Databases,
or DBMS in half a lecture

Hunter Jorgensen

Background

- 1950’s: First computers were basically giant calculators, and data was considered leftover
- 1960: As computers become commercially available, Charles Bachman created the Integrated Database System, the first DBMS
- 1970: E.F. Codd IBM develops the “relational” database at IBM
- 1974: IBM develops SQL (Structured Query Language), which allowed develops to retrieve and manipulate data. This ultimately replaced other querying languages, such as COBOL and QUEL.

Terminology

Key (primary or secondary): a attribute which can identify a set of data (i.e., unique ID's, phone number, usernames, etc.)

Table: a collection of related data (User table, food table, pokemon table, etc.)

Row: a single entry in a table (i.e., one person's user data)

Column: a single attribute on a table (user id, first name, last name, etc.)

Query: looking up a row by key (similar using index/key on array/map; fast and cheap)

Scanning: searching the entire database for row based on condition (similar to iterating over a list/array; very resource intensive)

New to Databases?

For those new to databases, you can think of databases (especially the one we use in this class) as a really big and effective Excel document. For instance, a “table” is a sheet, rows and columns are similar to their Excel counterpart, etc. There are some differences (keys, queries, etc.) but we can ignore that for now.

An example of this might be if you have an Excel spreadsheet of your friend’s addresses. If you were looking to update an address of a friend in a specific country, you can scroll down through your list until you find that country name in the “country” column, then proceed to fix the address as you see fit.

RDMS, in a few slides!

(this is all for historical context, but not terribly important for our class!)

Relational databases have 3 ways to query data:

- Using a primary key
- Moving relationships, or sets, to one record from another
- Scanning all records in sequential order

Relational databases and SQL are so intertwined now that you can use the terms interchangeably.

RDMS

Let’s say we have 2 tables (i.e., collections of data, usually with fixed attributes or ‘columns’) where we track users and track those users’ favorite foods:

Id	Username	First Name
12345	hunter167	Hunter
54321	aaron227	Aaron

ID	User ID	Food	Flavor
1	12345	Banana	
2	12345	Cookie	Chocolate Chip
3	54321	Sandwich	Veggie

3 ways to find foods in the table:

- Know ID of row (very fast)
- Know user, and search via relationship (i.e., join; slower)
- Iterate through entire table and search based on attributes (food == ‘Banana’; very slow)

NoSQL

Originally NoSQL referred to databases that were relational but didn't use SQL. However, by 2009 this changed and referred to non-relational databases.

"Not only" SQL gained popularity in the late 90's/early 2000's as companies started seeing more unstructured and large data sets. There are many kinds of NoSQL databases (column stores, graph databases), but we'll focus on key/value and document storage.

NoSQL typically have no relationships that exist across tables. Instead, NoSQL usually access data through primary keys or scanning (i.e., searching line by line)

Benefits of NoSQL

Both NoSQL and SQL DB’s have strengths and weakness. NoSQL gained popularity because of:

- Higher scalability
- A distributed computing system
- Lower costs
- A flexible schema
- Can maintain unstructured/semi-structured data
- No complex relationship

Cons:

NoSQL DB’s typically are more resources intensive and are still a relatively new technology

Summary: NoSQL works well with the ‘single view’ page of a website

Scaling Up vs Out

One important comparison: SQL/NoSQL are often described us scaling up vs out. What this means is:

- SQL/Scaling Up: is better at handling more tables and more relationships between those tables; but if tables get too big, then this can have negative impact
- NoSQL/Scaling: is optimized for having more data on individual tables (almost no performance impact); not so good at referencing other tables

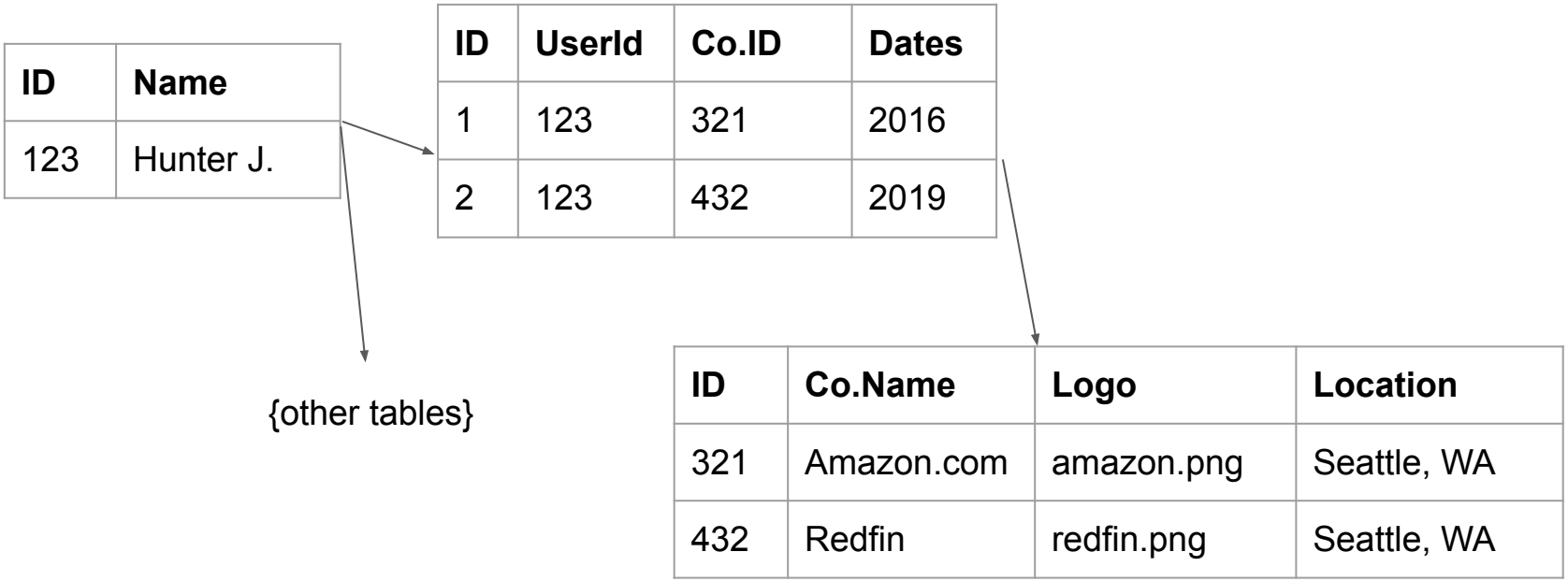
Thought Lab: LinkedIn

LinkedIn is a social media website aimed at connecting professionals. Every user's profile page lists their jobs, education, skills, and more.

How might this data be represented in a relational database/SQL? How might it be represented in a NoSQL/non-relational database?

What are the benefits of each of these?

LinkedIn: SQL



ID	Name
123	Hunter J.

ID	UserId	Co.ID	Dates
1	123	321	2016
2	123	432	2019

{other tables}

ID	Co.Name	Logo	Location
321	Amazon.com	amazon.png	Seattle, WA
432	Redfin	redfin.png	Seattle, WA

LinkedIn: NoSQL

ID	UserId	Company Name	Dates	Logo
1	123	Amazon.com	2016	amazon.png
2	123	Redfin	2019	redfin.jpeg

*The idea here is that essentially everything needed to be shown exists on a single table

History of MongoDB

MongoDB is a NoSQL database that is considered a document store, as it uses JSON-like documents.

MongoDB is an open-source database created in 2009.

Note: you'll want to make sure that you MongoDB instance is running before doing any development.

Note on Installing MongoDB

Catalina on Mac has broken the traditional path of installing and running MongoDB for many people. I provide 2 approaches, and you may need to try both!

For either approach, you'll want to create a folder to store the actual data. You should NOT use a root folder. I recommend:

mkdir ~/Documents/data

Installing MongoDB (Mac/Linux)

You will need **brew** to install MongoDB for most UNIX-based operating systems (it is usually auto installed):

/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"

Brew is a package manager and makes it easy to install new programs, technologies, libraries, etc. on your machine.

brew tap mongodb/brew

brew install mongodb-community

mkdir /PATH/TO/DATA/DIRECTORY (note: do not write “/PATH/TO/DATA/DIRECTORY”, rather create a folder location where you would want to store this data)

mongod --dbpath /PATH/TO/DATA/DIRECTORY

Installing MongoDB (PC)

You'll need to manually download MongoDB here.

Extract and install.

Right click Command Prompt to open in administrative mode:

"C:\Program Files\MongoDB\Server\4.4\bin\mongod.exe" --dbpath /PATH/TO/DATA/DIRECTORY

This command will start your MongoDB instance

If you're using WSL

Mongo Terminal

MongoDB comes with a simple terminal interface to help you get comfortable or debug your data.

mongo (or **./mongo**, if you're on PC)

Mongo Terminal

show collections

db.<*collection*>.insertOne({...})

db.<*collection*>.insertMany({...})

db.<*collection*>.find({...})

db.<*collection*>.updateOne({...}, {$set:{...}})

db.<*collection*>.deleteOne({...})

db.<*collection*>.drop();

Mongo IDs

You'll notice that Mongo will add an ObjectId(...) into attribute _id into entries that you don't do that for. If you do a find with the _id field, Mongo will perform a *query*. Without an _id, Mongo will perform a *scan*.

Query: an immediate lookup of a row based on an index. This is usually the fastest and quickest way to look up information, but can be costly for the database to maintain too many indices.

Scan: a one at a time search of all the entries in the database. This is slower and only good to do occasionally. It is also costly, since the database needs to bring all the items into memory.

You can also insert your own indices so that Mongo will query instead of scan.

Remember: this is NOT a database class so performance here is not a high priority. It is better to have your app slowly but correctly, rather than strive for efficiency but never reach it

Accessing MongoDB

Typically, when you access a database through code you have 2 options:

- Access the database via its native syntax (SQL, Mongo Shell, etc.) This faster, but can sometimes be hard to read and write.
- Using an Object Data Model ("ODM") or an Object Relational Model ("ORM"). An ODM/ORM represents the data as JSON, and maps that to the database. This approach is slower, but can make it easier to read and often times provides additional functionality.

Introducing Mongoose!

Mongoose is an ORM that is specifically designed to work with MongoDB. It works particularly well in an async environment.

Mongoose allows us to create schema, which allows us to define what our ‘documents’ in MongoDB will look like. It can also provide additional functionality that simplifies how we interact with our database.

```
npm install -s mongoose
```

Terminology

Schema: This is a Mongoose specific term. It is a document data structure (or shape of the document) that is enforced via the application layer and outside of the database proper.

Model: constructors that take a schema and create an instance of a document equivalent to records in a database. This converts an object in memory into something usable to the database and a row in the database into something that your code can interact with.

Connecting to Our DB

First, make sure your local instance of MongoDB is running (`mongod -dbpath='...'`, or however you do so.) Then we will add some code to our app.js or server.js file:

Code in Question

Note: that last line will throw an error if the code isn't working. You can change the **mongoEndpoint** URL to try this.

Pokemon Schema

Let’s create a schema for our core concept, our pokemon. Looking in **pokemon.js**, we can see that Pokemon have at least some attributes: id, owner, name, color, and health.

Let’s create a simple schema for our Pokemons!

Demo

Schema Info

Schemas allow for a lot of functionality. Take note of all the types there are and some of the features:

```
const schema = new Schema({
  name: String,
  binary: Buffer,
  isTrue: Boolean,
  created: { type: Date, default: Date.now() },
  age: { type: Number, min: 0, max: 65, required: true },
  mixed: Schema.Types.Mixed,
  _someId: Schema.Types.ObjectId,
  array: [],
  ofString: [String], // Array of a given type
  map: { stuff: { type: String, lowercase: true, trim: true } }, // Complex objects/nested data
})
```

What are some that work well for our Pokemon use case?

Connecting A Schema to a Model

Now that we have designed our Schema, Mongoose uses a model to map between a JSON to a MongoDB document, and vice versa. Mongoose Model also provides us with some essential API functionality to query, find, update and delete code in our database.

Code

Note: Model.create() and Model.*someQuery(...)*.exec() both return promises. You can chain the '.then(...)' at the end of it or even try to get async/await to work.

Putting it All Together

Now that we've figured out how our code is going to interface with MongoDB, we can go ahead remove our previous list and use an actual database.

Demo

Note: notice that this code separates out the Model/Schema from the actual API logic. This is to maintain our MVC model.

One More Example: Users

We can apply all of the same work to users as well!

Users Demo