# Section III - SELECT

### 3.1: Selecting all columns

SELECT \* FROM CUSTOMER;

To limit the number of records returned, use a LIMIT. To limit the results to just 2 records:

SELECT \* FROM CUSTOMER LIMIT 2;

### 3.2: Selecting specific columns

SELECT CUSTOMER\_ID, NAME FROM CUSTOMER;

### 3.3: Expressions

First, select everything from PRODUCT

SELECT \* FROM PRODUCT;

You can use expressions by declaring a TAXED\_PRICE. This is not a column, but rather something that is calculated every time this query is executed.

SELECT PRODUCT\_ID,  
DESCRIPTION,  
PRICE,  
PRICE \* 1.07 AS TAXED\_PRICE  
FROM PRODUCT;

In SQliteStudio, you can hit CTRL + SPACE on Windows and Linux to show an autocomplete box with available fields. For Mac, you will need to enable that configuration in preferences.

You can also use aliases to declare an UNTAXED\_PRICE column off the PRICE, without any expression.

SELECT PRODUCT\_ID,  
DESCRIPTION,  
PRICE as UNTAXED\_PRICE,  
PRICE \* 1.07 AS TAXED\_PRICE  
FROM PRODUCT;

**SWITCH TO SLIDES** FOR MATHEMATICAL OPERATORS

### 3.4: Using round() Function

SELECT PRODUCT\_ID,  
DESCRIPTION,  
PRICE,  
round(PRICE \* 1.07, 2) AS TAXED\_PRICE  
  
FROM PRODUCT;

### 3.5: Text Concatenation

You can slap a dollar sign to our result using concatenation.

SELECT PRODUCT\_ID,  
DESCRIPTION,  
PRICE AS UNTAXED\_PRICE,  
'$' || round(PRICE \* 1.07, 2) AS TAXED\_PRICE  
FROM PRODUCT

You can merge text via concatenation. For instance, you can concatenate two fields and put a comma and space , in between.

SELECT NAME,  
CITY || ', ' || STATE AS LOCATION  
FROM CUSTOMER;

You can concatenate several fields to create an address.

SELECT NAME,  
STREET\_ADDRESS || ' ' || CITY || ', ' || STATE || ' ' || ZIP AS SHIP\_ADDRESS  
FROM CUSTOMER;

This works with any data types, like numbers, texts, and dates. Also note that some platforms use concat() function instead of double pipes ||

**SWITCH TO SLIDES** FOR EXERCISE

## 3.6: Comments

To make a comments in SQL, use commenting dashes or blocks:

-- this is a comment  
  
/\*  
This is a  
multiline comment  
\*/

## Section IV- WHERE

### 4.1: Getting year 2010 records

SELECT \* FROM station\_data  
WHERE year = 2010;

### 4.2: Getting non-2010 records

SELECT \* FROM station\_data  
WHERE year != 2010;

SELECT \* FROM station\_data  
WHERE year <> 2010;

### 4.3: Getting records between 2005 and 2010

SELECT \* FROM station\_data  
WHERE year BETWEEN 2005 AND 2010

### 4.4: Using AND

SELECT \* FROM station\_data  
WHERE year >= 2005 AND year <= 2010

### 4.5: Exclusive Range

This will get the years between 2005 and 2010, but exclude 2005 and 2010

SELECT \* FROM station\_data  
WHERE year > 2005 AND year < 2010

### 4.6: Using OR

SELECT \* FROM station\_data  
WHERE MONTH = 3  
OR MONTH = 6  
OR MONTH = 9  
OR MONTH = 12

### 4.7: Using IN

SELECT \* FROM station\_data  
WHERE MONTH IN (3,6,9,12);

### 4.8: Using NOT IN

SELECT \* FROM station\_data  
WHERE MONTH NOT IN (3,6,9,12);

### 4.9: Using Modulus

The modulus will perform division but return the remainder. So a remainder of 0 means the two numbers divide evenly.

SELECT \* FROM station\_data  
WHERE MONTH % 3 = 0;

### 4.10: Using WHERE on TEXT

SELECT \* FROM station\_data  
WHERE report\_code = '513A63'

### 4.11: Using IN with text

SELECT \* FROM station\_data  
WHERE report\_code IN ('513A63','1F8A7B','EF616A')

### 4.12: Using length() function

SELECT \* FROM station\_data  
WHERE length(report\_code) != 6

### 4.13A: Using LIKE for any characters

SELECT \* FROM station\_data  
WHERE report\_code LIKE 'A%';

### 4.13B: Using Regular Expressions

If you are familiar with regular expressions, you can use those to identify and qualify text patterns.

SELECT \* FROM STATION\_DATA  
WHERE report\_code REGEXP '^A.\*$'

### 4.14: Using LIKE for one character

SELECT \* FROM station\_data  
WHERE report\_code LIKE 'B\_C%';

For LIKE, % is used in a different context than modulus %

### 4.15: True Booleans 1

SELECT \* FROM station\_data  
WHERE tornado = 1 AND hail = 1;

### 4.16: True Booleans 2

SELECT \* FROM station\_data  
WHERE tornado AND hail

### 4.17: False Booleans 1

SELECT \* FROM station\_data  
WHERE tornado = 0 AND hail = 1;

### 4.18: False Booleans 2

SELECT \* FROM station\_data  
WHERE NOT tornado AND hail;

### 4.19: Handling NULL

A NULL is an absent value. It is not zero, empty text ' ', or any value. It is blank.

To check for a null value:

SELECT \* FROM station\_data  
WHERE snow\_depth IS NULL;

### 4.20: Handling NULL in conditions

Nulls will not qualify with any condition that doesn't explicitly handle it.

SELECT \* FROM station\_data  
WHERE precipitation <= 0.5;

If you want to include nulls, do this:

SELECT \* FROM station\_data  
WHERE precipitation IS NULL OR precipitation <= 0.5;

You can also use a coalesce() function to turn a null value into a default value, if it indeed is null.

This will treat all null values as a 0.

SELECT \* FROM station\_data  
WHERE coalesce(precipitation, 0) <= 0.5;

### 4.21: Combining AND and OR

Querying for sleet or snow

Problematic. What belongs to the AND and what belongs to the OR?

SELECT \* FROM station\_data  
WHERE rain = 1 AND temperature <= 32  
OR snow\_depth > 0;

You must group up the sleet condition in parenthesis so it is treated as one unit.

SELECT \* FROM station\_data  
WHERE (rain = 1 AND temperature <= 32)  
OR snow\_depth > 0;

# Section V- GROUP BY and ORDER BY

### 5.1: Getting a count of records

SELECT count(\*) as record\_count FROM station\_data

### 5.2 Getting a count of records with a condition

SELECT count(\*) as record\_count FROM station\_data  
WHERE tornado = 1

### 5.3 Getting a count by year

SELECT year, count(\*) as record\_count  
FROM station\_data  
WHERE tornado = 1  
GROUP BY year

### 5.4 Getting a count by year, month

SELECT year, month, count(\*) as record\_count  
FROM station\_data  
WHERE tornado = 1  
GROUP BY year, month

### 5.5 Getting a count by year, month with ordinal index

SELECT year, month, count(\*) as record\_count  
FROM station\_data  
WHERE tornado = 1  
GROUP BY 1, 2

### 5.6 Using ORDER BY

SELECT year, month, count(\*) as record\_count  
FROM station\_data  
WHERE tornado = 1  
GROUP BY year, month  
ORDER BY year, month

### 5.7 Using ORDER BY with DESC

SELECT year, month, count(\*) as record\_count  
FROM station\_data  
WHERE tornado = 1  
GROUP BY year, month  
ORDER BY year DESC, month

### 5.8 Counting non-null values

SELECT COUNT(snow\_depth) as recorded\_snow\_depth\_count  
FROM station\_data

### 5.9 Average temperature by month since year 2000

SELECT month, AVG(temperature) as avg\_temp  
FROM station\_data  
WHERE year >= 2000  
GROUP BY month

### 5.10 Average temperature (with rounding) by month since year 2000

SELECT month, round(AVG(temperature),2) as avg\_temp  
FROM station\_data  
WHERE year >= 2000  
GROUP BY month

### 5.11 Sum of snow depth

SELECT year, SUM(snow\_depth) as total\_snow  
FROM station\_data  
WHERE year >= 2005  
GROUP BY year

### 5.12 Multiple aggregations

SELECT year,  
SUM(snow\_depth) as total\_snow,  
SUM(precipitation) as total\_precipitation,  
MAX(precipitation) as max\_precipitation  
  
FROM station\_data  
WHERE year >= 2005  
GROUP BY year

### EXERCISES

Flip to slides

### 5.13 Using HAVING

You cannot use WHERE on aggregations. This will result in an error.

SELECT year,  
SUM(precipitation) as total\_precipitation  
FROM station\_data  
WHERE total\_precipitation > 30  
GROUP BY year

You can however, use HAVING.

SELECT year,  
SUM(precipitation) as total\_precipitation  
FROM station\_data  
GROUP BY year  
HAVING total\_precipitation > 30

Note that some platforms like Oracle do not support aliasing in GROUP BY and HAVING.

Therefore you have to rewrite the entire expression each time

SELECT year,  
SUM(precipitation) as total\_precipitation  
FROM station\_data  
GROUP BY year  
HAVING SUM(precipitation) > 30

### 5.14 Getting Distinct values

You can get DISTINCT values for one or more columns

SELECT DISTINCT station\_number FROM station\_data

You can also get distinct combinations of values for multiple columns

SELECT DISTINCT station\_number, year FROM station\_data

# Section VI - CASE Statements

### 6.1 Categorizing Wind Speed

You can use a CASE statement to turn a column value into another value based on conditions. For instance, we can turn different wind\_speed ranges into HIGH, MODERATE, and LOW categories.

SELECT report\_code, year, month, day, wind\_speed,  
  
CASE  
 WHEN wind\_speed >= 40 THEN 'HIGH'  
 WHEN wind\_speed >= 30 AND wind\_speed < 40 THEN 'MODERATE'  
 ELSE 'LOW' END  
AS wind\_severity  
  
FROM station\_data

### 6.2 More Efficient Way To Categorize Wind Speed

We can actually omit AND wind\_speed < 40 from the previous example because each WHEN/THEN is evaluated from top-to-bottom. The first one it finds to be true is the one it will go with, and stop evaluating subsequent conditions.

SELECT report\_code, year, month, day, wind\_speed,  
  
CASE  
 WHEN wind\_speed >= 40 THEN 'HIGH'  
 WHEN wind\_speed >= 30 THEN 'MODERATE'  
 ELSE 'LOW' END  
as wind\_severity  
  
FROM station\_data

### 6.3 Using CASE with GROUP BY

We can use GROUP BY in conjunction with a CASE statement to slice data in more ways, such as getting the record count by wind\_severity and year.

SELECT year,  
  
CASE  
 WHEN wind\_speed >= 40 THEN 'HIGH'  
 WHEN wind\_speed >= 30 THEN 'MODERATE'  
 ELSE 'LOW' END  
as wind\_severity,  
  
COUNT(\*) as record\_count  
  
FROM station\_data  
GROUP BY 1,2

### 6.4 "Zero/Null" Case Trick

There is really no way to create multiple aggregations with different conditions unless you know a trick with the CASE statement. If you want to find two total precipitation, with and without tornado precipitations, for each year and month, you have to do separate queries.

**Tornado Precipitation**

SELECT year, month,  
SUM(precipitation) as tornado\_precipitation  
FROM station\_data  
WHERE tornado = 1  
GROUP BY year, month

**Non-Tornado Precipitation**

SELECT year, month,  
SUM(precipitation) as non\_tornado\_precipitation  
FROM station\_data  
WHERE tornado = 0  
GROUP BY year, month

But you can use a single query using a CASE statement that sets a value to 0 if the condition is not met. That way it will not impact the sum.

SELECT year, month,  
SUM(CASE WHEN tornado = 1 THEN precipitation ELSE 0 END) as tornado\_precipitation,  
  
SUM(CASE WHEN tornado = 0 THEN precipitation ELSE 0 END) as non\_tornado\_precipitation  
  
FROM station\_data GROUP BY year, month

### 6.5 Using Null in a CASE to conditionalize MIN/MAX

Since NULL is ignored in SUM, MIN, MAX, and other aggregate functions, you can use it in a CASE statement to conditionally control whether or not a value should be included in that aggregation.

For instance, we can split up max precipitation when a tornado was present vs not present.

SELECT year,  
MAX(CASE WHEN tornado = 0 THEN precipitation ELSE NULL END) as max\_non\_tornado\_precipitation,  
MAX(CASE WHEN tornado = 1 THEN precipitation ELSE NULL END) as max\_tornado\_precipitation  
FROM station\_data  
GROUP BY year

*Switch to slides for exercise*

### Exercise 6.1

SELECT the report\_code, year, quarter, and temperature, where a “quarter” is “Q1”, “Q2”, “Q3”, or “Q4” reflecting months 1-3, 4-6, 7-9, and 10-12 respectively.

**ANSWER:**

SELECT  
  
report\_code,  
year,  
  
CASE  
 WHEN month BETWEEN 1 and 3 THEN 'Q1'  
 WHEN month BETWEEN 4 and 6 THEN 'Q2'  
 WHEN month BETWEEN 7 and 9 THEN 'Q3'  
 WHEN month BETWEEN 10 and 12 THEN 'Q4'  
END as quarter,  
  
temperature  
  
FROM STATION\_DATA

### Exercise 6.2

Get the average temperature by quarter and month, where a “quarter” is “Q1”, “Q2”, “Q3”, or “Q4” reflecting months 1-3, 4-6, 7-9, and 10-12 respectively.

**ANSWER**

SELECT  
year,  
  
CASE  
 WHEN month BETWEEN 1 and 3 THEN 'Q1'  
 WHEN month BETWEEN 4 and 6 THEN 'Q2'  
 WHEN month BETWEEN 7 and 9 THEN 'Q3'  
 WHEN month BETWEEN 10 and 12 THEN 'Q4'  
END as quarter,  
  
AVG(temperature) as avg\_temp  
  
FROM STATION\_DATA  
GROUP BY 1,2

# Section VII - JOIN

### 7.1 INNER JOIN

(Refer to slides Section VII)

View customer address information with each order by joining tables CUSTOMER and CUSTOMER\_ORDER.

SELECT ORDER\_ID,  
CUSTOMER.CUSTOMER\_ID,  
ORDER\_DATE,  
SHIP\_DATE,  
NAME,  
STREET\_ADDRESS,  
CITY,  
STATE,  
ZIP,  
PRODUCT\_ID,  
ORDER\_QTY  
  
FROM CUSTOMER INNER JOIN CUSTOMER\_ORDER  
ON CUSTOMER.CUSTOMER\_ID = CUSTOMER\_ORDER.CUSTOMER\_ID

Joins allow us to keep stored data normalized and simple, but we can get more descriptive views of our data by using joins.

Notice how two customers are omitted since they don't have any orders (refer to slides).

### 7.2 LEFT OUTER JOIN

To include all customers, regardless of whether they have orders, you can use a left outer join via LEFT JOIN (refer to slides).

If any customers do not have any orders, they will get one record where the CUSTOMER\_ORDER fields will be null.

SELECT CUSTOMER.CUSTOMER\_ID,  
NAME,  
STREET\_ADDRESS,  
CITY,  
STATE,  
ZIP,  
ORDER\_DATE,  
SHIP\_DATE,  
ORDER\_ID,  
PRODUCT\_ID,  
ORDER\_QTY  
  
FROM CUSTOMER LEFT JOIN CUSTOMER\_ORDER  
ON CUSTOMER.CUSTOMER\_ID = CUSTOMER\_ORDER.CUSTOMER\_ID

## 7.3 Finding Customers with No Orders

With a left outer join, you can filter for NULL values on the CUSTOMER\_ORDER table to find customers that have no orders.

SELECT CUSTOMER.CUSTOMER\_ID,  
NAME AS CUSTOMER\_NAME  
  
FROM CUSTOMER LEFT JOIN CUSTOMER\_ORDER  
ON CUSTOMER.CUSTOMER\_ID = CUSTOMER\_ORDER.CUSTOMER\_ID  
  
WHERE ORDER\_ID IS NULL

You can use a left outer join to find child records with no parent, or parent records with no children (e.g. a CUSTOMER\_ORDER with no CUSTOMER, or a CUSTOMER with no CUSTOMER\_ORDERs).

## 7.4 Joining Multiple Tables

Bring in PRODUCT to supply product information for each CUSTOMER\_ORDER, on top of CUSTOMER information.

SELECT ORDER\_ID,  
CUSTOMER.CUSTOMER\_ID,  
NAME AS CUSTOMER\_NAME,  
STREET\_ADDRESS,  
CITY,  
STATE,  
ZIP,  
ORDER\_DATE,  
PRODUCT.PRODUCT\_ID,  
DESCRIPTION,  
ORDER\_QTY  
  
FROM CUSTOMER INNER JOIN CUSTOMER\_ORDER  
ON CUSTOMER.CUSTOMER\_ID = CUSTOMER\_ORDER.CUSTOMER\_ID  
  
INNER JOIN PRODUCT  
ON CUSTOMER\_ORDER.PRODUCT\_ID = PRODUCT.PRODUCT\_ID

## 7.7 Using Expressions with JOINs

You can use expressions combining any fields on any of the joined tables. For instance, we can now get the total revenue for each customer.

SELECT ORDER\_ID,  
CUSTOMER.CUSTOMER\_ID,  
NAME AS CUSTOMER\_NAME,  
STREET\_ADDRESS,  
CITY,  
STATE,  
ZIP,  
ORDER\_DATE,  
PRODUCT.PRODUCT\_ID,  
DESCRIPTION,  
ORDER\_QTY,  
ORDER\_QTY \* PRICE as REVENUE  
  
FROM CUSTOMER INNER JOIN CUSTOMER\_ORDER  
ON CUSTOMER.CUSTOMER\_ID = CUSTOMER\_ORDER.CUSTOMER\_ID  
  
INNER JOIN PRODUCT  
ON CUSTOMER\_ORDER.PRODUCT\_ID = PRODUCT.PRODUCT\_ID

## 7.6 Using GROUP BY with JOINs

You can use GROUP BY with a join. For instance, you can find the total revenue for each customer by leveraging all three joined tables, and aggregating the REVENUE expression we created earlier.

SELECT  
CUSTOMER.CUSTOMER\_ID,  
NAME AS CUSTOMER\_NAME,  
sum(ORDER\_QTY \* PRICE) as TOTAL\_REVENUE  
  
FROM CUSTOMER INNER JOIN CUSTOMER\_ORDER  
ON CUSTOMER.CUSTOMER\_ID = CUSTOMER\_ORDER.CUSTOMER\_ID  
  
INNER JOIN PRODUCT  
ON CUSTOMER\_ORDER.PRODUCT\_ID = PRODUCT.PRODUCT\_ID  
  
GROUP BY 1,2

To see all customers even if they had no orders, use a LEFT JOIN

SELECT  
CUSTOMER.CUSTOMER\_ID,  
NAME AS CUSTOMER\_NAME,  
sum(ORDER\_QTY \* PRICE) as TOTAL\_REVENUE  
  
FROM CUSTOMER LEFT JOIN CUSTOMER\_ORDER  
ON CUSTOMER.CUSTOMER\_ID = CUSTOMER\_ORDER.CUSTOMER\_ID  
  
LEFT JOIN PRODUCT  
ON CUSTOMER\_ORDER.PRODUCT\_ID = PRODUCT.PRODUCT\_ID  
  
GROUP BY 1,2

You can also use a coalesce() function to turn null sums into zeros.

SELECT  
CUSTOMER.CUSTOMER\_ID,  
NAME AS CUSTOMER\_NAME,  
coalesce(sum(ORDER\_QTY \* PRICE), 0) as TOTAL\_REVENUE  
  
FROM CUSTOMER LEFT JOIN CUSTOMER\_ORDER  
ON CUSTOMER.CUSTOMER\_ID = CUSTOMER\_ORDER.CUSTOMER\_ID  
  
LEFT JOIN PRODUCT  
ON CUSTOMER\_ORDER.PRODUCT\_ID = PRODUCT.PRODUCT\_ID  
  
GROUP BY 1,2

# Section VIII - Database Design

Refer to slides for database design concepts

To view source code for SQL Injection Demo, here is the GitHub page: https://github.com/thomasnield/sql-injection-demo

To read about normalized forms (which we do not cover in favor of a more intuitive approach), you can read this article:

http://www.dummies.com/programming/sql/sql-first-second-and-third-normal-forms/

## 7.1 - Creating a Table

In SQLiteStudio, navigate to *Database* -> *Add a Database* and click the green plus icon to create a new database. Choose a location and name it surgetech\_conference.db.

Create the COMPANY table. To create a new table, use the SQLiteStudio wizard by right-clicking the surgetech\_conference database and selecting Create a table. You can also just execute the following SQL.

CREATE TABLE COMPANY (  
 COMPANY\_ID INTEGER PRIMARY KEY AUTOINCREMENT,  
 NAME VARCHAR(30) NOT NULL,  
 DESCRIPTION VARCHAR(60),  
 PRIMARY\_CONTACT\_ATTENDEE\_ID INTEGER NOT NULL,  
 FOREIGN KEY (PRIMARY\_CONTACT\_ATTENDEE\_ID) REFERENCES ATTENDEE(ATTENDEE\_ID)  
);

After each field declaration, we create "rules" for that field. For example, COMPANY\_ID must be an INTEGER, it is a PRIMARY KEY, and it will AUTOINCREMENT to automatically generate a consecutive integer ID for each new record. The NAME field holds text because it is VARCHAR (a variable number of characters), and it is limited to 30 characters and cannot be NULL.

Lastly, we declare any FOREIGN KEY constraints, specifying which field is a FOREIGN KEY and what PRIMARY KEY it references. In this example, PRIMARY\_CONTACT\_ATTENDEE\_ID "references" the ATTENDEE\_ID in the ATTENDEE table, and it can only be those values.

## 7.2 - Creating the other tables

Create the other tables using the SQLiteStudio *New table* wizard, or just executing the following SQL code.

CREATE TABLE ROOM (  
 ROOM\_ID INTEGER PRIMARY KEY AUTOINCREMENT,  
 FLOOR\_NUMBER INTEGER NOT NULL,  
 SEAT\_CAPACITY INTEGER NOT NULL  
);  
  
CREATE TABLE PRESENTATION (  
 PRESENTATION\_ID INTEGER PRIMARY KEY AUTOINCREMENT,  
 BOOKED\_COMPANY\_ID INTEGER NOT NULL,  
 BOOKED\_ROOM\_ID INTEGER NOT NULL,  
 START\_TIME TIME,  
 END\_TIME TIME,  
 FOREIGN KEY (BOOKED\_COMPANY\_ID) REFERENCES COMPANY(COMPANY\_ID)  
 FOREIGN KEY (BOOKED\_ROOM\_ID) REFERENCES ROOM(ROOM\_ID)  
);  
  
CREATE TABLE ATTENDEE (  
 ATTENDEE\_ID INTEGER PRIMARY KEY AUTOINCREMENT,  
 FIRST\_NAME VARCHAR (30) NOT NULL,  
 LAST\_NAME VARCHAR (30) NOT NULL,  
 PHONE INTEGER,  
 EMAIL VARCHAR (30),  
 VIP BOOLEAN DEFAULT (0)  
);  
  
CREATE TABLE PRESENTATION\_ATTENDANCE (  
 TICKET\_ID INTEGER PRIMARY KEY AUTOINCREMENT,  
 PRESENTATION\_ID INTEGER,  
 ATTENDEE\_ID INTEGER,  
 FOREIGN KEY (PRESENTATION\_ID) REFERENCES PRESENTATION(PRESENTATION\_ID)  
 FOREIGN KEY (ATTENDEE\_ID) REFERENCES ATTENDEE(ATTENDEE\_ID)  
);

## Creating Views

It is not uncommon to save SELECT queries that are used frequently into a database. These are known as **Views** and act very similarly to tables. You can essentially save a SELECT query and work with it just like a table.

For instance, say we wanted to save this SQL query that includes ROOM and COMPANY info with each PRESENTATION record.

SELECT COMPANY.NAME as BOOKED\_COMPANY,  
ROOM.ROOM\_ID as ROOM\_NUMBER,  
ROOM.FLOOR\_NUMBER as FLOOR,  
ROOM.SEAT\_CAPACITY as SEATS,  
START\_TIME, END\_TIME  
  
FROM PRESENTATION          
  
INNER JOIN COMPANY  
ON PRESENTATION.BOOKED\_COMPANY\_ID = COMPANY.COMPANY\_ID  
  
INNER JOIN ROOM  
ON PRESENTATION.BOOKED\_ROOM\_ID = ROOM.ROOM\_ID

You can save this as a view by right-clicking *Views* in the database navigator, and then *Create a view*. You can then paste the SQL as the body and give the view a name, such as PRESENTATION\_VW (where "VW" means "View").

You can also just execute the following SQL syntax: CREATE [view name] AS [a SELECT query]. For this example, this is what it would look like.

CREATE VIEW PRESENTATION\_VW AS  
  
SELECT COMPANY.NAME as BOOKED\_COMPANY,  
ROOM.ROOM\_ID as ROOM\_NUMBER,  
ROOM.FLOOR\_NUMBER as FLOOR,  
ROOM.SEAT\_CAPACITY as SEATS,  
START\_TIME, END\_TIME  
  
FROM PRESENTATION  
  
INNER JOIN COMPANY  
ON PRESENTATION.BOOKED\_COMPANY\_ID = COMPANY.COMPANY\_ID  
  
INNER JOIN ROOM  
ON PRESENTATION.BOOKED\_ROOM\_ID = ROOM.ROOM\_ID

You will then see the PRESENTATION\_VW in your database navigator, and you can query it just like a table.

SELECT \* FROM PRESENTATION\_VW  
WHERE SEATS >= 30

Obviously, there is no data yet so you will not get any results. But there will be once you populate data into this database.

# Section IX - Writing Data

In this section, we will learn how to write, modify, and delete data in a database.

## 9.1 Using INSERT

To create a new record in a table, use the INSERT command and supply the values for the needed columns.

Put yourself into the ATTENDEE table.

INSERT INTO ATTENDEE (FIRST\_NAME, LAST\_NAME)  
VALUES ('Thomas','Nield')

Notice above that we declare the table we are writing to, which is ATTENDEE. Then we declare the columns we are supplying values for (FIRST\_NAME, LAST\_NAME), followed by the values for this new record ('Thomas','Nield').

Notice we did not have to supply a value for ATTENDEE\_ID as we have set it in the previous section to generate its own value. PHONE, EMAIL, and VIP fields have default values or are nullable, and therefore optional.

## 9.2 Multiple INSERT records

You can insert multiple rows in an INSERT. This will add three people to the ATTENDEE table.

INSERT INTO ATTENDEE (FIRST\_NAME, LAST\_NAME, PHONE, EMAIL, VIP)  
VALUES ('Jon', 'Skeeter', 4802185842,'john.skeeter@rex.net', 1),  
 ('Sam','Scala', 2156783401,'sam.scala@gmail.com', 0),  
 ('Brittany','Fisher', 5932857296,'brittany.fisher@outlook.com', 0)

## 9.3 Testing the foreign keys

Let's test our design and make sure our primary/foreign keys are working.

Try to INSERT a COMPANY with a PRIMARY\_CONTACT\_ATTENDEE\_ID that does not exist in the ATTENDEE table.

INSERT INTO COMPANY (NAME, DESCRIPTION, PRIMARY\_CONTACT\_ATTENDEE\_ID)  
VALUES ('RexApp Solutions','A mobile app delivery service', 5)

Currently, there is no ATTENDEE with an ATTENDEE\_ID of 5, this should error out which is good. It means we kept bad data out.

If you use an ATTENDEE\_ID value that does exist and supply it as a PRIMARY\_CONTACT\_ATTENDEE\_ID, we should be good to go.

INSERT INTO COMPANY (NAME, DESCRIPTION, PRIMARY\_CONTACT\_ATTENDEE\_ID)  
VALUES ('RexApp Solutions', 'A mobile app delivery service', 3)

### 9.3 DELETE records

The DELETE command is dangerously simple. To delete records from both the COMPANY and ATTENDEE tables, execute the following SQL commands.

DELETE FROM COMPANY;  
DELETE FROM ATTENDEE;

Note that the COMPANY table has a foreign key relationship with the ATTENDEE table. Therefore we will have to delete records from COMPANY first before it allows us to delete data from ATTENDEE. Otherwise we will get a "FOREIGN KEY constraint failed effort" due to the COMPANY record we just added which is tied to the ATTENDEE with the ATTENDEE\_ID of 3.

You can also use a WHERE to only delete records that meet a conditional. To delete all ATTENDEE records with no PHONE or EMAIL, you can run this command.

DELETE FROM ATTENDEE  
WHERE PHONE IS NULL AND EMAIL IS NULL

A good practice is to use a SELECT \* in place of the DELETE first. That way you can get a preview of what records will be deleted with that WHERE condition.

SELECT \* FROM ATTENDEE  
WHERE PHONE IS NULL AND EMAIL IS NULL

### UPDATE records

Say we wanted to change the phone number for the ATTENDEE with the ATTENDEE\_ID value of 3, which is Sam Scala. We can do this with an UPDATE statement.

UPDATE ATTENDEE SET PHONE = 4802735872  
WHERE ATTENDEE\_ID = 3

Using a WHERE is important, otherwise it will update all records with the specified SET assignment. This can be handy if you wanted to say, make all EMAIL values uppercase.

UPDATE ATTENDEE SET EMAIL = UPPER(EMAIL)

### 9.4 Dropping Tables

If you want to delete a table, it also is dangerously simple. Be very careful and sure before you delete any table, because it will remove it permanently.

DROP TABLE MY\_UNWANTED\_TABLE

### 9.5 Transactions

Transactions are helpful when you want a series of writes to succeed.

Below, we execute two successful write operations within a transaction.

BEGIN TRANSACTION;  
  
INSERT INTO ROOM (FLOOR\_NUMBER, SEAT\_CAPACITY) VALUES (9, 80);  
INSERT INTO ROOM (FLOOR\_NUMBER, SEAT\_CAPACITY) VALUES (10, 110);  
  
END TRANSACTION;

But if we ever encountered a failure with our write operations, we can call ROLLBACK instead of END TRANSACTION to go back to the database state when BEGIN TRANSACTION was called.

Below, we have a failed operation due to a broken INSERT.

BEGIN TRANSACTION;  
  
INSERT INTO ROOM (FLOOR\_NUMBER, SEAT\_CAPACITY) VALUES (12, 210);  
INSERT INTO ROOM (FLOOR\_NUMBER, SEAT\_CAPACITY) VALUES (13); --failure

So we can call ROLLBACK to "rewind" to the database state when BEGIN TRANSACTION was called.

ROLLBACK;

### 9.6 Creating Indexes

You can create an index on a certain column to speed up SELECT performance, such as the price column on the PRODUCT table.

CREATE INDEX price\_index ON PRODUCT(price);

You can also create an index for a column that has unique values, and it will make a special optimization for that case.

CREATE UNIQUE INDEX name\_index ON CUSTOMER(name);

To remove an index, use the DROP command.

DROP INDEX price\_index;

# Section X - Moving Forward

### SQL Resources

[Getting Started with SQL (O'Reilly)](http://shop.oreilly.com/product/0636920044994.do) by Thomas Nield

[Learning SQL (O'Reilly)](http://shop.oreilly.com/product/9780596520847.do) by Alan Beaulieu

[Using SQLite (O'Reilly)](http://shop.oreilly.com/product/9780596521196.do) by Jay A. Kreibich

[SQL Practice Problems](https://www.amazon.com/SQL-Practice-Problems-learn-doing/dp/1520807635/) by Sylvia Moestl Vasilik