- Correct table join:
  - “SELECT [cols] FROM tbl1 **LEFT JOIN** tbl2 **ON** condition;”

- **In Class:**
  - Rewrite our initial join in the correct format:
    - “SELECT text, name, description FROM statuses, users;”

Multiple Joins

- Just like everything else (column selection, conditions) we can join together multiple tables
  - However, this works slightly different than previous chains.

- Ex:
  - "SELECT * FROM friends
    JOIN users ON friends.friend_id = users.user_id
    JOIN statuses ON statuses.user_id = users.user_id;"

SQL - ORDER

Sorting Values

- Many times while working with data, we want to **order** them on some column(s)
  - This is where the ORDER clause comes into play

- “SELECT col(s) FROM tb1 ORDER BY col1, [col2, etc.]”
  - When providing multiple columns, it sorts the values in column order

- Ex.
  - “SELECT * FROM statuses ORDER BY created_date”

Selecting Top Results

- In MySQL we can get the first x results using the **LIMIT** clause
  - “SELECT * FROM tb1 LIMIT [x];”

- Ex.
  - “SELECT * FROM users LIMIT 5;”

- This can be used in conjunction with ORDER BY to get the top/bottom X results for a table

SQL - FUNCTIONS

Selecting the Minimum or Maximum

- We can get the **MIN()** or **MAX()** of columns, using their respective clauses
  - “SELECT MIN(col) FROM tbl WHERE cond;”
- Ex
  - “SELECT MAX(favorites_count) FROM statuses;”
- NOTE: MAX and MIN functions limit the number of records returned

SUM, AVG, COUNT

- SUM, AVG, and COUNT functions aggregate values across a table
  - “SELECT SUM(col) FROM tbl;”

- **In Class:**
  - Determine the average favorites_count for tweets that @ someone (include the @ symbol in the tweet’s text)

SQL - AGGREGATION

Grouping Elements

- Sometimes when selecting fields we want to **group** elements across a key identifier

- We can accomplish this goal using the **GROUP BY** clause. It states that we want to group our results by a given set of columns
  - “SELECT colX FROM tbl GROUP BY colY;”

- **In Class:**
  - Group the ‘statuses’ table by ‘user_id’

GROUP BY cont.

- When grouping elements we typically need to provide an aggregation function (AVG, SUM, COUNT, MIN, MAX, etc.), otherwise the group will pull top result for the given column

- **In Class:**
  - Modify our previous exercise to look for the AVG favorites_count and retweet_count per user

Conditional Grouping

- When grouping elements in tables we usually want to **have** certain conditions applied to our groups

- We can thus us the **HAVING** clause to further filter our groups
  - "SELECT colX FROM tbl GROUP BY colY HAVING cond;"
  - *NOTE: The HAVING clause can only be applied on GROUP BY columns or on aggregate functions*

- **In Class:**
  - Find the average retweet_count for verified users

SQL – Special Functionality

Selecting Distinct Elements

- When selecting columns we can ask for **distinct** results, to ensure that there are no duplicates in the resulting data
  - “SELECT DISTINCT(col) FROM tbl;”

- Exercise:
  - How many dates are stored in the statuses collection?

IN Clause

- The **IN** clause allows us to check if a value exists in a list
  - “SELECT * FROM tbl WHERE col in (value1, value2, . . .);”

Aliasing

- Many times it is easier or necessary to provide an alias for a column or table
  - An alias is simply an alternative representation for that element
  - “SELECT user_id AS id FROM users;”

- How do you think we alias a table? Why would we do this?

Nested Queries

- Nested queries allow us to query the results of other queries to form complex requests

- Nested queries can appear in a:
  - SELECT clause
  - FROM clause
  - WHERE clause

Nested Queries cont.

- FROM
  - Select avg(retweet_count), user_id FROM
    (SELECT retweet_count, user_id FROM statuses
    WHERE favorites_count > 500) as tmp_tbl
    Group By user_id;

- WHERE
  - SELECT * FROM statuses
    WHERE user_id IN (SELECT user_id FROM users WHERE verified=1);

Inserting Information

Adding New Information

- Frequently we will want to store new or generated information, we can achieve this by **inserting** the data
  - INSERT INTO tbl (field1, field2, …) VALUES (value1, value2, ….);
    - NOTE: You do not need to insert values for all fields
    - NOTE: You do not need to provide a field list if you provide values for all fields in order

- INSERT INTO users(user_id, screen_name) VALUES ('001', 'Joel Klein');
- INSERT INTO Friends VALUES ('001', 'Joel Klein', '001', 'Joel Klein');

Updating Information

- Sometimes we don’t want to insert new information, but rather **update** an existing record
  - UPDATE tbl SET col1=value1, col2=value2, … WHERE cond1[, cond2, …];
    - NOTE: The conditions are typically to align to the primary keys of a specific record

- UPDATE users SET favorites_count=78 WHERE user_id='001';

Delete, Alter, Indexes

Deleting Records

- Works very similar to a SELECT clause, but we just replace SELECT with DELETE
  - DELETE FROM tbl WHERE cond1, etc…;

- DELETE FROM users WHERE user_id='001'

Alter Table

- Altering a table gives us the ability to add/remove/modify a tables columns.
  - We can do other things, but these are the most common
  - ALTER TABLE tbl (Drop, Add, Modify) col (datatype);

Indexes

- Indexes are like phonebooks for your table. They store a quick lookup for all values in a table.
  - Indexes can be for indexing or fulltext (for optimized text queries)
  - Indexes can be multi-column
    - In these situations queries are indexed on the first column, and then the second column (e.g. think “group by” statements)
  - Indexes can be applied to any number of columns any number of times

End Slide

EMSE 6992 – DBMS for Data Analytics

BACKUP SLIDES

Selecting Distinct Elements

- When selecting columns we can ask for **distinct** results, to ensure that there are no duplicates in the resulting data
  - “SELECT DISTINCT(col) FROM tbl;”

- Exercise:
  - How many dates are stored in the statuses collection?

Aliasing

- Many times it is easier or necessary to provide an alias for a column or table
  - An alias is simply an alternative representation for that element
  - “SELECT user_id AS id FROM users;”

- How do you think we alias a table?

IN Clause

- The **IN** clause allows us to check if a value exists in a list
  - “SELECT * FROM tbl WHERE col in (value1, value2, . . .);”

Nested Queries

- Nested queries allow us to query the results of other queries to form complex requests

- Nested queries can appear in a:
  - SELECT clause
  - FROM clause
  - WHERE clause

Nested Queries cont.

- FROM
  - Select avg(retweet_count), user_id FROM
    (SELECT retweet_count, user_id FROM statuses
    WHERE favorites_count > 500) as tmp_tbl
    Group By user_id;

- WHERE
  - SELECT * FROM statuses
    WHERE user_id IN (SELECT user_id FROM users WHERE verified=1);