CS253 – Unit 3: Databases

Contents

Introduction

This unit is all about databases. In unit 2 we learned how to get data from our users using forms. In this unit we will see how to take that data and store it in a database so it can be used in other requests and to make more complex pages.

This lecture is divided into two parts. The first part will deal with higher-level concepts and how to think about databases. In the second half of the lesson we will work with Google App Engine, and it's data-store, and build a little project together.

This lecture is the reason we decided to use Google App Engine for the whole course. Google App Engine has a built-in database that we don't need to install, or deal with system administration and the like.

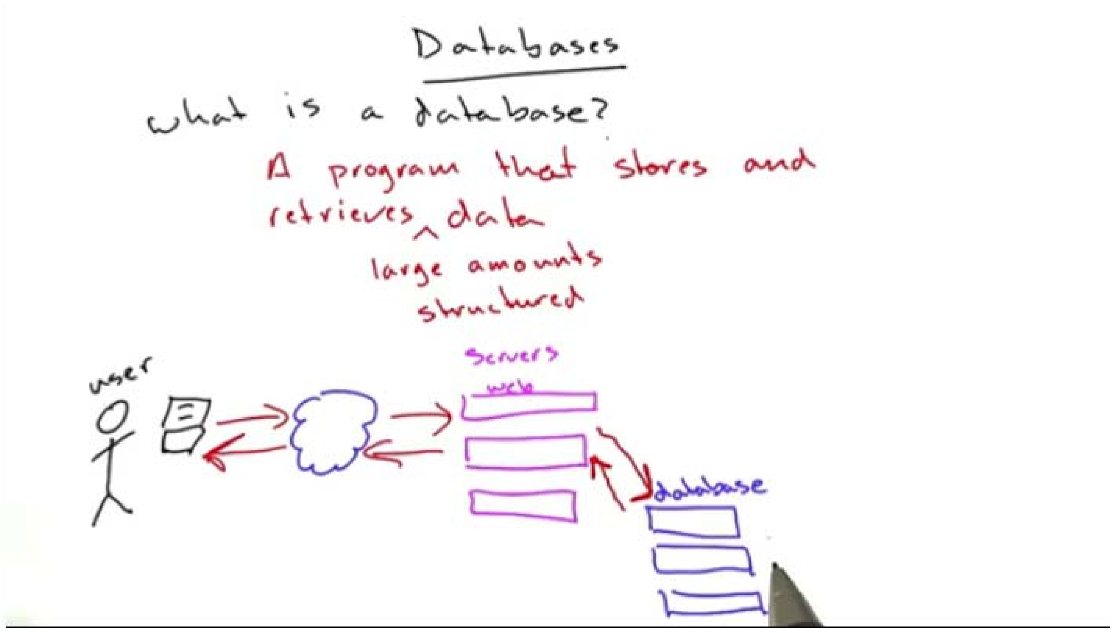
By the end of this lecture, you will be working on your next homework which is to actually start implementing the blog, so you will have basic blog entries in the project we have been alluding to.

Let's have some fun…

Databases

The first question we want to answer is "what is a database"?

There are a number of ways to answer this question. Probably the simplest definition is "A program that stores and retrieves data". Particularly, large amounts of data. Even more particularly, large amounts of structured data.



The term "database" can also be used to refer to the machine or server, that runs the database program, or even a system of machines that run together to operate the database program. This is similar to the way that the term "web server" can be used to refer to the web server program itself, or to the server or group of servers that run that program.

What is a Database? Quiz

What can a database refer to?
Select all that apply

- A program that stores and retrieves data
- The machine running that program
- A group of machines working together to store/retrieve data.

Tables

Almost every database implementation uses the concept of **tables**.

Let's imagine that we are building a site like Reddit that's going to take links from users, store them in a database, let users vote on them, and then display the most popular. To do this, we will probably have a table of links. Each link is made up of a number of things, for example:

- an ID
- a number of votes
- the user who submitted the link
- the date the link was submitted
- the title we display on the site
- the URL

Each of these will be a **column** of our database.

Every entry in our table is called a **row**. We expect to have many, many rows storing all the links that are submitted.

ID	Votes	User	Date	Title	URL
5	207	21	13:59	Zombie Dogs!	www.zombiedogs.com
6	0	27	20:00	…	…
…	…	…	…	…	…

Let's talk a little more about the columns.

The **ID** column is important in almost all tables. In general, every row will have an ID, and this is how you refer to that row specifically. The first row in the table above is link number 5. You are not required to have something called an ID in most databases, but it is really helpful. It is usually an integer, but can also be a string (Google App Engine allows either integers or string).

The **Votes** column is also an integer, and refers to the number of votes for that link.

The **User** column is another integer, and is a reference to the ID of the user who submitted the link. This will refer to another table, say User, with columns holding details of the user:

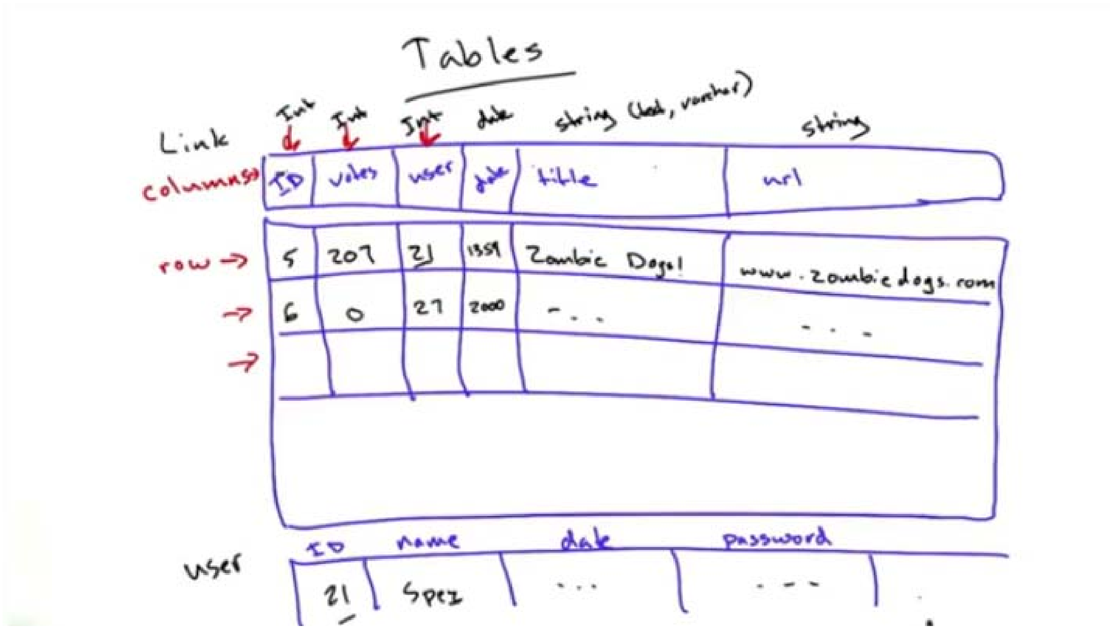
ID	Name	Date	Password	…
21	Fred	…	…	…

We will look into these types of relationships more later.

The **Date** column probably has the type 'date'. Not all databases support the date type, but those we'll be using in this class do, and it is really handy to have.

The **Title** and URL columns are strings. In some database systems these may also be known as ***text***, or ***varchar***.

Before we move onto specific database stuff, let's look at some examples in Python of how we can manipulate this kind of structured data.



Implementing Tables In Python Quiz

Implement the function query() to return the number of votes for the link whose ID is 15.

Querying Quiz

Modify the function query() to return a list of Links submitted by user 62443, sorted by submission time ascending

Why Databases

These problems are not too hard, but they are a little tedious. Querying data by hand has a number of downsides:

- It is very error prone. You have to write code for every query you want to answer.
- It is tedious!
- It is slow!

Databases allow us to take large chunks of data, perhaps millions of rows, and answer queries on that data in a reasonable amount of time, without having to write lots of custom code.

Types Of Databases

There are lots of different types of databases. We will focus on the major ones.

The first of these are **Relational Databases**, and these often use a language called SQL to manipulate the data. Some examples of relational databases are PostgreSQL which was used at Reddit and Hipmunk, and is also used at lots of other places, MySQL, which is extremely popular and is used by Facebook among many others. In fact, MySQL is currently far-and-away the most popular relational database system. Also really common are SQLite, which we will be using in this class, and Oracle. PostgeSQL, MySQL and SQLite are all free to use.

These relational databases all work with tables, and use the language SQL to manipulate those tables. They are useful general-purpose products.

There are other databases in the world, for example, Google App Engine's Datastore, which we will be using a lot in this course, and which has a lot in common with relational databases, but which also has a lot of differences. Another product is

Amazon Dynamo, and the range of NoSQL databases, of which some of the best known are Mongo and Apache Couch.

In this class, we will focus on the basics of SQL and on Google App Engine's Datastore.

Types Of Databases Quiz

Which database is the best?

- MySQL
- PostgreSQL
- Google App Engine Datastore
- Dynamo

SQL

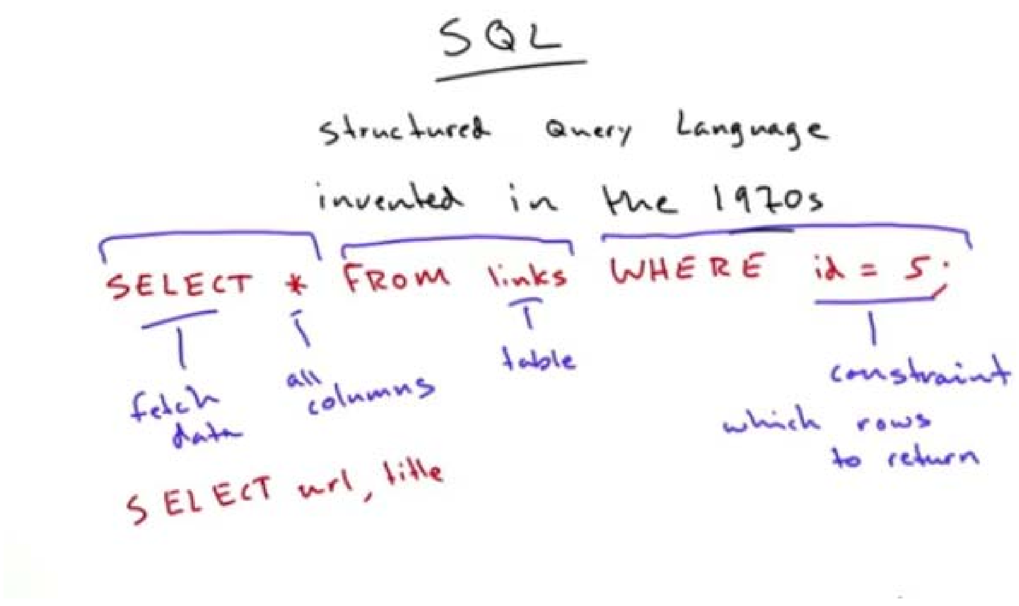
Lets talk a little bit about **SQL** or '***Structured Query Language***'. SQL is a language that is not-quite a programming language, but is a language for expressing queries. SQL is used on relational databases to ask questions about the data and to extract data from it. SQL was invented in the 1970s which was long before the Internet and web applications existed.

A basic SQL query may look something like this:

```
SELECT * FROM links where id = 5;
```

What this is actually saying is:

SELECT (i.e. fetch data) * (i.e. all columns) FROM links (i.e. from the table 'links') WHERE id = 5 (i.e. where the value in the id column is equal to 5).



The query can be considered as several parts:

> SELECT * is what we are retrieving. The * could actually be a list of specific columns. So, if you just wanted to retrieve the URL from table links, you would enter SELECT url.
>
> FROM links is where we will retrieve the data from.
>
> WHERE id = 5 is what is known as a '***constraint***', which limits which rows will be returned by the query.

Databases In Python

We can use SQL in Python by importing the module SQLite3 which is built into Python:

```
import sqlite3
```

This will allow us to use SQLite in Python. In the Python code used for the last examples, we have added some code to make – and populate – an in-memory SQL database:

```
# make and populate a table
db = sqlite3.connect(':memory:')
db.execute('create table links ' +
        '(id integer, submitter_id integer, submitted_time integer, ' +
        'votes integer, title text, url text)')
for l in links:
    db.execute('insert into links values (?, ?, ?, ?, ?, ?)', l)

# db is an in-memory sqlite database that can respond to sql queries using the
# execute() function.
```

The db variable represents our SQLite database.

The SQL code that actually creates the table is:

```
create table links  (id integer, submitter_id integer, submitted_time integer,
votes integer, title text, url text)
```

The fields are those we have seen before, but notice that the data type is specified.

To run SQL code on db within Python, we use the execute() function. To run the SQL statement we saw earlier we would simply enter the code as:

```
c = db.execute("select * from links where id = 5")
```

c is what is known as a "cursor" to the results of the query. The cursor is essentially a position in the database.

We can now use the fetchall() function on the cursor, c, to load all of the data from the database into this list of results. These results won't be Links; rather, they'll be tuples.

Putting a * in front of a tuple and passing it to a function the arguments get put in place to create the links. For example the object constructor:

```
link = Link(*link_tuple)
```

Let’s try a quiz.

Databases In Python Quiz

Add the SQL code to make the function query() return the number of votes the link with ID = 2 has

More Advanced SQL

We have seen how to select rows from a table where a specific constraint has been met:

```
select * from links where id = 5;
```

Let’s look at how we can make this a little fancier. We can have more complex constraints, for example:

```
select * from links where submitter_id = 5 OR votes > 23;
```

This will return all the links submitted by user 5, or where the link has more than 23 votes. This allows us to ask quite sophisticated questions of our database.

Advanced SQL In Python Quiz

Make the function query() return the ID of the link that was submitted by user 62443 and has > 1000 votes.

Order By

Next, let’s see how to sort our results. Consider the SQL statement:

```
SELECT * FROM links WHERE votes > 10 ORDER BY votes;
```

This will select all the links with more than 10 links, and will order them by the number of votes. By default, the votes will be in ascending order. You can make the order explicit by specifying either ASC or DESC in the ORDER by clause:

```
SELECT * FROM links WHERE votes > 10 ORDER BY votes DESC;
```

Order By Quiz

Make the function query() return a list of the IDs of the links that were submitted by user 62443 sorted by submission time ascending.

The solution to this problem follows the format we have been using in earlier quizzes. If we were actually to make a function just to return the ids of the links this solution is actually quite wasteful.

If all we are interested in is a list of ids, we don’t need to select * from links, we just need to select id:

```
"select id from links where submitter_id = 62443 order by submitted_time asc"
```

We also don’t need to create a Link object for every row of the results, we can return the ids almost directly. Since the SQLite library in Python returns tuples of the results, and since we’re only retrieving the id, we know that the first column will be the id. So what we need to do is to make a list of the first elements in the tuples for all the tuples in the cursor:

```
results = [t[0] for t in c]
```

This would make the solution:

```
def query():
   results = []
   c = db.execute("select id from links where submitter_id = 62443
                                             order by submitted_time asc")
   results = [t[0] for t in c]
   return results
```

Joins

Let's see how we can create SQL statements involving multiple tables. Recall the structure of the **links** table that we saw earlier:

ID	Votes	User	Date	Title	URL
5	206	**22**	01/10/01	Zombie Dogs!	www.zombiedogs.com
6	0	27	…	…	…
…	…	…	…	…	…

The User ID in the links table refers to the ID field in the **users** table:

ID	Name	Password	Date	…
22	Fred	Hunter2	20/06/08	…

There should be a valid entry in the users table for every unique user on the system.

One of the things that you can do in most SQL databases is something called a "**JOIN**". This is a SQL statement that involves two tables. Consider a basic SQL statement that looks something like:

```
SELECT * FROM link WHERE user_id = 22
```

This will return all of the links submitted by user 22. But what if we don't know the user's ID? What if we want to extract a list of all of the links submitted by users with the name 'Fred'?

In fact, there are a couple of ways we could achieve this. We could do a look up for the ID of the person name 'Fred' and then use this ID to search for links. Alternatively, instead of running two queries, we could combine this into one query:

```
SELECT link.* from links, users WHERE link.user_id = user.id
                                      AND user.name = Fred;
```

In reality, however, we don't use joins very often in web software. We'll see the reasons for this later in this lecture.

Indexes

So far, we have been doing what is known as **sequential scans**. A sequential scan is where you have a list of something, in this case links, and work through them in sequence. This works fine with relatively few items, but are very slow with large amounts of data (e.g. millions or even billions of links!).

An **index** on a computer is just like an index in a book. They make lookups faster. An index that you will already be familiar with in Python is the **hash table**. In Python you can have a dictionary that looks something like this:

```
index = {1: link1, 2: link2, 3: link3, …}
```

This provides a mapping of keys to values. You can do very fast lookups by writing something like:

```
index[2]
```

This refers to index key 2 of the hash table, and when we hash this value and find it in the hash table and return the value. We don't have to scan every value in the list, we can jump immediately to that element. This makes queries run much faster.

Querying Links Quiz

Implement the function link_by_id() that takes a link's ID and returns the link itself.

Using Dictionaries As Indexes Quiz

Implement the function build_link_index() that creates a Python dictionary that maps a link's ID to the link itself.

Lookup

We now have a function that will build our index. Let’s use it. If we assign the result of the function build_link_index() to the variable link index, we can re-write our function link_by_id() as:

```
def link_by_id(link_id):
   return link_index[link_id]
```

Now this works fine as long as the value link_id is found in our data, but if the value isn’t there then the Python code will generate an error.

It turns out that this is easy to fix. Instead of using the braces [] for link_index, we can use the Python hash table function **get**, thus:

```
def link_by_id(link_id):
   return link_index.get(link_id)
```

What **get** does is to check whether the key is in the hash table. If it does, it returns as before, otherwise it returns None.

So, we no longer have to scan all the way through the table when we want to run a query. Now we can just use the index.

Lookup Quiz

There’s one more function that we need. Implement the function add_new_link() that both adds a link to the "links" list and updates the link_index dictionary.

Advantages Of Indexes Quiz

Which of these statements are true?

- Indexes increase the speed of database reads.
- Indexes increase the speed of database inserts.

Real World Example

Steve illustrated the benefits of using indexes using the PostgreSQL database on Hipmunk.

As an example, Steve used the hotels table on Hipmunk, which had approximately 312,000 entries.

Running the SQL command:

```
select name from hotels where id = 51492;
```

returned one result.

Using the additional PostgreSQL command **explain analyze** with the above SQL command, we saw that the query ran a sequential scan on hotels which took 142ms.

Next, Steve created an index on the id field and ran the query again. This time the query only took 0.163ms. That is a substantial improvement!

Indexes For Sorting

One last thing to consider is the use of indices for sorting.

So far we have been talking about using hash tables to create an index. But one of the characteristics of a hash table is that they are not sorted. So we lose our sorting information when we use a hash table.

There is a different kind of mapping that we can use. This is called a **tree**. Trees are a basic data structure that accomplish something similar to a hash table, but have the additional nice property that they are sorted.

So why would you ever use a hash table when you could use a tree instead? Well, the downside to trees is that lookups are slower. In general, the time taken to look up a particular key in a hash table is not a function of the number of keys in the hash table. Hash tables have constant time lookups. In trees, lookup time is roughly a function of log(n), where n is the number of elements in the tree. Lookup speeds decrease as the size of the tree increases.

Another Real World Example

The Reddit Hotness Algorithm

If you've used Reddit, you'll know that it is just a big list of links, and that users can vote on those links, up or down. The front page is whatever is the most popular. Good links stay at the top of the page for a long time, while mediocre links may make an appearance but disappear after a short while. This is a really cool feature of Reddit and it's not that hard to compute. What they do is to use a special index. Let's look at how that works.

Reddit uses a link table similar to the one we have been using:

ID	ups	downs	date	score	
…	10	1	…	25	

The **score** column is the total score for the link.

The way that the Reddit hot algorithm works is that there is an index on the score field, called **hot_idx**.

One approach to generating the front page might have been to take all of the links submitted in the last 24 hours, do some fancy maths, sort them by how many up votes and down votes each has, and then display the page. However, this wouldn't capture the 'hotness' of how things rise and fall, so what Reddit actually does is that every time someone adds one up vote, Reddit increments the score by some other amount. The amount, **amt**, is computer through the 'hot' function which looks something like this:

```
amt = hot(ups, downs, date)
```

The function takes the number of up votes, the number of down votes and the submission date of the link. What happens is that, over time, the value of an up vote increases. This causes scores to constantly increase over time, so that newer links always have higher scores than older links. so that a link that is 1 day old with lots of votes may have the same score as a link that is 1 minute old with just a few votes, and this is what keeps the Reddit front page churning.

The page content can be generated from a very simple SQL statement like:

```
SELECT * FROM link ORDER BY score DESC;
```

Which is a very simple and fast query.

Scaling Databases

Let’s talk for a moment about how to scale databases. Like databases themselves, this is a very broad topic, but there are a couple of concepts that we need to introduce.

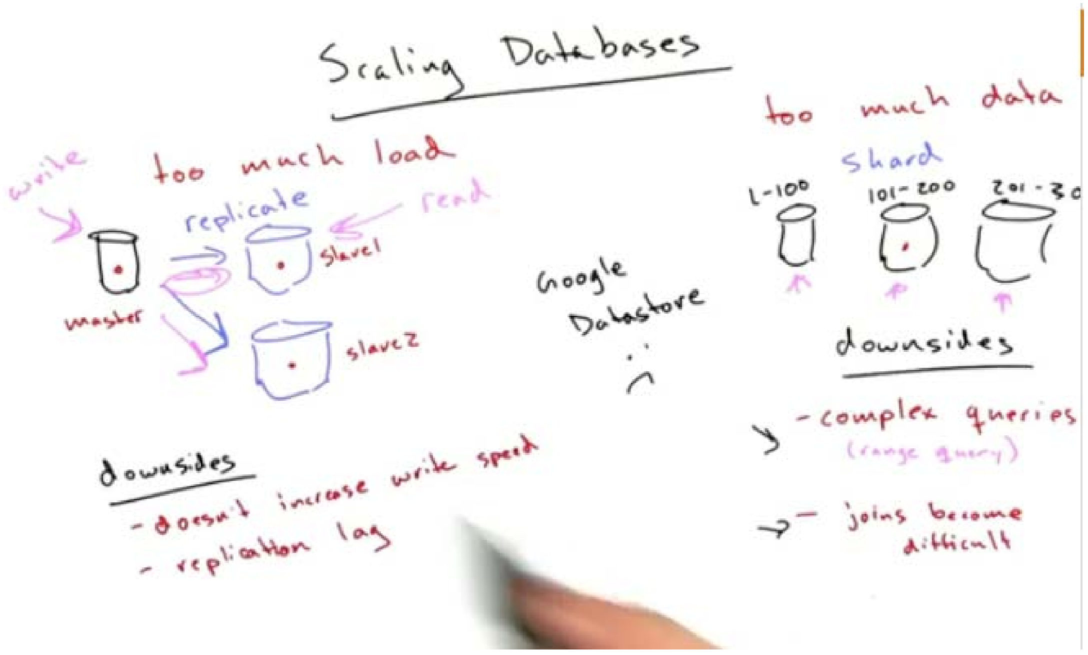
There are two reasons why you may need to scale your database:

1. Too much load – the database is just doing too much work.
2. Too much data for one machine.

In the first case, what you might do is to replicate the data to other databases. All data writes go to the master database, and are then replicated to the subordinate slave databases. Database reads can then be serviced from the slave databases.

There are some downsides to this approach. One is that it doesn’t increase the speed of writes. Another is the problem of replication lag where a read can occur an a slave database before the write to the master has been propagated,

One approach to dealing with the second case. where we have too much data for one machine is to **shard** the database. This is a fairly simple technique where, instead of having just one master, you have several. This also improves the write speed since there are now multiple masters to handle the load. A downside of sharding is that queries get much more complex. Another downside is that joins become difficult, or even impossible!



Scaling Databases Quiz

Which is an appropriate technique for increasing the read speed from a database?

- Get a faster machine.
- Replicate the database.
- Store less data.
- Press the turbo button.

Growing Databases Quiz

Which is an appropriate technique for growing a database that won't fit on one machine?

- Replicate the database.
- Get a bigger hard disc.
- Shard the database.
- Store less data.

Acid

There's one last set of concepts that we need to cover before we move onto some actual programming, and this is **ACID**. This stands for:

Atomicity – all parts of a transaction succeed or fail together.

Consistency – the database will always be consistent.

Isolation – no transactions can interfere with another's.

Durability – once the transaction is committed, it won't be lost.

One thing to be aware of is that it is very hard to create a database that is completely atomic, always consistent, where all transactions are isolated and which is completely durable. There is always an element of trade-off.

Acid Quiz

Replication lag is an example of the loss of which property?

- Atomicity
- Consistency
- Isolation
- Durability

Google App Engine Datastore

Let’s start working on some real code. We will work on an extended example that will reflect quite closely what you are expected to do for your homework for this unit.

We will be using the Google App Engine Datastore. This is the database provided by App Engine. There are a couple of things you need to know about Google App Engine Datastore.

What we have been referring to as tables are known as entities in Google App Engine Datastore:

Tables → Entities

Entities serve basically the same purpose as tables i.e. organising things of the same data-.type together.

A few important things to note about entities:

- The columns are not fixed
  - You can have whatever columns you want
  - You can even change the columns while developing the database
- Entities all have an id
  - id can be assigned automatically by the Datastore
  - You can make up your own id using integers or strings
- Entities have the notion of parents and ancestors.
  - Giving entities a parent can get around some of the limitations on consistency.

GQL

So far in this class we have been talking about SQL. In the App Engine Datastore we have something a little different, called **GQL**. GQL is basically a simplified version of SQL that only works in the Datastore.

The main difference between GQL and SQL is that all queries begin with SELECT *. There is no way to select individual columns. This also means we cannot do joins.

Another difference is that, in the generic SQL databases we have been talking about, we have been able to run arbitrary queries, no matter how slow, with or without indices. In App Engine, all queries must be indexed. For the most part, in the type of work we will be doing, you won’t have to build any indexes yourself. App Engine will build the indices for you.

Automatic Sharding And Replication

One last thing to consider is that the Datastore is sharded and replicated. Google doesn't share details of how this is implemented, but a lot of the constraints that we will be dealing with imply that the database is both sharded and replicated.

Some benefits for us are:

- We won't have to think about scaling (too much).
- Queries will be quick (because they have to be simple).

However, we will have to think about consistency.

Automatic Sharding And Replication Quiz

Do you understand everything there is to know about the App Engine Datastore>

- Yes.
- No.

ASCII Chan

We are going to build a website called ASCII Chan. This will be a message board for sharing ASCII art.

It will have the general structure where we will have a form that takes a title, some ASCII art and a submit button. A user can submit this and below the form the users will see ASCII art that ahs been submitted by other people.

ASCII Chan
title
Art
you
derp

This will be a one page website with the form at the top and the content below.

Getting Started on ASCII Chan

Shown below is the framework for a basic application:

```
import os
import webapp2
import jinja2

from google.appengine.ext import db

template_dir = os.path.join(os.path.dirname(__file__), 'templates')
jinja_env = jinja2.Environment(loader = jinja2.FileSystemLoader(template_dir),
autoescape=True)

class Handler(webapp2.RequestHandler):
        def write(self, *a, **kw):
                self.response.out.write(*a, **kw)

        def render_str(self, template, **params):
                t = jinja_env.get_template(template)
                return t.render(params)

        def render(self, template, **kw):
                self.write(self.render_str(template, **kw))

class MainPage(Handler):
        def get(self):
                self.write("asciichan!")

app = webapp2.WSGIApplication([('/', MainPage)],
                    debug=True)
```

This uses templates which allow us to keep the HTML in separate files that look like HTML with just little escapes for variables, rather than doing string substitutions for large wads of HTML.

Much of the code above will already be familiar to you. The class Handler includes some convenience functions, so the function **write()** saves us from having to type self.response.out.write() all the time.

The function **render_str()** takes a template name and returns a string of that rendered template, while the function **render()** which instead of returning just the string, calls write() on it.

At present, all the app does is to display the text string “asciichan” in the browser.

Creating The Form

Let’s take a brief look at how the template works. Consider the file front.html:

```
<!DOCTYPE html>
<html>
    <head>
        <title>/ascii/</title>
</head>
<body>
    <h1>/ascii/</h1>

</body>
</html>
```

The <h1> tag is a header tag.

If we now modify the MainPage handler in our app as follows:

```
class MainPage(Handler):
    def get(self):
        self.render("front.html")
```

The app will now display the **template** front.html.

OK, now we want to add our form to front.html. Initially, we add a text input field to allow the user to enter the title of their ASCII art:

```
<form method="post">
    <label>
        <div>title</div>
        <input type="text" name="title">
    </label>
</form>
```

Now we want to add an element to allow the user to enter their ASCII art.

Creating The Form Quiz

What is the most appropriate form element for inputting ASCII art?

- <input type=”text”>
- <textarea>
- <pre>
- <input type=”password”>

Textarea

Now we can add the <textarea> element to our form, remembering that <textarea> is not a void element like <input>, and it requires a closing tag:

```
<form method="post">
        <label>
                <div>title</div>
                <input type="text" name="title">
        </label>

        <label>
                <div>art</div>
                <textarea name="art"></textarea>
        </label>

        <input type="submit">
</form>
```

I have also added the submit button. The next thing we need to do is to add the form handling, by adding the post method in our application.

Form Handling

The first thing that we need the handler to do is to get the user input strings title and art from the form:

```
def post(self):
    title = self.request.get("title")
    art = self.request.get("art")
```

Now we add some error handling as follows:

```
    if title and art:
        self.write("thanks!")
    else:
        error = "we need both a title and some artwork!"
        self.render("front.html", error = error)
```

If the user has entered both a title and some artwork, then we will display a simple “thanks!” message for now.

If either the title or the artwork are missing then we want to re-render the page with the error message "we need both a title and some artwork!". The message is stored in the variable error, and so we now need to add a place for this error in our template.

We will add the error at the bottom of the form:

```
<form method="post">
    <label>
        <div>title</div>
        <input type="text" name="title">
    </label>

    <label>
        <div>art</div>
        <textarea name="art"></textarea>
    </label>

    <div class="error">{{error}}</div>

    <input type="submit">
</form>
```

We called the div with the error message **class=”error”** because we intend to style this later using css styles. The double curly-brackets, **{{}}**, are part of the template language and will render any variable in place.

We now have a functioning form, but we can make it better. First, we are going to be calling self.render("front.html", error = error) from both post and get, so let's make it a function in it's own right called render_front()

```
    def render_front(self, title="", art="", error=""):
        self.render("front.html", title=title, art=art, error = error)
```

We can then modify the post and get functions to use this new function, and our MainPage handler becomes:

```
class MainPage(Handler):
    def render_front(self, title="", art="", error=""):
        self.render("front.html", title=title, art=art, error = error)

    def get(self):
        self.render_front()

    def post(self):
        title = self.request.get("title")
        art = self.request.get("art")

        if title and art:
            self.write("thanks!")
        else:
            error = "we need both a title and some artwork!"
            self.render_front(error = error)
```

Now, we just need to add the variables for title and art to our form in front.html to preserve the user input if they forget to complete one of the fields like so:

```
    <form method="post">
        <label>
            <div>title</div>
            <input type="text" name="title" value="{{title}}">
        </label>

        <label>
            <div>art</div>
            <textarea name="art">{{art}}</textarea>
        </label>

        <div class="error">{{error}}</div>

        <input type="submit">

    </form>
```

And our form is ready. Or is it?

Form Handling Continued

If we run our application in a browser it does seem to work, but the user input still isn’t preserved. This is simply because we didn’t add the parameters title and art to our post function. Adding them as shown below makes the form work as intended:

```
def post(self):
    title = self.request.get("title")
    art = self.request.get("art")

    if title and art:
        self.write("thanks!")
    else:
        error = "we need both a title and some artwork!"
        self.render_front(title, art, error)
```

What about escaping risky HTML code if a user enters it into our form?

In fact, the form handles risky HTML perfectly. Try it for yourself. But we didn’t add any escape functions in our code, so how does the form handle these characters?

We have been using jinja as the template language to render the HTML in our application:

```
import os
import webapp2
import jinja2

from google.appengine.ext import db

template_dir = os.path.join(os.path.dirname(__file__), 'templates')
jinja_env = jinja2.Environment(loader = jinja2.FileSystemLoader(template_dir),
                               autoescape=True)
```

When we initiated the jinja environment we set the parameter autoescape=True, which will automatically escape anything we include from a variable.

Creating Entities

Now that we have our form, we can begin to add the database so that we can store the artwork submitted by our users.

The way to define an entity in Google App Engine is to define a class:

```
class Art(db.Model):
```

This inherits from **db.model** (which we imported near the top of the app).

Datastore Types

Google App Engine has a number of different options for the types of the properties of an entity (similar to the column types we saw earlier). Some of the most popular are:

Integer:	for storing integers
Float:	for storing floating-point numbers
String:	for storing strings
Date:	for storing dates
Time:	for storing times
DateTime:	for storing dates and times
Email:	stores emails
Link:	stores links
PostalAdd:	sores postal addresses

Datastore Types Quiz

We are going to use three properties in our entity:

1. Title
2. Art
3. Created (when the artwork was added)

What types should we assign to each of these properties?

Creating Entities Continued

In fact, we are actually going to use the type **Text** for the Art property.

The difference between **String** and **Text** is that a string must be under 500 characters and can be indexed, Text can be more than 500 characters, but cannot be indexed.

So, we now add the property types to our entity. The general format is:

```
property_name = db.TypeProperty()
```

So our entity will be:

```
class Art(db.Model):
    title = db.StringProperty(required = True)
    art = db.TextProperty(required = True)
    created = db.DateTimeProperty(auto_now_add = True)
```

The parameter **required=True** means that the property is required in the database. If we try to add ASCII art to our database without a title (or without art!) we will get an exception from Python.

The parameter **auto_now_add=True** automatically adds the current date and time to the property.

Working With Entities

We now need to extend the success case to add the new artwork.

First, we create a new piece of art a, passing in the **title** and **art** variables. We don’t need to add anything for created as the property is automatically created. We can then use a.put() store the new piece of art to our database.

Finally, we redirect to the front page, "/", to avoid the annoying reload message:

```
if title and art:
    a = Art(title = title, art = art)
    a.put()

    self.redirect("/")
```

This should now be working, but we have now way of knowing until we add the last feature to draw the artwork below the form.

Running Queries

We want to look up all the artwork in our database so that we can display it on our front page. We want to do this every time we render the front page so that we can display both the form and all the artwork.

To do this, we need to modify our **render_form()** function:

We add a GQL query using an expression of the form:

```
arts = db.GqlQuery("")
```

The actual GQL query goes between the quotes.

Running Queries Quiz

Write the SQL (or in this case, GQL) for the query we need to fetch all of the Arts from the database sorted by creation time ("created"), most recent first?

Now we are in a position to complete the **render_form()** function

```
def render_front(self, title="", art="", error=""):
        arts = db.GqlQuery("SELECT * FROM Art "
                                    "ORDER BY created DESC")

        self.render("front.html", title=title, art=art, error = error, arts = arts)
```

We have added the GQL query to look up all the artwork in our database and passed the variable **arts** into the **render()** function. Now, all that is left is to modify our template to use this arts variable.

First, we are going to add a horizontal rule tag, <hr> below the form, as a visual separator between our form and the users' artwork.

Next, we are going to create a loop in the template by using Python code embedded in the HTML.

The way to embed Python code when you are using jinja is to use a construct like this:

```
{% code %}
```

To go through all the artwork in arts we need the Python construct:

```
{% for art in arts %}
```

Next we will construct an HTML structure to display the artworks and their titles:

```
<div class="art">
        <div class="art-title">{{art.title}}</div>
        <pre class="art-body">{{art.art}}</pre>
</div>
```

So, we have created a division which we have called class=”art” (for when we apply styles to our HTML), within which there are two further structures for the title and the artwork itself.

The title we are going to hold in another <div>, this time the class=”art-title”. The variable **art.title** is shown in the double curly-brackets.

For the actual artwork, we are going to use a <pre> element (having the class=”art-body”) as the container. The <pre> element represents a block of preformatted text which preserves the whitespace within it. Again, the variable **art.art** is shown inside double curly-brackets.

This should display our ASCII art as it was intended to be shown.

Finally, we need to indicate that the loop is complete. We do this with the statement:

```
{% endfor %}
```

The body of our template now looks like this:

```
<body>
	<h1>/ascii/</h1>

	<form method="post">
		<label>
			<div>title</div>
			<input type="text" name="title" value="{{title}}">
		</label>

		<label>
			<div>art</div>
			<textarea name="art">{{art}}</textarea>
		</label>

		<div class="error">{{error}}</div>

		<input type="submit">
	</form>

	<hr>

	{% for art in arts %}
		<div class="art">
			<div class="art-title">{{art.title}}</div>
			<pre class="art-body">{{art.art}}</pre>
		</div>
	{% endfor %}

</body>
```

Styling

We’ve been including classes with out HTML template, and we’ve mentioned a few times that these can be used for styles which can improve the appearance of our web pages. Let’s take a look at some of these styles.

Styles can be defined using the <style> element:

```
<style type="text/css">
</style>
```

Within the style element, we can specify the characteristics for the various elements in our HTML code. As an example, we can define the default style for the body (i.e. between the <body> and </body> of our web page using the following:

```
body {
        font-family: sans-serif;
        width: 800px;
        margin: 0 auto;
        padding: 10px;
}
```

As you can see, we are able to specify the font-family (and we can also set the font size), the on-screen width, any margin and the padding that we want to apply.

Padding is space between the outside of the element and the contents.

The complete style specification that we are going to use for this page is shown below:

```
<style type="text/css">
        body {
                font-family: sans-serif;
                width: 800px;
                margin: 0 auto;
                padding: 10px;
        }
        .error {
                color: red;
        }
        label {
                display: block;
                font-size: 20px;
        }
        input[type=text] {
                width: 400px;
                font-size: 20px;
                padding: 2px;
        }
        textarea {
                width: 400px;
                height: 200px;
                font-size: 17px;
                font-family: monospace;
        }
        input[type=submit] {
                font-size: 24px;
        }
        hr {
                margin: 20px auto;
        }
        .art + .art {
                margin-top: 20px;
        }
        .art-title {
                font-weight: bold;
                font-size: 20px;
        }
        .art-body {
                margin: 0;
                font-size: 17px;
        }
</style>
```

Answers

What is a Database? Quiz

- **A program that stores and retrieves data**
- **The machine running that program**
- **A group of machines working together to store/retrieve data.**

Implementing Tables In Python Quiz

```
def query():
    for l in links:
        if l.id == 15:
            return l.votes
```

Querying Quiz

```
def query():
    submissions = []
    for l in links:
        if l.submitter_id == 62443:
            submissions.append(l)
    submissions.sort(key = lambda x: x.submitted_time)
    return submissions
```

Types Of Databases Quiz

The answer is none of them. There may be a most popular database, but each has it's place, and there is definitely no "best".

Databases In Python Quiz

```
def query():
    c = db.execute("select * from links where id = 2")

    link = Link(*c.fetchone())
    return link.votes

print query()
```

Advanced SQL In Python Quiz

```
def query():
    c = db.execute("select * from links where submitter_id = 62443 and votes > 1000")

    link = Link(*c.fetchone())
    return link.id

print query()
```

Order By Quiz

```
def query():
    results = []
    c = db.execute("select * from links where submitter_id = 62443 order by
submitted_time asc")
    for link_tuple in c:
        link = Link(*link_tuple)
        results.append(link.id)
    return results

print query()
```

Querying Links Quiz

```
def link_by_id(link_id):
    for l in links:
        if l.id == link_id:
            return l
```

Using Dictionaries As Indexes Quiz

```
def build_link_index():
    index = {}
    for l in links:
        index[l.id] = l
    return index
```

Lookup Quiz

```
def add_new_link(link):
    links.append(link)
    link_index[link.id] = link
```

Advantages Of Indexes Quiz

- **Indexes increase the speed of database reads.**
- Indexes increase the speed of database inserts.

Scaling Databases Quiz

- Get a faster machine.
- **Replicate the database.**
- Store less data.
- Press the turbo button.

Growing Databases Quiz

- Replicate the database.
- Get a bigger hard disc.
- **Shard the database.**
- Store less data.

Acid Quiz

- Atomicity
- Consistency
- Isolation
- Durability

Automatic Sharding And Replication Quiz

- Yes.
- **No.**

Creating The Form Quiz

- <input type=”text”>
- **<textarea>**
- <pre>
- <input type=”password”>

Datastore Types Quiz

We are going to use three properties in our entity:

1. Title: **String**
2. Art: **String**
3. Created (when the artwork was added): **DateTime**

In fact, we are actually going to use the type **Text** for the Art property. The difference between String and Text is that a string must be under 500 characters and can be indexed, Text can be more than 500 characters, but cannot be indexed.

Running Queries Quiz

select * from Art order by created desc