mySQL, Database Design, and Usage:

Sean al-Baroudi

August 2, 2022

Week 1:

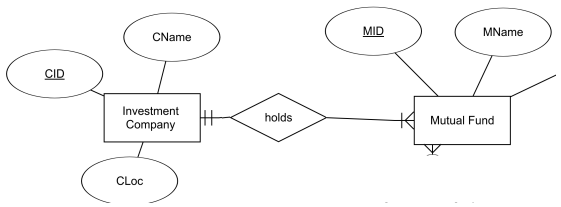
- Good Data Storage Allows:
  1. Easy Retrieval
  2. Easy Updating
  3. Access in parallel.
  4. Data Consistency
  5. Space Efficiency
  6. Low Access (R/W) Time
- **Simple Rules for Designing Relational Database Structure:**
- **idea:** Table Design should follow principles of Tidy Data:
  - Each column represents a unique piece of data.
  - Rows are unique (upto one column - primary key).
  - R and C are independant; not position dependent.
  - Try to atomize tables: never have attributes that have multivalues (like lists). This makes insertion and retreval more complex: use another table instead.

  item Tables are the smallest logical subsets of data.
- **Mathematical Abstraction for Tables: Naive Set Theory!** Sets require unique elements to be in them (not duplicates). This is achieved by having tables that have items that are unique (upto one property).

Week 2: Database Design Tools:

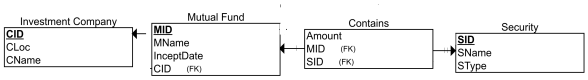
Entity Relationship Diagrams:

- For our database designs, it is useful to have graphical representations of our objects and relationships; this allows us to get a big picture and spot mistakes in our design.
- Entity-Relationship Diagrams: used to encode conceptual relationships in your dataset.
- **ER Objects and Grammar:**
  1. **Entity** □: Represent Categories of similar, but unique measurement. For example: A student number, timestamp or area code.
  2. **Attributes** ◯: Individual unique measurements within a catagory.
  3. **Relationship:** ♢: Represents a some mapping between two entities.
  4. **Cardinality Constraints:** Represent the many or one mappings between entities. The following symbols are used: | - one, $\leq$ - many, (a,b) - min to max, $\mathbb{M}$ infinite and O - optional.
  5. **Grammar of a Relation:** Subject → Verb → C.C → Object. More generally, this is a **data triple.** in more general language.
  6. **Note:** Diagrams can be read in both directions (switch subject and object). Cardinality constraints need to be on both sides of the mapping.
  7. Entity Strength ‖ Indicated with a double border; double border means weak entity.
  8. Derived Entity: Indicated with dashed border.
  9. Multivalued Entity: Indicated with Double Border.
  10. Example of An E-R Diagram:



Relational Schemas:

- **Key Definitions:**
- Primary Key (PK): A unique attribute in a tuple/row that makes it unique.
- Foreign Key (FK): An attribute in a table that is a primary key in another table.
- A mapping table that implements many to many relationships may have exclusive foreign and partial keys. The keys put together form a **Composite Key**
- **Relational Schema:** Another graphical representation, that is closer to Table and Column structure; it is akin to a blue print for a database.
- **RS Construction:**
  1. You represent a table as a box with a title (table name).
  2. Foreign Keys are indicated with (FK) in the table.
  3. Unique Keys are represented with a (U) in the table.
  4. Primary Keys are Keys are underlined.
- **Relationships Between Entities:**
- **Many to One Relationship: 1S - R - mO** If a Subject has MANY Objects, you put the subjects PKEY as a FKEY in the Object Table. [See Diagram Below]:
  Why is this done? If we tried to record our many to many-one relationship in the subject table, we would have to have **multivalued lists** in each row of our subject table. Instead, we can just point to the subject with each entry of the object table. This allows us to use simple attribute types.
- **Many to Many Relationships mS - R mO :** Here, we need a relation table with two foreign keys; the mappings are just pairs of each instance of a foreign key. Each row is a specific mapping between two elements. We can also add additional information as further attributes for every entity instance. [Many to Many Mapping relationship Diagram].
  Why? We can just constrain the FK table instead of having to do a Join operation.
- **Many relationships between two entities:** Each Foreign key must be renamed, as you have the same foreign key mapping two different relations.
  Why? You cannot have two independent relations using the same foreign key. Not enough dimensions to encode to relations (using a simple col data type).



-
-

Structural Interpretation and Equivalence of ER and RS Diagrams:

- Entity:Attritube $\iff$ Subject:Property
- An instance of an entity is a **row or tuple.**
- An instance of an attritube is just a cell or single piece of data.
- **Note: Relationships are implemented as TABLES**. Tables represent instances of data in catagories, but are also used to represent abstract connections between our sets of data as well.
- Every Entity must have one unique attribute (to avoid duplicates - and for set theory model to be valid).
- **Weak Entity:** One in which only a partial key is provided. Partial Key can become a full key if it is joined with a foreign key in another entity.
  Example: Building Number and Apartment Number; two Partial Keys Combine to form a unique apartment identifier.
-
-

Week2:

Outline of making SQL Queries:

- Database queries are very powerful: if unconstrained, operations can blow up and become very computationally intensive; as many deployed mySQL databases are used by a group of users, it is very difficult
- **Query**: A line of code that specifies a specific set of data from the database, and how the database should format it.
- Six main SQL Keywords (BoldFace indicates required):
  1. **SELECT:** $< cols >$ or the data that I want
  2. **FROM:** $< tablename >$ or these databases and tables
  3. WHERE: $< constraints >$ on rows.
  4. GROUP (BY): $< cols >$
  5. HAVING: $< constraints >$ on groups or aggregates
  6. ORDER (BY): $< ordering$ this field or list...
  7. LIMIT: N: restrict number of rows.

SQL Quirks:

- mySQL starts counting at zero.
- Values not entered indicated with NULL.
- Quoting Table Names: Use single-quotes if a name has a space in it. Use backquotes if a table/col name has an SQL Reserved word nested in it.
- When a query is done, SQL grabs data from the top of the table by default!

Basic Console Usage:

- You can load multiple databases in the beginning, with the USE command:

$$USE\ database1,\ database2, ...;$$

- Once your database is loaded, you need to see which tables are in it. Use the following command to list them:

$$SHOW\ TABLES\ FROM\ databasename$$

- Before you make a query, you must use the SHOW or DESCRIBE command on tables. There is no major difference between them (at a beginner/intermediate level).

$$DESCRIBE\ tablename$$

- **Columns Types:** Primary (PRI), Unique (UNI), Multiple (MUL) - non-unique, Empty (not indexed).
- in mySQL, there is usually a column that is denoted the primary key; this column is used in a search index to speed up queries for larger tables.
- **Example: Examine Data in a Table:**

```
SELECT col1,col2...
FROM table_name
LIMIT N
OFFSET Q;
```

- Exporting your QUERY: in SQL, the "SELECT, FROM..." statement returns a response. This can easily be written to a variable, and then written to a file using SQL functions:

```
var_name = SQLQUERY
varname.csv('filename.csv')
```

More on Columns:

- **Table Qualifiers:** Dealing with Shared Names between tables: Use Object Notation:

$$Tablename.colname$$

  To avoid ambiguity.
- **Deriving Columns:** In the SELECT statement, you can do basic operations to yield a new, derived column in your response. For Example:

```
SELECT col1 + Col2
FROM table
...
```

Some Keywords:

- **Applying Constraints to Rows using WHERE:**

```
SELECT col1,col2... FROM table_name
WHERE [EXPR ...];
```

- The WHERE statements takes expressions and applies them one by one to each column, which narrows down the response set. Multiple expressions are joined by Logical Operators (AND, OR).
- The usual 1-D comparison operations apply $(!=, >, \leq)$
- **Other Comparison Keywords:**
- IN: Used to check if a column actually matches values in a list. Used as follows:

$$WHERE\ col1\ IN\ ('val1', 'val2'...)$$

- LIKE: Used to apply pattern matching to particular column entries. Use basic regex principles to filter words.

$$WHERE\ col1\ LIKE\ '\%STRING\%'$$

  The % sign represents a wildcard.
- **Aliasing:** Sometimes it helps to rename difficult entities in SQL queries. Use the AS statement inline in the SELECT and FROM lines of a query, to so do. For example:

```
SELECT col1 AS C1, col2 AS C2,
FROM Longtablename AS Tab1
...
```

- **Removing Duplicates: The DISTINCT Keyword:**
- This keyword is used in the SELECT statement; *to avoid any duplication of data across entire rows.*

```
SELECT DISTINCT col1, col2...
FROM tablename
...
```

  Recall that rows are non-distinct if two exist in which all fields match. If **just one field** differs, they are still considered distinct.
- Interaction: DISTINCT and LIMIT: DISTINCT will cause unique results to be gathered, upto the LIMIT imposed.
- **Sorting your Data: ORDER BY:**
- Once a response set has been given, you can order the rows in particular ways. For a given colname, you specify Ascending (ASC) or Desending (DESC). Each ordering is applied sequentially, as it is listed.
- Order matters! Not commutative. **Example: Ordering Multiple Columns**

```
SELECT col1, col2, col3
FROM tablename
Where Constraints
ORDER By col1 ASC, col2 DESC...
...
```

Week 3:

Data Analytic Habits:

1. Being paranoid and skeptical is a good thing!
2. Explore the dataset as thoroughly as you can; don't trust that it is in good shape. Expect massive cleaning and transformations to make is usable.
3. If you see a Positive Anomoly, asssume its due to chance or error (or hypothesis test to prove it).
4. If you see a Negative Anomony, assume it indicates a deeper problem, until proven otherwise.
5. Seek details, and make/break your own hypotheses.
6. Periodically ask yourself: Is my data correct? How could it be off?
7. Your Goal: to get correct answers to real like questions; these answers must stand up critical analysis.

Grouping and Constraints of Groups:

Aggregate Functions:

- Aggregate Functions: Are applied in the SELECT section of a Query, and map functions to numbers:

$$AggFun:\ Cols\ \rightarrow\ \mathbb{R}$$

- There are many simple aggregate functions that accept Cols as an input. Including AVG(), COUNT(), MEDIAN(), ISNULL(), etc. Read the SQL functions page to understand their usage.
- ISNULL(Col): COL → (1 and 0 Col).
- Aggregation can only be done in HAVING and SELECT statements. You can't use it in a WHERE Clause, as your derived columns don't yet exist when WHERE conditions are processed. See the SQL Statement order chart, below.

Group By:

- This keyword causes SQL to stratify your result tuples in terms of the possible values of the Group Column. **For example:** if you have a gender column, all of your rows would be ordered by Femaile, and then the Male set would appear right after them.
- **Group By Multiple Columns:** if you do this, you will effectively see **catagories and subcatagories** down the rows. The nesting is dependent on the order in which you list the group columns.

HAVING:

- This is a WHERE constraint that is **applied to groups**. It is very important to know when to apply constraints to individual tuples, and groups of tuples. Suffice to say, swapping constraints between these two catagories can produce wildly different results.

Derived Columns:

- **Def:** Derived or Computed Column: One created from operations on, or a function on our original set of columns. For Example: *SELECT c1 + c2 AS ctotal* is a derived column.
- **WHERE cannot utilize Derived Columns:** You cannot use WHERE to constrain derived columns. These are virtual columns that are in memory, and not apart of the original table.
- *How do you use your derived columns in a where statement?* You need to turn a **virtual column into a derived table**. Return the table+derived columns with a subquery, and then use WHERE in the outer query!

Aggregate Functions:

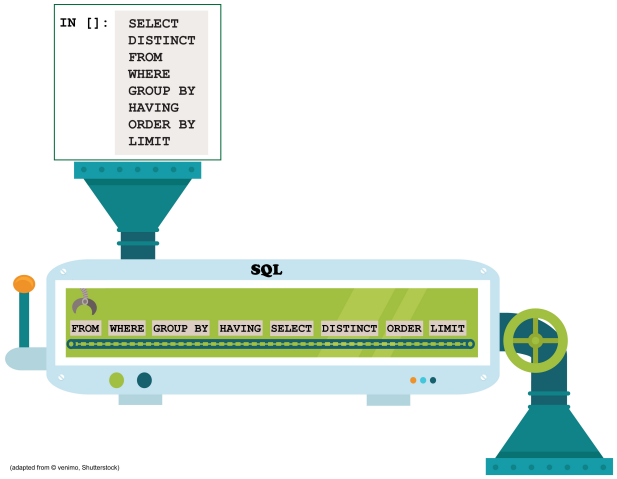
- When Aggregate functions are calculated on their own without the GROUP By statement, the entire table is just one default "group".
- **Note: Aggregate functions specified are calculated for the lowest sub-category in the query.**
- When multiple groups are specified (in categories and subcatagories), *the aggregate function is then applied to each of the most nested subcatagories*
- **Result: GROUP BY allows us to map aggregate functions to more than one value.**

Other Common Group By Pitfalls:

- Spreadsheets are not SQL Tables: Spreadsheets do row and column operations; in SQL the transformations and filtering can be much more complex.
- Queries that **mix aggregated and non-aggregated** results will fail because the dimensional output may not match. You can get weird or erroneous results.
- **Mixing Non-Aggregates and Agreggates while Grouping:** If you apply aggregates to a list of groups, but then also specify a non-aggregate (unaltered) column in the SELECT statement, you will have a many to one mapping for every row in the table. mySQL will just choose random values from the subset to put here; you get worthless data in this column if this is done.
  Result: Aggregate rows are mixed with single record data - which doesn't make sense.
- **Solution:** To avoid the many to one, random selection problem, include non-aggregates in the GROUP BY statement.
- If you mix aggregates and columns together, mySQL will just pick random col values to stuff into the output, with no apparent rhyme or reason.
- In general: mySQL will do its best to deliver an output, even if what you typed in doesn't make sense. This masks errors and allows for sloppyness.
- **Avoiding dimensionality mismatch:** every aggregate in a select statement should be used in a GROUP BY statement.

SQL Query Parsing Order:

- It is important to realize that the SQL query is structured in a Human Centric Way. The way SQL interprets a query is not at all like practitioners are taught to concieve of queries, as evidenced by the diagram below:



- Notice that SELECT occurs after HAVING; this means constraints with derived columns are processed before derived columns are even identified!

Joins:

Basic Idea:

- Two tables are joined when we connect the rows of one table, to that of another.
- To bind rows together, a key needs to be specified for each row. If the two keys of the rows match in the join condition, the two rows are bound in the new table.
- Typical Join Syntax:

```
table t1
JOIN
table t2
ON t1.key ?= t2.key
```

- A table join produces a new temporary table. This can be embedded in multiple areas of a query (if used as a subquery)
- Mathematical Model: Cartesian Product (with contraints).
- Constraints are applied using rows that share keys or other values; this cuts down the output size, and useless rows we don't need.
- Note: You don't have to rename the tables (in the example syntaxes below).
- **ON keyword**: This is used to apply constraints to the rows, as does WHERE. You can also use WHERE, but separating row conditions from join conditions with ON gives a cleaner look.

Manual or Implicit Join:

- You do not have to specify a "JOIN" keyword to do this form of join. It is implemented by applying tuple level constraints with the WHERE clause. As in:
- **Constraint Type:** One to One

```
SELECT t1.cols,...,t2.cols
from t1, t2
WHERE t1.col1 = t2.col2
```

- Note: WHERE was used here. This is the most basic way to do a join.

Inner Join (also Self-Joins):

- This is the most restricted form a join: There must be a key match in both tables in order for a join row to be formed.
- If there is no corresponding match for one row in another table, it is omitted from the newly built join table.
- Constraint Type: One to One
- Syntax:

```
SELECT cols....
FROM table1 t1
INNER JOIN
table2 t2
ON t1.col1 = t2.col1 ....
```

- The INNER keyword is optional (in mySQL).
- For a self-join: use the same table twice.

Directional Joins (Left and Right):

- These can be LEFT or RIGHT in direction. Between two tables, one is kept in its entirity, and the other is paired up using the WHERE and ON constraints. The table that is kept is on the literal side of the directional keyword.
- Constraint Type: Many to One or One to Many.
- Syntax:

```
SELECT cols...
table1 t1
LEFT/RIGHT JOIN
table2 t2
ON t1.key = t2.key
```

- **Understanding Direction:**
- A LEFT JOIN B → A is kept in full.
- For A Left Join B: *A is the table being joined*, *B is the table that is joining* (A)
- **Tip:** The direction specifies which table is the superset. LEFT JOIN → Left side table is superset.
- The number of rows in the superset table bounds the number of rows in the constructed table *(assuming no duplicates exist in either table)*.
- **Directional Join Anti-Commutativity:**

$$A\ (Left\ \ Join)\ \ B\ \ =\ \ B\ \ (Right\ \ Join)\ \ A$$

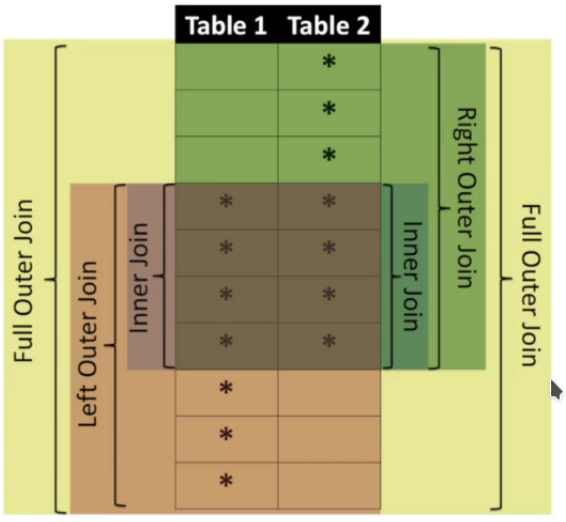
Cross Joins:

- mySQL does not support full outer join; this is generated with L and R Joins and Union operations**
- This join **can blow up**. Two tables of size O(M) and O(N) will end up with an unconstrained join table of size O(MN)! **SQL will always give you the most number of results, by default.**
- Constraint Type: Many to Many.
- Syntax:

```
SELECT cols...
table1 t1
OUTER JOIN
table2 t2
ON t1.key = t2.key
```

- Both the dangling nulls of Left and Right Joins will be included. Null to Null will not be added to the temp table.

The Join Infographic:



Worries about Joins:

- Tables can easily have duplicate rows (both tables in a JOIN). If this causes multiplicitive effects that can be substantial.
- Other Comparison Operators: For all joins, you can use operations other than "=" in the ON clause. This will of course increase the size of the produced table.
- **TEST GUARD:** Do elementary counts on row uniqueness (via GROUPING) to assess this risk.
- Look at query output tuples, to normalize and make sense of your aggregate calcualtions.
- Never do a series of joins; always *build them up one JOIN at a time*.
- Check-Expect: What you output should match your prediction of what you were supposed to get. If it does not match or deviates, you need to investigate and correct on the spot (don't propogate uncertainty and error).

1 Week4: Subqueries and Derived Tables:

On Expressions, Derived and Temporary Tables:

- SQL has strict grammar rules for all statements.
- An Expression is a statement (simple or complex) that returns a value.
- The Values that an SQL Expression can return are; Single Value, Column and Table.

Temporary Tables:

- This is a table that persists beyond the lifetime of a given query in a session.
- These tables persist for the length of the session.
- Syntax: To add to session:

```
CREATE TEMPORARY TABLE
(Subquery to make a table)
```

- Syntax: To drop the table in session:

```
DROP TEMPORARY TABLE
```

- The aliasing that occurs during the subquery references this table in the session.

Derived Tables:

- Is considered a virtual table (not persistent, exists only in memory).
- Subqueries produce derived tables. However not every subquery is a derived table.

$$Derived\ Table \rightarrow \text{Produced by Subquery}$$

  Only **FROM clause** subqueries produce derived tables.
- A derived table is a subquery that is aliased.
- A derived table **must be aliased** in SQL.

Subqueries:

Basics:

- Usage Cases:
  1. Used to isolate parts of a complex query; useful for incremental debugging and teasing out errors.
  2. Can run faster than a join (in some cases)??
  3. Testing Membership: Via In/Not in, or Exists/Not Exists
- Subqueries can be inserted anywhere an Expression can be Inserted. They are typically found in WHERE and FROM clauses. These cases will be covered below:
- ORDER BY statements must be omitted from a subquery
- (The Subquery must be encapsulated in parenthesis)
- **Output:** Can be a value, Column or Table (SQ is an expression in the end).
- You can reference the result of a subquery, but you cannot reference the table calls executed in a subquery. For example:

```
SELECT derivtable1.col1, u.col2, derivtable.col3
FROM (SELECT * from table u) AS derivtable1
```

  Referencing u would result in an error: the table was used to output our derived table - and can no longer be referenced after the subquery has ended.

FROM Clause SQ:

- These are considered to be derived tables. They persist for the lifetime of the outer query.
- These **subqueries** must be aliased.
- Syntax:

```
SELECT col1 col2
FROM
   (Subquery)
...
```

WHERE Clause SQ:

- For our WHERE filtering expressions, we can use SQs to return values, or columns for comparison. There are a number of constraints:
- **Single Value Comparison:**

```
SELECT X
FROM Y
WHERE Z > (SQ:Single Value)
```

- **IN or NOT IN Comparison:**
- This checks if a given column or quantity is in a Column of values.
- `SELECT X`
  `FROM Y`
  `WHERE Z IN/NOT IN (SQ:Column)`
- **EXISTS and NOT EXISTS Checks:**
- These are functions that accept subqueries as input.
- If the SQ returns (in place) one or more rows, EXISTS returns TRUE. Else, it returns false.
- NOT EXISTS has reversed logic.
- Note: Exists is a bit slow, as the function is applied to every row [2]

```
SELECT X
FROM Y
WHERE EXISTS(SQ:Column or Table)
```

- **Correlated SQs:**
- This is simply a subquery that references entities in the Outer Query that it is nested in.
- It is easier to understand this with a practical example:

```
SELECT
    productname,
    buyprice
FROM
    products p1
WHERE
    buyprice > (SELECT
            AVG(buyprice)
        FROM
            products
        WHERE
            productline = p1.productline)
```

- We can clearly see that p1.productLine is used in the inner subquery.

Control Flow Functions:

- Note: Don't confuse these with control flow statements: these are used in full SQL scripts with bindings and other constructs. Our functions here are used strictly for queries.
- **Definition: Statement:** A line of code that contains expressions, and may bind a value to a name.
- These conditional functions are ment to be placed in SELECT and WHERE statements; they are used to assign group values to a *derived column*, using conditional tests.
- You can reference a derived column generated via an IF/CASE in the GROUP BY clause.

IF Function:

- These can be used in SELECT or WHERE statements.
- Syntax:

```
SELECT
  IF(condition, result_true, result_false) col1,
  ....
FROM X
...
```

- The condition must evaluate to a Boolean. The results must be single values.
- The IF Funciton will be called for every matching row of the table.
- How it works: If you call:

```
SELECT
   5 col1,
   col2,
FROM table
```

- SQL will literally set every row of col1 to 5; it does this in general for basic arithmetic expressions. As IF() returns a value, SQL is set up to do this for every row.

Case Expression:

- There are two types of Expressions: The Value type, and the Condition Type:
- CASE Value Syntax:

```
SELECT
col1,
(CASE value
WHEN compare_value_1 THEN result_1
WHEN compare_value_2 THEN result_2
ELSE result END) As Col2,
...
FROM X
...
```

- CASE Condition Syntax:

```
SELECT col1, col2
   CASE
      WHEN con1 THEN result_expr1
      WHEN con2 THEN result_expr1
      ...
      ELSE result_expr n
   END AS name
FROM ...
```

- END is necessary, but the ELSE is not.
- The case expression must be given a derived name.
- Logical Operators: NOT, AND, OR: Have an order of presedence (as presented), and can be put in parentheses to make logical expressions unambiguous.

Residual Questions:

-

References

[1] `https://www.coursera.org/learn/analytics-mysql`

[2] `https://www.techonthenet.com/mysql/exists.php`

[3] `http://www.mysqltutorial.org/`