**Introduction to Databases**

* A database (DB) is any collection of related information. Examples: shopping list, text file, excel file, amazon db.
* A database management system (DBMS) is a special software program that helps users create and maintain a database. A DBMS isn’t the actual DB, but rather, it is the software application that creating/maintaining/deleting/updating information from the actual database.
* Graphical user interface, text, application

  Description automatically generated
* CRUD represents the main operations we will do with information in a database.
* A picture containing text, clipart

  Description automatically generated

Types of Databases

Graphical user interface

Description automatically generated with low confidence

SQL

* A relational database is very much like an excel spreadsheet.
* Graphical user interface, application

  Description automatically generated
* SQL is a programming language. MySQL is RDBMS.Text

  Description automatically generated

No SQL

Diagram

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Summary

Text

Description automatically generated

**Tables and Keys**

Table

Description automatically generated

Columns define a single attribute such as the name column. Inside the name column, we have the names of all the students. A row is an individual entry in the table which represents a single student.

Primary Key

We always want to have a special column called the primary key. The primary key is an attribute that uniquely defines a row in the DB. The primary key can be anything, it can be a number, a string of text, whatever. Just make sure the primary key is unique. The student id attribute is the primary key. Notice that even the two jacks have the same name and major, they have different primary keys.

There are two kinds of primary keys, surrogate keys and natural keys.

A surrogate key is just a key that has no context in the real world, it is just used for DB purposes. In the example below, the number 100 is just some random number we chose. 100 is used to represent the entry in the DB but it has no meaning outside of the DB.

Table

Description automatically generated

A natural key is a key that has context in the real world. For instance, below, the primary key is the social security number. While the SSN plays a role in the database for uniquely defining each entry, it also has context in the real world since that SSN belongs to that person.

Table

Description automatically generated

Foreign Key

* A foreign key is an attribute we can store in a DB table that will link us to another DB table.
* A foreign key allows us to define relationships between tables.
* A foreign key is the primary key of another table

Table

Description automatically generated Table

Description automatically generated

Notice we have two DB tables. In the Employee table, there is a branch\_id attribute which is a foregin key. The foreign key stores the primary key of a row in another table. For example, we can say that employees belong to a specific branch in the company. We can define which branch a specifc employee belongs to by referring to the primary key of the branch table. Ex: Jan Levinson has a branch id of 1. So when we go to the branch table, we look for the entry with a primary key of 1. This entry has a branch name of corporate. So Jan is in the corporate branch.

In this example, the branch table has a foreighn key which is the manager\_id attribute which points to the employee id of the manager. So for example, if we look at the branch with id of 2, the manager of the branch has an id of 101. Thus, we can go to the employee table and find the entry where the employee id is 102, which corresponds to the person of michael scott.

Notice that the employee table has a foreign key(branch id) that links it with the branch table. As well the branch table has a foreign key(manager id) that links it with the employee table.

Note that a table can have more than 1 foreign key as shown below. In the tabe below each employee not only has a branch they belong to but they also have a supervisor. We can find out more information about that supervisor by going to the employee with id of whatever our current employee’s super\_id is. For example, Michael Scott’s supervisor has an id of 100. When looking at the employee with id of 100, we get Jan. So Jan is Michael’s supervisor.

Table

Description automatically generated

Composite Key

* A primary key that needs two attributes

Table

Description automatically generated

Branch supplier tells us who the supplier for the different branches is. For instance looking at the first entry we can see that Hammer Mill supplies paper for the branch with id of 2 which is Scranton. Notice that the composite key consists of two attributes, the branch id and the supplier name. We need both these attributes since the branch id by itself does not uniquely define each row and the supplier name by itself does not uniquely define each row. Only together can they uniquely define each row. For instance, the supplier name of Hammer Mill appears twice. But the combination of Hammer Mill supplying to branch 2 only appears once.

It is possible that the attributes to create a composite key can actually be foreign keys as well as shown below. Below we introduce the client and works\_with branch. The client branch is the customer and the works\_with table tells us which employee works with which client.

Table

Description automatically generated

Consider the works\_with table. Again, notice that the emp id and the client id by themselves cannot uniquely define a row. However, together the two attributes uniquely define a row. As well, notice that these two attributes are foreign keys. The employee id is going to refer to an employee in the employee table and the client id is going to refer to an employee in the employee table. For example, employee 101 has sold client 401 267000 sales of something. Looking at where the foreign keys point to, this says Michael Scott (employee 101) has solid Lackawanna country (client 401) 267000 sales of something.

**SQL Basics**

* + A picture containing text, person

    Description automatically generated
  + SQL is a programming language as it provides instructions to a RDBMS
  + In order to update/delete/etc information in a RDBMS, we need to use SQL.
  + Text

    Description automatically generated
  + SQL has formal specifications which defines how SQL needs to be used
  + There are a bunch of RDBMs such as postgres, mysql which implement SQL, but each slightly differently.
  + SQL is used on all the RDBMs, just differently
  + Text

    Description automatically generated
  + Note: Schemas is the layout/design of the DB

Queries

Queries is just asking the database for some information.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

**MySQL Windows Installation**

* Installing MySQL
  + <https://youtu.be/OM4aZJW_Ojs>
  + Password: a!c5AA8E
* Using MySQL Workbench
  + <https://youtu.be/7S_tz1z_5bA?t=909>

**Syntax**

* use all caps for SQL keywords. Note that all caps is convention even though sql is case insenstivie so sElecT is the same as SELECT
* Every command ends with a semicolon
* Line breaks are ignored so we can go to the next line but SQL still thinks its one line

MySQL Workbench

* click the first icon to execute all the scripts in a file. Click the second icon to execute the script the cursor is at in a file.

Comments

* + n SQL, there are two types of comments: single-line comments and multi-line comments.
  + Single-line comments start with two dashes (--), and continue until the end of the line. For example:
  + 
  + Multi-line comments start with /\* and end with /. Anything between the / and \*/ characters is considered a comment. For example:
  + Graphical user interface, text

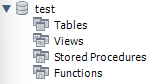
    Description automatically generated

**Data types**

* Below are the datatypes that can be used in MySQL. Most of these data types can be used in any RDBMS, but they might differ a bit depending on the datatype.

|  |  |
| --- | --- |
| INT | integer |
| DECIMAL(M,N) | decimal numbers. M is the total number of digits we want to store for the number. N is the number of digits we want to store after the decimal point. |
| FLOAT | Decimal numbers with 15 digit precision. |
| VARCHAR(L) | string of text of length L |
| TEXT | string of arbitrary length |
| BLOB | Binary Large Object. A structure that stores large amounts of binary data. |
| DATE | ‘YYYY-MM-DD’ |
| TIMESTAMP | 'YYYY-MM-DD HH:MM:SS' |

Create a database:

* + 
  + In the above example, test is the name of the database
  + This will create a test database which is shown in the schema tab on the left:
  + 
  + Note that MySQL workbench doesn’t update the UI so you have to click the refresh button to see the change

Drop a database

* + 
  + In the above example, the database with a name of test is dropped/deleted
  + Dropping a db deletes all the data inside it, so we rarely use it

Use a database

* + In order for our sql queries run against a desired database, we have to use/target that specific db.
  + To use a db, we run 
  + Note that In the above example, the db with a name of record\_company is used

Creating a table

* + Tables have columns that represent the different properties of the object it’s representing.
  + Thus, when we create our table, we need to tell it what columns to create with
  + Graphical user interface, text, application

    Description automatically generated
  + In this example, we create a table called test which has an “id” and “name” column. The id must be of type int, it cannot be null. Additionally, this id is automatically created each time we add a new band entry. So the first band entry has id 1, the second has id 2, and so on. Additionally, the id column is the identifier of our table (what distinguishes different entries), the id acts as a primary key which we let sql know by saying PRIMARY KEY (id). The other column we have is the “name” column and it stores strings with a max length of 255 and the name cannot be null.
  + This new table is reflected in MySQL workbench if we refresh.

Composite Key

* + Suppose we have this following table:
  + Table

    Description automatically generated
  + Notice that we need both the branch\_id and supplier\_name properties to uniquely identify a row. Thus, this table makes use of a composite key.
  + Graphical user interface, text

    Description automatically generated
  + This would be the command to make that table, notice there are two arguments passed to the primary key.

Altering a table

* + If we want to alter an existing table by adding a new column, we can run the following script
  + 
  + In the above example, we are alteraing the table with a name of ‘test’ by adding a new column called ‘another\_column’ and the value of the items in ‘another\_column’ is a string with a max length of 255.

Dropping a table

* + 
  + In this example, we drop the table whose name is ‘test’

Foreign Key/Table Relationships

* + A foreign key is a key that references another table.
  + Suppose we have this table:
  + Graphical user interface, text, application

    Description automatically generated with medium confidence
  + We can now create this following table that is related to the one above:
  + Graphical user interface, text

    Description automatically generated with medium confidence
  + Notice when we create an album, we want its data to be related to its band since an album belongs to a band.
  + To do so, we reference the band table within this album table by having the “band\_id” column. This column will store the value of the band\_id that corresponds to the band that owns the current album. To indicate to sql that this “band\_id” is not just any integer, but the id column of a band table entry, we have the line FOREIGN KEY (band\_id) REFERENCES bands(id).
  + The syntax for foreign key is:
    - FOREIGN KEY (other\_tables\_column) REFERENCES other\_tables\_name(current\_tables\_column)
  + By creating this foreign key, sql will not allow us to create an album entry if it’s band\_id does not correspond to an existing band’s id.
  + If we try to delete a band entry that has albums linking to that band, it will throw an error. In order to delete that band entry, we must first delete the corresponding albums.
  + Now, if we go the schema UI tab on the left in MySQL workbench, we see the following:
  + Graphical user interface, text, application, Word

    Description automatically generated
  + Notice how there is a foreign key inside the albums table.

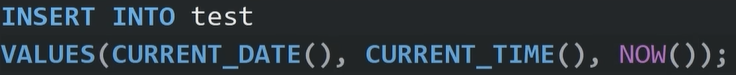
Adding entries to tables

* + 
  + In this above example, we are adding to the “albums” table. We will indicate that the in the following values, the first, second, and third parameters represent the “name”, “band\_id”, and “release\_year” respectively. Thus, we added an album entry with a name of “The Number of the Beasts”, a band\_id of 1, and a release\_year of 1985).
  + Notice the order of the columns “name, band\_id, release\_year” does not have the be same order as when the table was created.
  + We can add multiple entries at a time as shown below:
  + Text

    Description automatically generated with low confidence
  + Notice that the first two albums correspond to the same band. Notice that the last two albums have the same name and release year. Notice how the second album doesn’t have a release year.
  + The id of these albums are automatically created. Since the “The Number of the Beasts” album is created first, then “Power Slave”, then “Nightmare”, then the other “Nightmare”, then the id of the “The Number of the Beasts” is less than the id of “Power Slave” which is less than the id of “Nightmare” which is less than the id of “Nightmare”.

Current\_date() and current\_time()

* + Text

    Description automatically generated
  + 
  + 
  + Output: 
  + Note that CURRENT\_DATE()+1 refers to tomorrow and CURRENT\_DATE()-1 refers to yesterday. A similar thing can be done to the the CURRENT\_TIME() and NOW() functions as we can add/subtract seconds.

Select

* + To query data from our table, we will use the select keyword. We follow the select keyword with the names of the columns we want to select. If we want to select all the columns, then we use \*. We also need to indicate which table we are selecting from.
  + 
  + In this example, we are selecting all the columns from the bands table.
  + Graphical user interface, table

    Description automatically generated
  + 
  + In this example, we are selecting the “name” column from the bands table.
  + Graphical user interface, text, application

    Description automatically generated

Limit

* + If we want to limit the number of entries returned from a select, we can use the limit keYword.
  + 
  + In this example, we are selecting all the columns from the bands table and returning only the first two entries.
  + Graphical user interface, text, application

    Description automatically generated

As/Alias

* + When we select a column, we can rename the column so that it is easier to read
  + 
  + In this example, we are selecting the “id” column and “name” columns from the bands table and renaming them to “ID” and “BAND Name” respectively.
  + Graphical user interface, application

    Description automatically generated

Order By

* + We can order the way our entries are selected
  + 
  + Output: Graphical user interface, text, application, table

    Description automatically generated
  + In this example, we are getting all the columns from the bands table and ordering the entries by name. By default, the order is ascending.
  + If we want the order of the names to be descending, we can write the following script:
  + 
  + Output: Table

    Description automatically generated with medium confidence

Distinct Select

* + Suppose when we run this  script, we get the following output:
  + Graphical user interface, text, application

    Description automatically generated
  + Notice “nightmare” appears twice.
  + To display only the unique names, we run this command 
  + Output: Graphical user interface, text, application

    Description automatically generated

Update

* + To update an entry we can use the set keyword and filter for a specific entry to apply the update to.
  + Chart

    Description automatically generated
  + In this example, we are updating release\_year to 1982 for all the entries in the albums table that have an id of 1.
  + Note that if there was no “where” and the command was just , we would update the release\_year to 1982 for all album entries.

Filtering

* + 
  + Output: Graphical user interface, application, table

    Description automatically generated
  + In this example, we are filtering for the album entries with a release year less than 2000

Like

* + 
  + Output: Graphical user interface, application

    Description automatically generated
  + In this example, we are filtering for the album entries with a name that matches the “%er%” pattern. The % represents an arbitrary pattern of characters (including the pattern that consist of no characters, ie empty string).
  + Note that album names like “er”, “aer”, “era”, “aera” would all satisfy the above condition.
  + Another wildcard character is “\_” which represents any one character.

Or

* + 
  + Output: Table

    Description automatically generated
  + In this example, we are filtering for the album entries with a name that matches the “%er%” pattern or have a band\_id = 2.
  + Notice how the “nightmare” album doesn’t match the “%er%” pattern but has a band\_id of 2 so it’s displayed.

And

* + 
  + Output: Table

    Description automatically generated with medium confidence
  + In this example, we are filtering for the album entries with a release year of 1984 and have a band\_id = 2.

Between

* + 
  + Output: Graphical user interface, table

    Description automatically generated
  + In this example, we are filtering for the album entries with a release year of between 2000 and 2018.
  + The BETWEEN operator is inclusive: begin and end values are included.

Is Null

* + 
  + Table, Teams

    Description automatically generated
  + In this example, we are filtering for the album entries with a release year of NULL.

Subqueries

* + A subquery is a query nested inside another query in SQL. It is used to retrieve data from one or more tables, and then use that data as a temporary table to perform additional operations.
  + There are two types of subqueries in SQL:
    - Scalar subqueries - a subquery that returns a single value, which can be used in place of a literal value in a query. For example: SELECT name, age, (SELECT MAX(age) FROM users) AS max\_age FROM users
    - Table subqueries - a subquery that returns a table, which can be used in place of a table name in a query. For example: SELECT \* FROM users WHERE age IN (SELECT age FROM seniors)
  + Subqueries can be used in various ways, including in the WHERE clause, HAVING clause, SELECT statement, and JOIN clause. They can also be nested multiple times to perform complex queries.
    - Some common use cases for subqueries include:
    - Filtering data based on a condition from another table or subquery
    - Calculating aggregate functions on subsets of data
    - Checking for the existence of data in another table or subquery
    - Joining tables based on a condition from a subquery.
  + When using subqueries, it's important to optimize the query performance by avoiding unnecessary subqueries and using appropriate indexing.
  + Ex:
    - Let's say we have two tables - "students" and "grades". The "students" table has columns for student\_id, name, and age. The "grades" table has columns for student\_id, subject, and score.
    - We want to find the names of all students who have a score greater than the average score for their subject.
    - The SQL query using a subquery to accomplish this would be:
    - Text

      Description automatically generated
    - In this example, the subquery (SELECT AVG(score) FROM grades WHERE subject = 'Math') calculates the average score for the subject 'Math'. The outer query then uses this value to find all students who have a score greater than this average for the subject 'Math'. The IN keyword is used to match the student\_id values returned by the subquery with the student\_id values in the students table.

Indexes

* + When we want to search for a specific entry via a column, we sequentially search through the column from top to bottom. The longer the column, the more expensive the operation.
  + In SQL, indexes are used to improve the performance of queries by allowing the database to quickly locate the data needed to satisfy a particular query. An index is a data structure that is built on one or more columns of a table and contains a sorted copy of the data in those columns.
  + There are several types of indexes that can be used in SQL, including:
    - B-tree index: This is the most common type of index in SQL, and is used for simple value lookups and range queries.
    - Hash index: This type of index is used for equality lookups, where the value being searched for is an exact match.
    - Bitmap index: This type of index is used for tables with a large number of rows and a small number of distinct values in a particular column.
  + It's important to note that creating too many indexes on a table can actually decrease performance, as each additional index must be maintained and updated whenever the table is modified. Therefore, it's important to carefully consider which columns to index and how many indexes to create for a given table. Additionally, indexes should be periodically monitored and potentially rebuilt to ensure optimal performance.
  + By using indexes, updating takes more time, selecting takes less time. Thus, if a table is constantly being updated, it wouldn’t be a good candidate for an index. But if the table is being searched often, it would be a good candidate for an index.
  + Showing indexes
    - To show the indexes in a table, run: SHOW INDEXES FROM table\_name
    - Output:
  + Creating an index
    - To create an index, run: CREATE INDEX index\_name ON table\_name(table\_column)
  + Creating a multi-column index
    - Multi-column indexes allow us to search for an entry based on its values in multiple columns
    - Run: CREATE INDEX index\_name ON table\_name(column1, column2)
    - MySQL has a left-most-prefix. Thus, column1 is searched before column2.
  + Dropping an index
    - To delete an index, run: ALTER TABLE table\_name DROP INDEX index\_name
  + Ex:
    - Text

      Description automatically generated
    - 
    - Output: 
    - 
    - 
    - Output: Graphical user interface

      Description automatically generated
    - Notice that there are two indexes (primary key and last\_name\_idx) which are represented by the two rows.
    - Now, if we search for a customer by last name it would be faster. Ex: Text

      Description automatically generated
    - 
    - 
    - Output: Graphical user interface, text, application

      Description automatically generated
    - Thus, if we had this  query, it would benefit from the multi-column index. This is because there is a sequence in the query where the last\_name is the first query and then the first\_name is the second query. This matches the order of the columns when we created the multi-column index.
    - Note that if we had this  query, it would benefit from the multi-column index even though there is no first\_name query.
    - Note that if we this query, it would not benefit from the multi-column index since there first column to be searched is the first\_name column, but the we made the index search the last\_column first.

Delete an entry

* + To update an entry we can use the set keyword and filter for a specific entry to apply the update to.
  + 
  + In this example, we deleting all the entries in the albums table that have an id of 4.
  + Note that if there was no “where” and the command was just , we would delete all album entries.

On Delete

* + the ON DELETE clause is used in a foreign key definition to specify what action should be taken when a referenced row in the parent table is deleted.
  + the ON DELETE clause is used in conjunction with the FOREIGN KEY constraint, which is used to enforce referential integrity between tables.
  + There are several options for the ON DELETE clause, including:
    - CASCADE: This option deletes all the rows in the child table that reference the deleted row in the parent table. This can result in the deletion of multiple rows in one or more child tables.
    - SET NULL: This option sets the foreign key value in the child table to NULL when the referenced row in the parent table is deleted.
    - SET DEFAULT: This option sets the foreign key value in the child table to the default value specified for the column when the referenced row in the parent table is deleted.
    - RESTRICT: This option prevents the deletion of the row in the parent table if there are any rows in the child table that reference it.
  + Ex:
    - Let's consider two tables “orders” and “customers”. The orders table has a foreign key constraint on the customer\_id column that references the customer\_id column in the customers table. We want to delete all the rows in the orders table that reference the deleted row in the customers table.
    - Here's how we can create the orders table with the foreign key constraint:
    - Graphical user interface

      Description automatically generated with low confidence
    - In this example, the ON DELETE CASCADE option is specified in the foreign key constraint. This means that if a row in the customers table is deleted, all the rows in the orders table that reference the deleted row will also be deleted.
    - If we have the following data in the orders table:
    - Text

      Description automatically generated
    - And if we delete a row in the customers table where customer\_id = 1234, all the rows in the orders table that reference the deleted row will also be deleted, like this:
    - 
    - A screenshot of a computer

      Description automatically generated with low confidence
    - Note that the rows with order\_id = 1 and order\_id = 3 are deleted because they were referencing the deleted row in the customers table where customer\_id = 1234. The row with order\_id = 2 is still in the orders table because it was not referencing the deleted row.
  + Ex:
    - Let's consider two tables “orders” and “customers”. The orders table has a foreign key constraint on the customer\_id column that references the customer\_id column in the customers table. We want to set the customer\_id column in the orders table to NULL if a referenced row is deleted from the customers table.
    - Here's how we can create the orders table with the foreign key constraint:
    - Graphical user interface

      Description automatically generated with medium confidence
    - In this example, the ON DELETE SET NULL option is specified in the foreign key constraint. This means that if a row in the customers table is deleted, the corresponding customer\_id values in the orders table will be set to NULL instead of causing an error or deleting the rows in the child table.
    - If we have the following data in the orders table
    - Text

      Description automatically generated
    - And if we delete a row in the customers table where customer\_id = 5678, the customer\_id value in the orders table where order\_id = 2 will be set to NULL, like this:
    - 
    - Text

      Description automatically generated
    - Note that the customer\_id value in the row with order\_id = 2 is now NULL because it was referencing the deleted row in the customers table.

Begin End

* + In SQL, a BEGIN...END block is a way to group multiple SQL statements together into a single logical unit or transaction. The BEGIN keyword marks the beginning of the block, and the END keyword marks the end of the block. Any SQL statements between the BEGIN and END keywords are considered to be part of the block and will be executed together as a single unit.
  + Here's an example of a BEGIN...END block in SQL:
  + Text

    Description automatically generated
  + In this example, we are using a BEGIN...END block to group together three SQL statements that will be executed together as a single transaction. The first statement is an INSERT statement that inserts a new row into the my\_table table. The second statement is an UPDATE statement that updates an existing row in the my\_table table. The third statement is a DELETE statement that deletes one or more rows from the my\_table table.
  + By using a BEGIN...END block to group these statements together, we can ensure that they are executed together as a single logical unit or transaction. If any of the statements fail, the entire transaction will be rolled back and none of the changes will be committed to the database.
  + BEGIN...END blocks are commonly used in SQL stored procedures and triggers, where multiple SQL statements need to be executed together as a single unit of work.

Triggers

* + In SQL, a trigger is a special type of stored procedure that automatically executes in response to certain events, such as an insert, update, or delete operation on a table. Triggers are commonly used to enforce business rules, maintain referential integrity, and audit changes to data.
  + A trigger consists of three main parts: the trigger event, the trigger condition, and the trigger action. The trigger event specifies the type of operation that will trigger the execution of the trigger, such as an INSERT or UPDATE operation. The trigger condition specifies the criteria that must be met for the trigger action to be executed, such as a certain column value being changed. The trigger action is the set of SQL statements that will be executed when the trigger is activated.
  + Here's an example of creating a trigger in SQL:
  + Text

    Description automatically generated
  + In this example, we are creating a trigger called my\_trigger that will execute after an insert operation is performed on the my\_table table. The FOR EACH ROW clause indicates that the trigger action will be executed once for each row that is affected by the insert operation.
  + The trigger action in this example consists of an SQL INSERT statement that inserts a new row into an audit table my\_audit\_table with information about the event that triggered the trigger, the table that was affected, the row ID that was inserted, and the user who performed the insert operation.
  + Triggers can be useful for a variety of purposes, such as enforcing data integrity constraints, implementing complex business rules, and auditing changes to data. However, triggers should be used judiciously and with care, as they can affect database performance and introduce potential points of failure if not designed and tested properly.

Union

* + In SQL, the UNION operator is used to combine the result sets of two or more SELECT statements into a single result set. The result set returned by the UNION operator contains all the rows that are returned by the SELECT statements that are used in the UNION operation.
  + The basic syntax of the UNION operator is as follows:
  + Text

    Description automatically generated with medium confidence
  + Note that the number of columns in the select statements must match.
  + The UNION operator eliminates duplicate rows from the result set. If you want to include duplicates, you can use the UNION ALL operator instead.
  + Ex:
    - Text

      Description automatically generated with medium confidence
    - Graphical user interface, application

      Description automatically generated
    - Note that there are two albums named “The Number of the Beasts”, but only one is shown since UNION removes duplicates.

Join

* + The “join” keyword allows us to join two related tables together.
  + Ex:
    - Suppose our bands table looks like this: Graphical user interface, application, table

      Description automatically generated
    - Suppose our albums table looks like this: Table

      Description automatically generated
    - If we run this command: ,
    - the output is: Graphical user interface, text, application

      Description automatically generated
    - In this example, we are joining the “albums” table onto the “bands” table. We need to tell sql how we want to join these tables since all the “join” keyword does is it checks a query. The query we provide is bands.id = albums.band\_id.
    - Thus, this command selects all the columns from the “bands table”, then joins the “albums” table onto the “bands” table if the bands.id = albums.band\_id.
    - Notice there are two “Iron Maiden” rows since there are two albums that correspond to the “Iron Maiden” band.
    - Notice we say bands.id since if we just say id, it is ambiguous as to if it’s the “bands” table’s “id” property or the “albums” table’s “id”. Saying albums.band\_id is not necessary as only the “albums” table has a “band\_id” property but it shows it can be done regardless.
  + The example we just say makes use of the “join” keyword which is an inner join. Thus, the command in the example and this are equivalent.
  + An inner join only returns entries that have a match. So if there is a band with no albums, the band wouldn’t be displayed since there is no matching table entry such that bands.id = albums.band\_id.
  + In addition to an inner join, there is also a left and right join.

Left Join

* + A left join works like an inner join except it displays all the entries from the table on the left side of the join (the table that is being joined onto) regardless of if there is a match or not.
  + Ex:
    - 
    - Output:Graphical user interface, text, application, table

      Description automatically generated with medium confidence
    - In this example, the “bands” table is on the left side of the join (it comes before the “joins” keyword). Thus, all the entries in the “bands” table are displayed.
    - Notice that even though there is no album entry with an id equal to the “Ankor” band’s id of 4, the “Ankor” band is still displayed since this is a left join.

Right Join

* + A right join works like an inner join except it displays all the entries from the table on the right side of the join (the table that is joining the other table) regardless of if there is a match or not.
  + Ex:
    - Logo

      Description automatically generated with medium confidence
    - Output: Graphical user interface, text, application

      Description automatically generated
    - In our specific example, all albums must have a corresponding band so the right join is the same as the inner join.
  + Note that a right join is basically the same as a left join if swap the order of the tables in the left join. However, the order of the columns will be swapped. Thus, we rarely use right joins, just inner and left joins.
  + In other words, these two commands are the same:
    - 
    - Output: Graphical user interface, text, application

      Description automatically generated
    - 
    - Output: Table

      Description automatically generated

Self Join

* + A self join is when we join another copy of a table to itself.
  + Suppose we have the following “customers” table.
  + Table

    Description automatically generated
  + Note that referral\_id represents the id of the customer that referred the current entry’s customer. For example, Bubble Bass and Poppy Puff were both referred by Larry Lobster.
  + Now if the run the following command;
  + Text

    Description automatically generated
  + Notice we have to use table alias to distinguish the table that is being joined to and the table that is joining onto.
  + The following is ouput:
  + Table

    Description automatically generated
  + Ex:
    - Table

      Description automatically generated
    - Note that in this example, the manager\_id is a foreign key that references another entry within the same table, as a manager is still an employee.

Aggregate Functions

* + There are many aggregate functions such as avg, sum, count, etc.
  + 
  + Output: 
  + In this example, we are calculating the average of the release years from all the entries within the “albums” table.
  + 
  + Output: 
  + In this example, we are counting the number of names within the “albums” table.
  + Note that count does not count the distinct names, but the total number of names.
  + To count the number of entries in a table, we can use the following SQL statement:
  + SELECT COUNT(\*) FROM TABLE\_NAME

Group By

* + Group by is a SQL statement that groups rows based on a specified column and allows you to perform aggregate functions (such as SUM, COUNT, AVG, etc.) on those groups.
  + The basic syntax of the GROUP BY statement is:
  + Text

    Description automatically generated
  + Ex:
    - To show what the table looks like: Table

      Description automatically generated
    - 
    - Output: Graphical user interface, application, table

      Description automatically generated
    - This command makes the first and second columns be “name” and “SUM(release\_year)”.
    - This command groups the entries by name. The distinct names in the albums table are “The Number of the Beasts”, “Power Slave”, and “Nightmare”. Thus, these are the three rows for the returned table. The SUM aggregate function is applied to each group. Thus, for all the album entries with the same name, it sums the release years of those entries. In the example, there are 2 album entries with the name of “Nightmare”, and their release years are 2018 and 2010 so the sum of their release years is 4028.

Table Alias

* + You can use aliasing to give a table or column a temporary name to make your SQL queries more readable and concise. To alias a table, you can use the AS keyword followed by the desired alias name.
  + Consider the following command:
  + 
  + We can use a to replace albums and b to replace bands as follows to write an equivalent command:
  + 

Having vs Where

* + This command displays the number of albums each band has:
  + Text

    Description automatically generated with medium confidence
  + Output: Graphical user interface, application

    Description automatically generated
  + Suppose we want to filter this table based on the aggregate.
  + We can’t use the ‘where’ keyword since “where” statements happen before the groupby. Thus, we can’t actually filter by the aggregate since the aggregate happens after the groupby.
  + To solve this issue, we use the “having” keyword. This “having” keyword is the same as the “where” statement except the “having” keyword happens after the groupby.
  + For example, we want to only show the rows that have a count of 1.
  + Chart

    Description automatically generated
  + Output: Graphical user interface

    Description automatically generated with medium confidence

Views

* + A view is a virtual table. They are not real. They are made up from other real tables in the database. Views can be interacted as if they were real.
  + Note that any changes made to a real able will be updated in the view.
  + The reason we may want to use views instead of another real table is because
    - The view and the original real table have a lot of common data, so we don’t want to store repeated info
    - If we want to remove an entry that is shared between the view and the original real table, we have to remove it in two tables as opposed to one
  + Here's an example of how to create a view in SQL:
  + A picture containing text

    Description automatically generated
  + In this example, we are creating a view called my\_view based on the result set of a SELECT statement. The SELECT statement specifies the columns we want to include in the view and the criteria for filtering the data. Once the view is created, we can use it like a regular table in SQL queries:
  + 
  + This query will return the same results as if we had executed the original SELECT statement used to create the view.

Stored procedures

* + Stored procedures are basically functions that allow you to reuse commands that you save.
  + A stored procedure is a set of SQL statements that can be executed as a single unit. It is stored in a database and can be called by other programs or procedures to perform a specific task. Stored procedures provide a way to encapsulate complex database operations, simplify code maintenance, and improve database performance.
  + Stored procedures are good at:
    - Reducing network traffic
    - Increasing performance
    - Secure, admin can grant permission to use
  + However, stored procedures increase the memory usage of every connection
  + Ex:
    - Text

      Description automatically generated with medium confidence
    - In this example, we create a procedure called find\_customers(). The commands within the BEGIN and END are the commands that will be executed when we call find\_customers. Notice we have to temporarily change the delimiter from ‘;’ to “$$”. This is so the ‘;’ within the BEGIN and END blocks don’t interfere with the command of creating the procedure. Notice we have an “id” parameter passed to the procedure.
    - To invoke the procedure run: 
    - Output: 
    - To drop a procedure run: 

Autocommit, commit, rollback

* + In SQL, autocommit, commit, and rollback are commands used to manage database transactions.
    - autocommit: When autocommit is enabled, every individual SQL statement is treated as a transaction and is automatically committed as soon as it is executed. This means that each SQL statement is executed as a separate transaction and cannot be rolled back. Autocommit is typically the default setting in most database systems.
    - commit: The commit command is used to explicitly commit a transaction and make the changes permanent in the database. When a transaction is committed, all the changes made to the database during the transaction are saved permanently. After a commit, a new transaction begins.
    - rollback: The rollback command is used to undo a transaction and restore the database to the state it was in before the transaction began. When a transaction is rolled back, all the changes made to the database during the transaction are undone and the database is returned to its previous state.
  + Here's an example of how these commands can be used in a transaction:
    - Text

      Description automatically generated
    - In this example, we begin a transaction with BEGIN TRANSACTION. We then perform some SQL operations, including an INSERT and an UPDATE. If these operations are successful, we use the COMMIT command to commit the changes to the database. If there is an error or issue with the transaction, we use the ROLLBACK command to undo the changes and return the database to its previous state.
    - It's important to use transactions and commit/rollback commands properly to ensure data integrity and consistency in a database system.
  + Ex:
    - 
    - By turning autocommit off, we can manually make commits to make checkpoints
    - 
    - We make a checkpoint by executing the command “commit”
    - 
    - Output: Table

      Description automatically generated
    - If we select all the columns from our albums, we see the above.
    - 
    - Suppose we accidentally deleted all the entries from our albums as shown above.
    - 
    - Since we haven’t committed the “delete” command, we can still rollback to our previous “commit”. This essentially undos the “delete” command.
    - 
    - Output: Table

      Description automatically generated
    - If we select all the columns from our albums, we see the above. This is the same as before we did the “delete command”.

ER Diagrams

* + <https://youtu.be/HXV3zeQKqGY?t=13334>

One to Many Relationships

* + In database design, a one-to-many relationship is a type of relationship between two tables, where each record in the first table can have multiple corresponding records in the second table, but each record in the second table can only have one corresponding record in the first table. This is also known as a "parent-child" relationship or a "master-detail" relationship.
  + To establish a one-to-many relationship, a foreign key is added to the child table that references the primary key of the parent table. This ensures that each record in the child table is associated with a single record in the parent table.
  + A screenshot of a computer

    Description automatically generated with low confidence
  + This example shows how a customer can have many orders but all orders are associated with only 1 customer.

Many to Many Relationships

* + In database design, a many-to-many relationship is a type of relationship between two tables, where each record in the first table can be associated with multiple records in the second table, and each record in the second table can be associated with multiple records in the first table. This type of relationship requires an intermediate table, known as a junction or linking table, to connect the two tables.
  + A screenshot of a computer

    Description automatically generated with low confidence
  + This example shows how a student can have many courses and a course can have many students.
  + Ex:
    - Table

      Description automatically generated
    - Suppose we have the above two tables, how do we show a relationship between the two tables?
    - To solve the above issue, we make use of a joining table. Sometimes it is called a junction/bridging table.
    - This joining table, which in our case is the student\_class table, captures the relationship between students and classes.
    - Graphical user interface, application, table

      Description automatically generated with medium confidence
    - Notice how we to represent a many-to-many relationship, we have two one-to-many relationships.
    - Table

      Description automatically generated
    - Note that we can add more columns to the joining table.
    - Table

      Description automatically generated
    - When naming joining tables, it is possible to name them as table1name\_table2name as we did in our example. But sometimes we should try having a name that better describes what is being stored, such as “enrolment” or “class\_enrolment” in our case.

Connecting MySQL With Python

* + <https://www.youtube.com/watch?v=MhaH7o3lf4E>
  + <https://dev.mysql.com/doc/connector-python/en/>
  + Hosting with aws: <https://www.youtube.com/watch?v=v3jH1YxJqaY>