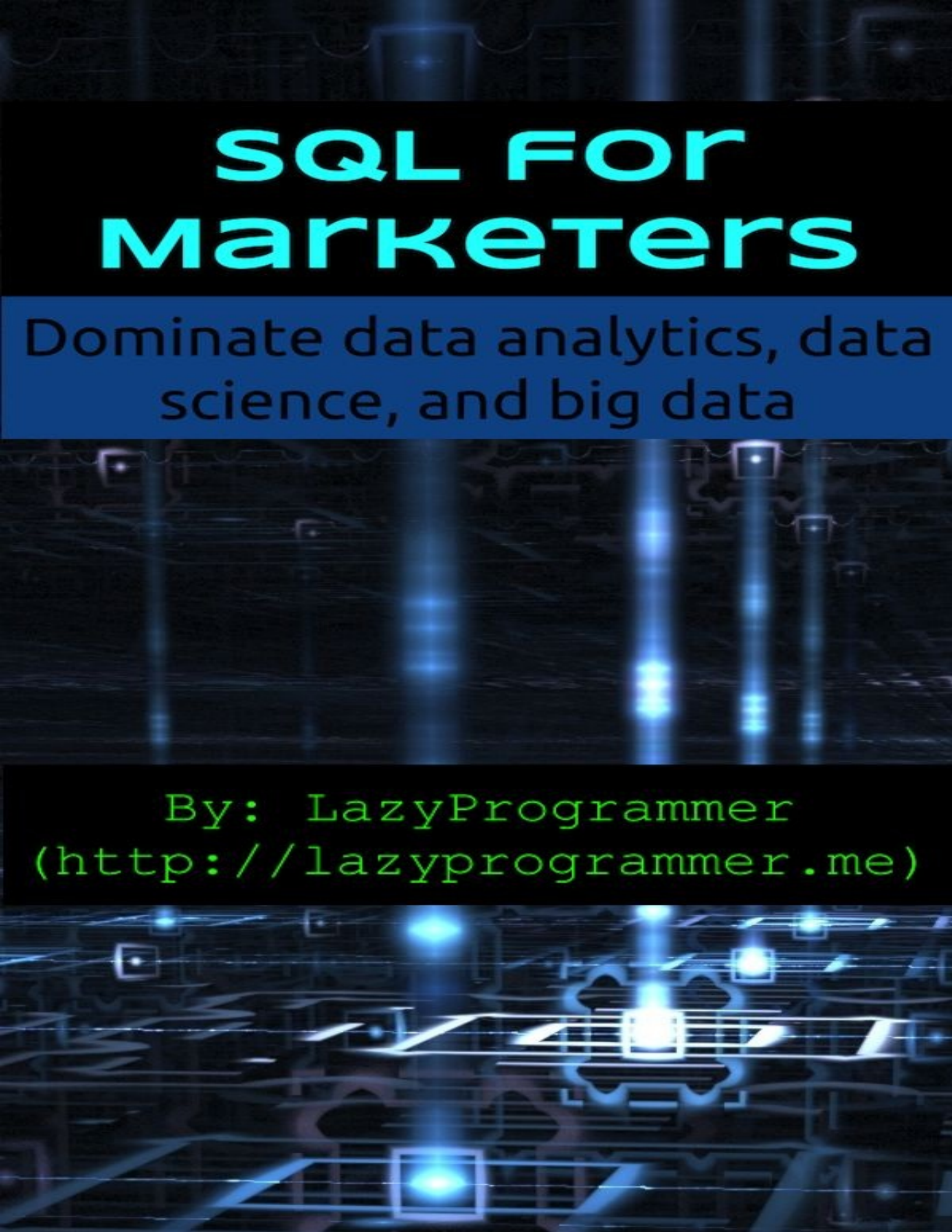
SQL for Marketers
Dominate data analytics, data science, and big data
By: LazyProgrammer
(http://lazyprogrammer.me)

# SQL for Marketers

## Dominate data analytics, data science, and big data

By: The LazyProgrammer (http://lazyprogrammer.me)

# Introduction

More and more companies these days are learning that they need to make DATA-DRIVEN decisions.

With big data and data science on the rise, we have more data than we know what to do with.

One of the basic languages of data analytics is SQL, which is used for many popular databases including MySQL, Postgres, Microsoft SQL Server, Oracle, and even big data solutions like Hive and Cassandra.

I’m going to let you in on a little secret. Most high-level marketers and product managers at big tech companies know how to manipulate data to gain important insights. No longer do you have to wait around the entire day for some software engineer to answer your questions - now you can find the answers directly, by yourself, using SQL!

Do you want to know how to optimize your sales funnel using SQL, look at the seasonal trends in your industry, and run a SQL query on Hadoop? Then join me now in my new book, SQL for marketers: Dominate data analytics, data science, and big data!

# Why marketers and product managers (PMs) need to know SQL

Are you tired of depending on crufty “analytics” software?

Do you have to ask an engineer to help you whenever you have a question about the data?

This is not ideal and won’t help you do your job efficiently.

SQL, short for “structured query language”, is a language that can be used for all kinds of databases - from the tiny databases stored in your iPhone, to large big data databases that span multiple continents.

Engineers have done a great job of creating these different types of complex data stores, while still allowing you to use the same language, more or less, for all of them.

What does that mean for you?

It means as long as you know SQL, you can take advantage of ALL of this software, and gain insights into this data, no matter what kind of database it is stored in, as long as it supports SQL.

You can ask questions like:

- How many people are falling into each stage of the sales funnel?
- What is my year over year profit?
- Are there any differences in the demographics between the people who are buying product X and product Y?
- What is our most profitable month?
- What are the seasonal trends in our industry?

I’m an engineer, so I probably haven’t even thought of all the questions you’ve already had for years! But I guarantee you, knowing SQL will help you answer these questions.

On various teams I’ve worked on in the tech world - I’ve noticed that marketing people and product managers have SQL skills and sometimes even coding skills! So if you are looking to not only make your day more productive, but make yourself more marketable to employers and catch up to the other go-getters in your field - then you should most definitely learn SQL.

# Chapter 1: Overview of SQL databases and Installing SQL

# Overview of SQL Databases

In this section I’m going to go over a number of different technologies that use SQL.

You have have already heard of MySQL. It’s very popular on the web, and if you get hosting from sites like Namecheap, it usually comes with PHP and MySQL pre-installed.

Another similar database is Postgres. Postgres has made a lot of great improvements recently, one being JSON support which allows it to have a lot of functionality similar to MongoDB.

MySQL and Postgres can be run on your local machine, so you can have a PHP application or a Ruby on Rails application running alongside MySQL on the same server. But you can imagine this eats up resources.

So what engineers usually do at scale is to put these databases on their own servers. Now you can have multiple machines running the application code, all talking to the same database.

What happens when data gets really really big? Well then we can't even store it all on one machine. You may have heard of a popular big data technology called Hadoop. The Hadoop file system allows us to do 2 interesting things. 1) is that it splits up our data into chunks. So we could potentially store a 1TB file. It just means we would have more chunks. We can split these chunks across different machines. 2) As you know, machines can fail, and the more machines you have, the higher the chance of failure. So another thing we do is replication. We make multiple copies of these chunks in different places, so that if one machine fails, we still have a copy somewhere else.

Hive is one framework that allows you to use Hadoop as a database, and supports a similar language to SQL called Hive QL. Cassandra is another big database technology that supports a similar language, called Cassandra Query Language or CQL.

At the end of this book we'll look at just plain SQL on a system called Spark which runs on Hadoop.

Throughout the entirety of this book, we're going to use yet another type of database, called SQLite.

## Why SQLite?

SQLite is a file-based database, so it isn’t as large scale as some of the others, but it’s more than enough so support a wide range of scenarios.

SQLite is even on your iPhone! Every app can have its own SQLite database. Many of the apps on your computer do the same thing.

Another great thing about SQLite is it’s super easy to install and use no matter what system you’re on - so if you use Windows, Linux, or Mac, you can do all the exercises in this course.

## Installing SQLite on Mac, Windows, or Linux

If you have a Mac or you use Linux, then you’re in luck. SQLite already comes with the Mac, so there’s nothing to do here.

If you are using Windows and have a machine powerful enough to run VirtualBox, I would recommend grabbing a lightweight version of Linux like Lubuntu or Xubuntu.

In the Linux console, you simply have to enter the command:

sudo apt-get install sqlite

If you really want to stick with Windows, then head over to SQLite.org and go to the download page. Download the DLL zip file and the tools zip file, since we'll be using the command line interface throughout this course.

You can unzip these files to C:\sqlite, and add this directory to your PATH environment variable.

You can find instructions for how to do that here: https://stackoverflow.com/questions/23400030/windows-7-add-path

To check that you've done it correctly, open up cmd.exe, and type in sqlite3 - it should open the SQLite command line shell.

# Chapter 2: Relational databases and getting data

## What is a relational database?

In this section I’m going to answer the question - what is a relational database? - and give you some concrete examples.

A relational database is a collection of tables. As you know a table has rows and columns.

The columns are referred to as “fields” or attributes. Each row is a new record.

| | | |
|---|---|---|

| | field1 | field2 |
|---|---|---|
| record1 | | |
| record2 | | |

So for example I could have a table called “users”, with the fields “name” and “email”:

Users:

| name | email |
|---|---|
| Bob | bob@gmail.com |
| Jane | jane@gmail.com |

Each table, or relation, usually refers to a specific entity. So users was one

example. Products might be another example. Orders would be another example.

The “relational” part of a relational database comes from the fact that different tables can be related to each other.

For example, if your users table had an “id” field like this:

Users:

| id | name | email |
|---|---|---|
| 1 | Bob | bob@gmail.com |
| 2 | Jane | jane@gmail.com |

And an orders table like this:

Orders:

| id | user_id | total_cost |
|---|---|---|
| 50 | 1 | 9.99 |

The order with ID 50 means it belongs to Bob, since Bob has the user ID 1.

## Loading the data used in this book

In this section we're going to load some data into our database for the first time.

In order to get the files needed for this course, you'll need to download them from Github. If you don't already know, Github is a version control system, and it allows me to easily keep all the files for this course up-to-date and allows you to easily get updates from the command line without having to re-download them from a website. You'll want to go to:

https://github.com/lazyprogrammer/sql_class

I’ve created a pre-made SQL script and some data files for you to load, so you just need to run these commands:

```
sqlite3 the_database.db < create_actions.sql
```

Now go into the DB:

```
sqlite3 the_database.db
```

Type in these commands to import the data:

```
.mode csv

.import small_actions.csv user_actions
```

Now the data is there, and we’ll look at how to view it in a later lecture.

If you’re interested in knowing how the data was generated, look at the Python file generate_actions.py - it basically creates a CSV, which is like an Excel table but in plain text, with 4 columns of data - name, product, action, and price. You can even open the file in Excel and look at the raw data if you want to.

# Chapter 3: Basic commands

## Basic commands

Commands in SQLite start with a “dot”. We’ll just introduce them as needed in this course, but here are a few important ones:

See the databases you are currently connected to:

.databases

.schema - Shows us the table we just created - it is called user_actions, and has 4 fields - name, product, action, and price. It also shows us the data types which we’ll look at more in depth later.

.tables - This will show us a list of all the tables in our database.

This command turns headers on - so we can see the names of the fields when we display query results - this will be useful in the future:

.header on

And lastly, .exit will exit the SQLite shell.

If you ever need a quick overview of all the SQLite commands, just use:

.help

## Querying a table

In this lecture I’m going to show you the basic structure of a query. This is the simplest form:

```
SELECT *

FROM user_actions;
```

Note 3 things about this.

One is select and from are capitalized - we don’t really need to do this, but it helps some people see the formatting better.

Two is that newlines and whitespace are all treated the same, so it doesn’t matter if I use a space, or a newline, or 5 newlines. But writing it like I did above is somewhat of a standard way to write SQL queries.

Three is that all SQL statements need to end in a semicolon, just like Java or C++.

Give it a try in the console!

First let’s turn on some useful formatting features:

.header on

.mode column

select * from user_actions;

Now let’s look at a variation of this - let’s say I only want to know about the rows where the action is addtocart. The format of this query is:

SELECT *

FROM user_actions

WHERE action = ‘addtocart’

Now let’s say I don’t care about seeing the action or price, I just want to know the name and product they bought. To do this I replace the star with the columns

I want to see.

```
SELECT name, product

FROM user_actions

WHERE action = ‘addtocart’
```

You can also have multiple conditions to check. You can combine conditions using AND or OR. And you can have different comparison operators like “=”, “!=”, “<”, “>”, “IS”, “IS NOT” (the latter 2 of which are used for checking NULL).

```
SELECT name, product

FROM user_actions

WHERE action = ‘addtocart’ AND name = ‘Bob’
```

## Creating a table

So earlier we used a script to create a table but we never looked at what was inside it. Now we are going to discuss the syntax of how to create a table.

CREATE TABLE table_name(

Column_name1 TYPE,

Column_name2 TYPE,

…

);

So you see you need to define both what the name of each column is as well as its type. The .sql file we used before had these contents:

CREATE TABLE user_actions(name TEXT, product TEXT, action TEXT, price REAL);

## Data Types

What are the different types in SQLite? They are very similar to what you see in other programming languages like C++ and Java.

TEXT - for strings like we have been using (VARCHAR can also be used but it's treated the same in SQLite, this is easier and shorter)

INTEGER - this is for integer values

REAL - this is for any integer or float value

BLOB - this is for storing binary data

DATETIME *DATE* TIME - technically these are treated as one of the above 4 data types depending on what data you put in it. We use functions to interact with them.

Don't worry if you don't understand this fully right now, we'll see more examples throughout this book.

# Modifying a table’s structure

One powerful feature of relational databases is that we’re not stuck with the same data format forever.

Let’s say you want to add a new column to an existing table - we would like to know the timestamp of our user_actions. Now we can’t retroactively populate the timestamp unless we have that data somewhere else, but what we can do is add a new column that can hold the timestamp, and any data that would be inserted or retrieved in the future can make use of this new column.

So let’s look at the syntax of this:

```
ALTER TABLE user_actions ADD COLUMN created_at DATETIME;
```

In general it would be like:

```
ALTER TABLE table_name ADD COLUMN new_column_name TYPE;
```

So let’s run this on our table.

Now let’s select everything from this table:

```
select * from user_actions;
```

Notice how all the existing rows just get a null value.

Let’s play around a little to see what we can do with this column and to get you used to different SQL queries.

I’m going to set a datetime for every row that has the name “Gina”:

```
update user_actions

set created_at = datetime('2016-01-01 00:00:01’)

where name = 'Gina';
```

Check the table again:

```
select * from user_actions;
```

Notice the format for a datetime is year-month-day hour-minute-second.

Now I’m going to select all the rows where created_at is before right now:

```
select * from user_actions where created_at < datetime('now');
```

The other ones don’t show up since they don’t have datetimes.

Note that an alternative way of adding columns would be to create a new table with the columns you want, copying the data over, and deleting the old table. Since there is no way to remove a column in SQLite, for some operations this methods is your only option.

# Chapter 4: Speeding things up with indexes

# Indexes

If I gave you a list of numbers, and I told you to look for a specific one, how would you find it?

[5,2,3,9,7]

You would have no choice but to check every single element until you find all the ones equal to the one I asked you to find.

You can imagine how this would be prohibitive if you had a very large data set.

What about if I gave you a sorted list of numbers and asked you to find one?

[1,2,3,4,5,6,7,8,9,10]

Let’s say I’m looking for the number 9. If you looked in the middle, 6, and see that 6 is less than 9, I now know I only need to keep looking in the top half.

[6,7,8,9,10]

Now I look in the middle again, and I see 8. I know 9 is bigger than 8, so I look in the top half again.

[9,10]

Now I can easily retrieve the 9.

So you see that putting the numbers in a specific format allows you to search more efficiently.

Indexes are a special data structure that allow you to look things up in a table more efficiently. There are several types of indexes that you can have in a relational table.

Vanilla/plain index - I could make the actions column easier to search.

Unique index - I could make the actions column easier to search AND force them to be unique. Obviously I wouldn't want to do this for the actions column since multiple users can perform the same action multiple times.

Composite index - I could make an index on both (action, price) simultaneously.

Syntax:

CREATE INDEX index_name ON table_name (column_name);

CREATE UNIQUE INDEX index_name ON table_name (column_name);

CREATE INDEX index_name ON table_name (column1, column2);

# Primary Keys

There is a special type of index called the primary key.

Let’s say we have an orders table: id | user_id | total_price

And a users table: id | name | email

The user_id in the orders table, which refers to the id in the users table - is usually what we would call the primary key. You can make something else the primary key but that would be unconventional.

Primary keys are unique indexes that also come with another special capability - the ability to autoincrement. That means if I have user IDs 1 and 2 in the database already, I can insert a new row by name and email only - and it will automatically have the ID 3.

How to I create a table with a primary key?

```
CREATE TABLE users(

id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT,
```

email TEXT

);

You can see that because I have to specify both primary key and the autoincrement keywords, both are actually optional and I could put them somewhere else, but at the same time, this method is pretty much standard.

## Proving that indexes make things faster

In this section I’m going to demonstrate how indexes can speed things up.

First we’re going to need a larger dataset so we can see a measurable difference in speed. I didn’t include this file in the repo because it would’ve made the repo too big. Just run:

python generate_data.py 1000000 big_actions.csv

Now let’s go into SQLite.

First let’s recreate the user_actions table since I don’t want the created_at column.

```
drop table user_actions;

CREATE TABLE user_actions(name TEXT, product TEXT, action TEXT, price REAL);
```

Let’s turn the timer on and import the data:

```
.timer on

.mode csv

.import big_actions.csv user_actions
```

And let’s run a query searching for the name Gina.

```
select count(*) from user_actions where name='Gina';
```

Count just counts the number of rows in the answer to a query - we'll cover that more later.

Now let's create an index on name.

```
create index name_index on user_actions (name);
```

Run the query again.

It should be about 10 times as fast.

# Chapter 5: Modifying a table's data

In this lecture we're going to talk about how to modify data inside a table. There are 3 basic operations we can do: insert, update, or delete.

## Insert

Let's say I want to insert a new row into a table. Let's look at our orders table again. Suppose "John" buys a "Mango", so the action would be "purchase" - how would I get that into the database? The general syntax is:

INSERT INTO table_name VALUES (?, ?, ?, …);

This requires that every column in the table is accounted for. What if you want to insert only certain columns, and have the rest of the columns, like price, take on a default value?

INSERT INTO user_actions (name, product, action) VALUES (‘John’, ‘Mango’, ‘purchase’);

Now let’s look for it:

select * from user_actions where name = 'John';

Notice how price is just NULL.

## Update

Now let’s look at updating. Let’s say the price of John’s Mango is 2.99. The general syntax for updating is:

UPDATE table_name

UPDATE table_name

SET column1=value1, column2=value2, …

WHERE condition;

So in our case it would be:

UPDATE user_actions

SET price=2.99

WHERE name=’John’;

Check if it’s been updated correctly (you can hit up to use the previous command).

## Delete

Now let’s say we want to get rid of this new row. The general syntax for deletion is:

DELETE FROM table_name

WHERE condition;

So for us it would be:

DELETE FROM user_actions WHERE name = ‘John’;

Check if it’s been updated correctly.

# CRUD

In this section we're going to draw an analogy between SQL and web development.

A lot of the time, separating the data from the application gives us a nice architecture where we can split up, the data, the UI, and the application code. You can think of this as model-view-controller or MVC.

The UI is what you see in your browser, like HTML, CSS, and Javascript.

The application code is a server language like PHP, Ruby, or Python.

The database can be an SQL database or a NoSQL database.

What's neat about this is that the application, which accepts and responds to HTTP requests, has analogous verbs to SQL, and we can generalize them as create, read, update, and delete, or CRUD.

Create is an INSERT in SQL, or a POST in HTTP, which you may have seen if

you’ve ever looked at an HTML form.

Read is a SELECT in SQL, or GET in HTTP.

Update is an UPDATE in SQL, or PUT in HTTP.

Delete is just delete in all of them.

This is a very important idea to keep in mind for application programmers, because you have a lot of freedom when you’re developing a web application. You technically could delete rows in the database or update them on a GET request, but it’s a good idea not to.

# Chapter 6: Joining tables together

In this chapter we are going to talk about joins. So far we've been working with a user_actions table - it shows us the name of a user, the product they were taking an action on, what the action was - whether it was a view, addtocart, or purchase, and the price.

But of course this necessitates having a users table too - it might contain information like the user's location and their age - information you can pull useful analytics from.

So let's say we wanted to know the location where most people are buying mangoes - that's information from 2 different tables - and to merge these 2 tables together we do what is called a join. In following sections I'm going to talk about the different kinds of joins you can do.

This is the user_actions table we'll be referring to in the next sections:

user_actions

user_actions

| name | product |
|---|---|
| Bob | Mango |
| Jane | Apple |
| Carol | Banana |

And this is the users table we'll be referring to:

users

| name | location |
|---|---|
| Alice | Las Vegas |
| Bob | New York |
| Jane | Chicago |

# Inner Join

The first kind of join we'll talk about is an inner join. The new SQL keywords here are INNER JOIN where you choose the 2nd table to join with, and the ON keyword where you choose HOW to join the 2 tables. Note that you can join on more than one column.

```
SELECT users.name, product, location

FROM user_actions

INNER JOIN users

ON user_actions.name = users.name;
```

Result:

| name | product | location |
|---|---|---|
| Bob | Mango | New York |
| Jane | Apple | Chicago |

So what happened to Alice and Carol here? They go away because Alice doesn't show up in the user actions table and Carol doesn't show up in the users table. That's what we mean by in INNER join.

## Outer Join

Next let's look at OUTER JOINs. The only difference in syntax here is we say "full outer join" instead of "inner join". Notice how Alice and Carol have

reappeared, and any information they didn’t have is just set to NULL.

```
SELECT users.name, product, location

FROM user_actions

FULL OUTER JOIN users

ON user_actions.name = users.name;
```

Result:

| **name** | **product** | **location** |
|---|---|---|
| Bob | Mango | New York |
| Jane | Apple | Chicago |
| Carol | Banana | |
| Alice | | Las Vegas |

Note that SQLite does not support full outer joins - but other SQL databases do.

# Left Join

Next let’s look at LEFT OUTER JOINs. Carol stays, but Alice goes away, since Alice was on the right-side table.

```
SELECT users.name, product, location

FROM user_actions

LEFT OUTER JOIN users

ON user_actions.name = users.name;
```

| name | product | location |
|---|---|---|
| Bob | Mango | New York |
| Jane | Apple | Chicago |
| | | |

| | | |
|---|---|---|
| Carol | Banana | |

# Right Join

Now let’s look at RIGHT OUTER JOINs. This is the opposite. Alice stays and Carol goes away. Note that SQLite only supports LEFT outer joins but not RIGHT or FULL outer joins.

```
SELECT users.name, product, location

FROM user_actions

RIGHT OUTER JOIN users

ON user_actions.name = users.name;
```

| **name** | **product** | **location** |
|---|---|---|

| | | |
|---|---|---|
| Bob | Mango | New York |
| Jane | Apple | Chicago |
| Alice | | Las Vegas |

# Chapter 7: Calculations and Aggregations - Group, Sort, Limit

In this chapter we're going to finally start to look at some functions that will help you do analytics. The previous lectures helped to form the foundation and basic skillset that we can now apply to do more complex things.

# Count

The first thing we'll do is simple counting. Let's say I want to know how many people bought mangoes. We simply use the COUNT() function you saw me use earlier in the book.

```
SELECT COUNT(*)

FROM user_actions

WHERE product = 'Mango';
```

# Distinct

Now let’s say I want to know how many different fruits there are in the database. I only want to count the number of distinct fruits. I can’t use COUNT() by itself. For this we can use the distinct keyword.

```
SELECT COUNT(DISTINCT product)

FROM user_actions;
```

## Sum

Now let’s say I want to find the total amount people are spending on mangoes. So I only want the rows where product is equal to mango AND the action was purchase. And I want to sum the price column.

```
SELECT SUM(price)

FROM user_actions

WHERE product = ‘Mango’ AND action = ‘purchase’;
```

## Min, Max, Avg

By now you get the idea. We are taking some aggregate of the price column. MIN(), MAX(), and AVG() will all return the same answer since the script assigns $0.99 to all the prices. In the next section we’ll do more complex things with these functions.

```
SELECT AVG(price)

FROM user_actions

WHERE product = ‘Mango’;
```

# Group By

In the last section we were only summing one thing at a time - like the price where the product was Mango. But what if we wanted to do something like, find the total price for EACH product in the same table?

```
SELECT product, SUM(price)

FROM user_actions

WHERE action = ‘purchase’

GROUP BY product;
```

So now there are multiple parts to this query. I’m going to select the product name and the sum of the prices. The prices are per product since I’m grouping by product at the end. And I’m using WHERE so that we only look at purchases.

# Sorting

Now let’s say I want to order by the revenue in descending order. Then I need to add 2 things. One is I need to add an “order by” statement to the end. But I need to give that column a name so I can order by it. So I give the column a name in the select clause, and I order by that name in the order by clause. Note that the default ordering is ascending.

```
SELECT product, SUM(price) as total_revenue

FROM user_actions

WHERE action = ‘purchase’
```

```
GROUP BY product

ORDER BY total_revenue desc;
```

# Limit

Now let’s say I only want to look at the top 3 performing products. Then I can use the limit keyword. This is really useful if your SQL query has a lot of results and is taking a long time to print.

SELECT product, SUM(price) as total_revenue

FROM user_actions

WHERE action = ‘purchase’

```
GROUP BY product

ORDER BY total_revenue desc

LIMIT 3;
```

# Chapter 8: More examples - sales funnel, YOY revenue, sales by location

In this chapter we're going to use the techniques we learned in the previous lectures to answer questions that sales and marketing will commonly have about the data.

# Sales Funnel

First is funnels. We want to know how many people are making it to each checkpoint so that we can see where the large drop-offs are, analyze why that's happening, and try to improve the conversion rate at that point. So this is just a simple group by and count as we've seen.

```
SELECT action, COUNT(*)

FROM user_actions

GROUP BY action;
```

# Year-over-year Revenue

Next is year over year revenue. Typically we'd want to display a chart in Excel, so we would need a table with 2 columns, date and revenue.

Since we don’t have a table with any datetime, we’ll use a script I included to the repo to create one, just run:

```
python generate_actions_dt.py
```

Next, create a table in SQLite:

```
CREATE TABLE user_actions_with_dt(name TEXT, product TEXT, action TEXT, price REAL, dt DATETIME);
```

Then do the import:

```
.mode csv

.import dt_actions.csv user_actions_with_dt
```

We have a slight problem because this new table has datetimes, but we don’t really need times, nor do we need the day. We just want to group by month. So how do we convert datetimes into months?

The key is to use a function that is common in all programming languages, called strftime(), which usually takes in 2 parameters, a string that allows you to specify the output format, and a datetime object - the output is a string in the format you specified, created from the datetime object.

strftime(output_format, datetime_object)

We'll use this to create a new column called month, and group by that column.

```
SELECT strftime('%Y-%m', dt) as month, SUM(price) as revenue
FROM user_actions_with_dt
GROUP BY month;
```

And so now you can see it's in a format that's appropriate for an Excel chart. You can either copy and paste it manually or output to a CSV. To output to a CSV make sure you set the mode to CSV, and then use the .out command to specify an output file. Then the next query you enter will be output to that file.

```
.mode csv
```

.mode csv

.out outputfile.csv

<your query here>

# Sales by Location

The last example we'll do is sort the number of sales by location. In order to do this we'll have to make use of joins, because the sales data is in the actions table and the locations are in the users table.

```
SELECT COUNT() as sales, location

FROM (SELECT product, location

FROM user_actions
```

INNER JOIN users

ON user_actions.name = users.name)

GROUP BY location

ORDER BY sales DESC;

So I snuck in a few new concepts here.

One is notice that COUNT() doesn’t actually need any arguments, it’ll just count the number of rows in that group.

Two is that I’m using a nested query. The outer query is selecting from a table that’s not actually a table, but the result of another query!

Technically I didn’t need to select product, but I thought that would make the output more clear.

# Chapter 9: SQL on Spark

In this chapter we're going to discuss how to do SQL on a big data framework called Spark. This is a little more advanced so you'll need to know a little bit of Python coding to really be comfortable.

Spark also allows you to write code in Scala or Java, if you happen to know these. You can start your own Spark cluster using a command line script, which I'll explain in the next lecture, or you can install it locally. In this lecture we'll look at a local installation.

The cool thing about Spark is that it can access data from anywhere: your local file system, the Hadoop file system, and S3, which is Amazon's simple storage service.

Most likely your data resides in one of these 3 places.

## Installing Spark Locally

# Installing Spark Locally

So when you're ready to install Spark:

Download Spark at https://spark.apache.org/downloads.html

Decompress and run the build script from the root folder:

```
build/mvn -Pyarn -Phadoop-2.4 -Dhadoop.version=2.4.0 -DskipTests clean package
```

You may need to install Java 1.8, and set JAVA_HOME environment variable.

Run a test to make sure it's all working:

```
./bin/run-example SparkPi 10
```

One interesting thing about Spark SQL is it doesn't read CSVs, but instead it reads JSON files. JSON files have a convenient key-value format as you can see here.

{"name": "Bob", "email": "bob@gmail.com", "location": "Las Vegas"}

You'll need to run a few Python scripts to get the data file we need for this example.

If you haven't run this yet, do so now:

```
python generate_actions.py 1000000 big_actions.csv
```

Then run this to get the JSON:

```
python convert_actions.py
```

# Running a Spark Query

If you look at spark.py, this is just a very simple example that shows you how to load some data and display it in a table.

```
from pyspark import SparkContext

from pyspark.sql import SQLContext


sc = SparkContext("local", "Simple App")

sqlContext = SQLContext(sc)


df = sqlContext.read.json("data.json")

df.show()
```

So if we go through this line by line you'll see some Spark imports and some initializations. You need to create a Spark context if you're running a script, but

if you’re using the Spark shell it’s created for you.

We create an SQL context form the Spark context and use the functions that it has to load the JSON file we created earlier.

Notice the use of df - that stands for “data frame” - so if you know R or Pandas there are some other things you can do that treat the data as a data frame similar to those tools.

The last step is to just print the data frame in the console.

Let’s look at spark2.py. This is a script that loads the JSON we just created and runs a simple query on it.

```
from pyspark import SparkContext

from pyspark.sql import SQLContext


sc = SparkContext("local", "Simple App")
```

```
sqlContext = SQLContext(sc)

df = sqlContext.read.json("big_actions.json")

df.registerTempTable("user_actions")

df2 = sqlContext.sql("SELECT COUNT(*), product FROM user_actions
WHERE action = 'purchase' GROUP BY product")

df2.show()
```

The new step is to give that data frame a table name because as you know a SQL query needs a table name.

And then we'll run a simple query, just selecting the number of sales of each product.

And the last step is again to print the results.

This is powerful because now you have some boilerplate code, in which you can insert any SQL query you can think of.

Note that we don't just run "python spark2.py" - we need Spark to run this Python script. It'll be responsible for copying the script over to all the machines in the cluster and coordinating the work. First I always like to create an alias for the spark-submit binary:

alias spark-submit="*Users*macuser/Code/spark-1.6.1/bin/spark-submit"

Then we'll run spark submit. This command line argument is telling Spark that the master machine is localhost on port 7077.

spark-submit --master spark://localhost:7077 spark2.py

What this means is you could potentially run a script where some other machine is the master.

# Creating a Spark Cluster

So let's say no one at your organization is using Spark yet, but your data is on Hadoop or S3. You can still use Spark! In this lecture I'm going to show you a very simple way to create your own Spark cluster.

The first thing you'll want to do is create a Key Pair inside AWS and download the PEM file. You can also use an existing Key Pair if you have one.

Next step is to set your AWS environment variables.

```
export AWS_ACCESS_KEY_ID=…

export AWS_SECRET_ACCESS_KEY=…
```

The next step is to actually create your cluster. You can copy this command, of course putting in your real key pair name, the path to your PEM file, the number of slave machines you want, the name of your cluster, and the zone you want to

create the machines in.

./ec2/spark-ec2 -k “Your Key Pair Name” -i *path*to/key.pem -s <number of slaves> launch <cluster name> —copy-aws-credentials -z us-east-1b

You can look up on Amazon what zones are available.

To run the job, it’s exactly the same as before, but you just switch out localhost with the IP address of your master machine, which you can get inside the AWS console.

spark-submit --master spark://<master-ip>:7077 spark2.py

After you’re done doing your data analysis, don’t forget to destroy your cluster since AWS can get very expensive.

./ec2/spark-ec2 destroy <cluster name>

Use the same spark-ec2 script from before but use the destroy command. You'll need to remember the cluster name you used to create your cluster because that's the same name you'll use here.

# Conclusion

I really hope you had as much fun reading this book as I did making it.

Did you find anything confusing? Do you have any questions?

I am always available to help. Just email me at: info@lazyprogrammer.me

Do you want to learn more about data science? Perhaps online courses are more your style. I happen to have a few of them on Udemy.

The companion course to this book is located at:

SQL for Marketers: Dominate data analytics, data science, and big data

https://www.udemy.com/sql-for-marketers-data-analytics-data-science-big-data

One of the most powerful data science techniques today is deep learning. You can learn the fundamentals of deep learning at:

Data Science: Deep Learning in Python

https://udemy.com/data-science-deep-learning-in-python

Are you comfortable with this material, and you want to take your deep learning skillset to the next level? Then my follow-up Udemy course on deep learning is for you. Similar to this book, I take you through the basics of Theano and TensorFlow - creating functions, variables, and expressions, and build up neural networks from scratch. I teach you about ways to accelerate the learning process, including batch gradient descent, momentum, and adaptive learning rates. I also show you live how to create a GPU instance on Amazon AWS EC2, and prove to you that training a neural network with GPU optimization can be orders of magnitude faster than on your CPU.

Data Science: Practical Deep Learning in Theano and TensorFlow

https://www.udemy.com/data-science-deep-learning-in-theano-tensorflow

Would you like an introduction to the basic building block of neural networks - logistic regression? In this course I teach the theory of logistic regression (our computational model of the neuron), and give you an in-depth look at binary classification, manually creating features, and gradient descent. You might want to check this course out if you found the material in this book too challenging.

Data Science: Logistic Regression in Python

https://udemy.com/data-science-logistic-regression-in-python

To get an even simpler picture of machine learning in general, where we don't even need gradient descent and can just solve for the optimal model parameters directly in "closed-form", you'll want to check out my first Udemy course on the classical statistical method - linear regression:

Data Science: Linear Regression in Python

https://www.udemy.com/data-science-linear-regression-in-python

If you are interested in learning about how machine learning can be applied to language, text, and speech, you'll want to check out my course on Natural Language Processing, or NLP:

Data Science: Natural Language Processing in Python

https://www.udemy.com/data-science-natural-language-processing-in-python

Finally, I am *always* giving out **coupons** and letting you know when you can get my stuff for **free**. But you can only do this if you are a current student of mine! Here are some ways I notify my students about coupons and free giveaways:

My newsletter, which you can sign up for at http://lazyprogrammer.me (it comes with a free 6-week intro to machine learning course)

My Twitter, https://twitter.com/lazy_scientist

My Facebook page, https://facebook.com/lazyprogrammer.me (don’t forget to hit “like”!)