Introduction to Databases

Fall 2021

AGENDA

- Why databases?
- Structure of the course
- Thinking about storage

WHY DATABASES?

- Applications need to manage data
  - internal data structures for storing data
  - arrays, hashmaps, dataframes, ...
- BUT how do you persist the data structures?
  - so that they survive an application stopping and restarting
- BUT how do you share the data structures?
  - maybe more than one user needs to interact with the data at the same time
- BUT how do you manage data that's larger than memory?
  - historical weather data
  - system logs
  - large machine learning training sets

Databases

- Pulling data out of the application and persist to disk
  - DIY in the early days
  - tends to be slow and error-prone to implement by hand
  - tooling: dbm (unix) - key/value pairs store
- Provide a convenient interface to the data
- Efficient storage and retrieval mechanism
- Run as a server accessed by clients
- Support concurrent access
- Support replication
- Now of course available in the cloud, managed by others

- Every database supports a data model (what data can be represented)

Relational databases

Data represented as rows of primitive values in tables

- Original data model
- Massive convergence to a unified model in the 70s
    - Postgres, MySQL, MariaDB, Oracle, Microsoft SQL Server, ...
- Standardized query language: SQL
- Requires pre-defined data schemas (think static typing)
- ACID properties: atomicity, consistency, isolation, durability

Non-relational databases

Also known as NoSQL databases — any database that is not relational:

- Document databases: MongoDB, CouchDB, Firebase Realtime…
- Key-value stores: Cassandra, Redis, DynamoDB, HBase, ...
- Graph databases: Neo4j, ...

They relax the structural and consistency restrictions of relational databases

- No need for schema definitions (think dynamic typing)
- More resilient replication
- What's the catch?

Looks complicated...

As a casual programmer, beyond figuring out whether you'll be storing arrays (relational) or storing JSON objects (non-relational), it mostly doesn't matter which database you use for your projects

They're all pretty much equivalent for casual use

There are big differences, but they don't really appear until you have millions of data points, high traffic, or extensive database replication to reduce latency

- Most likely, an architect at your company will have chosen for you

The annoying corner case where it matters

A startup, starting small, growing huge fast

- The database choices made early often won't scale
- Need for database migration
- Made easier by abstracting the interface to the DB as much as possible
    - DAL = data access layer, which interacts with the DB directly
    - interact with the DAL via REST calls
    - migrating databases requires changing the DAL, not the client
- There are other approaches (ORMs, etc)

STRUCTURE OF THE COURSE

We're going to study the various classes of databases that exist

- relational vs non-relational databases
- learn how to interact with them
- learn the key features of each class
    - data structuring
    - query languages
    - transactions, concurrency
    - replication, consistency
- learn a little bit how they work inside
    - this is not a database theory course
    - but enough to know you don't want to roll-your-own

STRUCTURE OF THE COURSE

- Lecture-based — I like talking
    - I also like to demo code, but there won't be as much
    - Textbook *Principles of Database Management*
- Homeworks
    - Python code to build up an understanding of the concepts we study
    - hands-on experience with databases: install them, populate them, query them
    - work in pairs (choose your team)
- Final project
    - choose project mid-semester
    - can be a deeper exploration of a topic we talked about, or an introduction to a topic we skipped
    - final presentations during events week
- Meet on Mondays, office hours probably after class
    - No ninja, but I'm available for questions, and we can find tutors if you need one
    - Speak up, reach out — as usual, the more interactive it is the more fun it is

THINKING ABOUT STORAGE

Plenty of data structures for storing data in Python

- arrays
- dictionaries
- dataframes (pandas)

Most of the issues we have about choosing + structuring data in databases already arise when choosing an internal data structure for storing stuff

CRUD operations

CRUD operations are primitive operations for any storage-based data structure

- Create: create an entry in the data structure
- Read: find entries in the data structure
- Update: modify entries in the structure
- Delete: delete entries in the structure

Big-O notation

Remember: O(-) notation

- Roughly the number of "primitive" steps performed by an operation as a function of the size of the input
- Up to a constant, and keeping only the highest order terms
- E.g., $O(1)$ = constant time
  $O(n)$ = linear in the size *n* of the input
  $O(n^2)$ = quadratic in the size *n* of the input
  $O(2^n)$ = exponential in the size *n* of the input

Example: books

A book has a title, an author, a publisher, a year, a count of pages, an ISBN

How do you represent collections of books in Python?

We are going to analyze the efficiency of some of the CRUD operations for a few obvious representations

- Collection as array of books
- Collection as sorted array of books
- Collection as dictionary of books

(Read operation: find entries for a particular value of a book field)

Collection as array of books

Probably the simplest representation:

$$\texttt{BOOKS = [}book_1\texttt{, }book_2\texttt{, }book_3\texttt{, …]}$$

Create operation: append to the end of the (dynamic) array O(1)

Read operation: scan the array for all entries matching the value O(n)

Collection as sorted array of books

We can do a bit better if we keep the array sorted — need a sorting key

$\texttt{BOOKS = [}\mathit{book}_1\texttt{, }\mathit{book}_2\texttt{, }\mathit{book}_3\texttt{, …]}$

Create operation: insert into a sorted array by binary search O(log n)

Read operation:

- Find by sorting key value — if one entry per value: O(log n)
  - binary search to find the entry
- Find by non-sorting key value? O(n)
  - scan the array

- Can you sort on two keys independently?
- What if you have more than one entry per key value?

Binary Search

Search for 55

1	3	6	10	15	21	28	36	45	55	66	78

Binary Search

Search for 55

1	3	6	10	15	21	28	36	45	55	66	78

Binary Search

Search for 55

1	3	6	10	15	21	28	36	45	55	66	78

Binary Search

Search for 55

1	3	6	10	15	21	28	36	45	55	66	78

Binary Search

Search for 55

1	3	6	10	15	21	28	36	45	55	66	78

Binary Search

Search for 55

1	3	6	10	15	21	28	36	45	55	66	78

Binary Search

Search for 55

1	3	6	10	15	21	28	36	45	55	66	78

Binary Search

Search for 55

1	3	6	10	15	21	28	36	45	55	66	78

Collection as dictionary of books

Python has a primitive data structure dictionary implemented as a hash map

Choose a hashing key — assume one entry per hashing key

BOOKS = { $isbn_1$: $book_1$, $isbn_2$: $book_2$, $isbn_3$: $book_3$, …}

Create operation: hashmap update O(1)

Read operation:

- Find by hashing key value: lookup key value O(1)
- Find by non-hashing key value: scan the values O(n) *

If more than one value per hashing key? Use arrays of entries per key value