Databases

This is CS50. Harvard University. Fall 2015.

Cheng Gong

Table of Contents

Databases

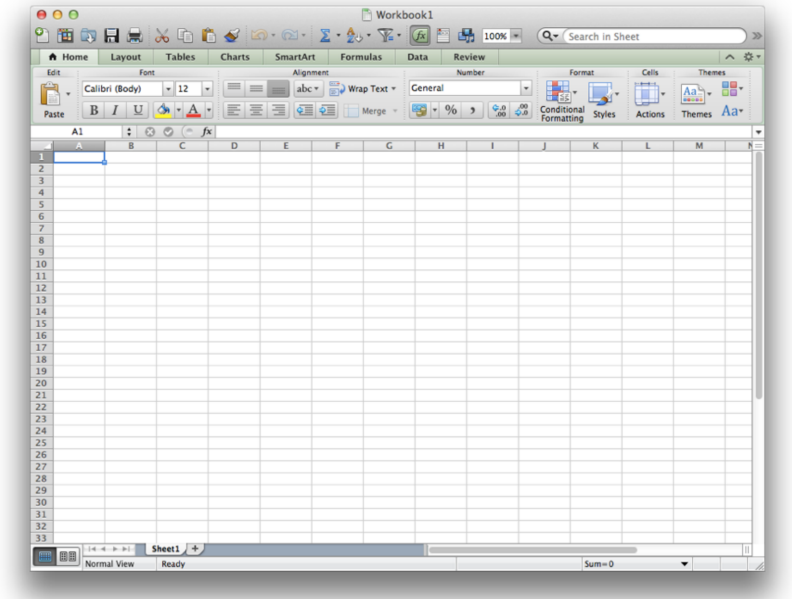
- **Back-end servers** include web servers running software like Apache (httpd), which is a program that serves web pages in response to requests, IIS (Internet Information Server), or Nginx.
- Those web servers are behind one or more **front-end** servers like load balancers that pass messages between the public and the back-end servers.
- We might have two such load balancers, with one running as the primary server and the other as a "hot spare" that's ready to take on the IP address of the primary load balancer if it stops working.
- But the back-end web servers need to store information from users in some shared place, since otherwise they would need to spend a lot of time copying that information back and forth.
- Having the one server saving the shared state introduces a single point of failure again, but the tradeoff for saving money and effort in redundancy at that level might be worth the risk of lost business during downtime.
- So the way that last server saves the shared state more permanently is the topic for today.
- We might have data like transaction data, order data, etc. that we want to keep safely. So we might have multiple database servers that keep copies or segments of data.
- For example, the very first version of Facebook, that allowed students from just a few college campuses, might have kept user data from each campus on a separate database or server. This method of separating data is called **sharding**. This helps us naturally load balance, but searching over all of our data is not as easy, and eventually

we might have a feature that requires us to combine our data. And eventually, we might have too many buckets for our data, to maintain each of them separately.

- **Horizontal scaling** is increasing the number of servers that we have that do the same thing, for example buying more web servers to handle more visitors. **Vertical scaling**, on the other hand, is increasing the power of a single server by upgrading its hardware, but there is a limit to how far that can get us.
- And when we have too much data to combine, we might separate them by function or feature, so that we are able to find them more logically.
- But assuming we can fit all our data on the same (large) database server, we might duplicate it to another server to handle more users. Another way to do this would be to have one large server for writes (saving data) but many smaller servers for reads (accessing data). And this might make sense if our website has users reading information (like a News Feed) more often than they write (posting new statuses).
- One characteristic we might want is **high availability**, which can be achieved with having two servers **replicating**, or duplicating data, to each other (with software) and both being allowed to read and write.
- We might also have two databases that write data, and duplicate data to two databases in the middle, each of which duplicate data again to two servers, such that we have a tree-like structure for effective data replication.
- If this process takes a long period of time, users might be reading outdated information from the read database servers.
- Now that we've looked at the high-level overview of setting up our database servers, we'll look at some lower-level details. But before that, we take a look at this article[1] about WordPress, a blogging platform, turning on HTTPS encryption for its sites. Though this might be an upside, it might also break websites who include images with URLs that are normal HTTP, since browsers generally disallow mixing protocols (since including HTTP requests within an HTTPS means that parts of the page won't be encrypted, and therefore not as secure).
- So **CRUD** is an acronym for the operations that a typical database needs: **create**, **read**, **update**, and **delete**.
- **SQL**, Structured Query Language, is popularly used for database software and has the matching operations of **INSERT**, **SELECT**, **UPDATE**, and **DELETE**.

[1] http://techcrunch.com/2016/04/08/wordpress-com-turns-on-https-encryption-for-all-websites/

- We can think of **databases** as like a workbook in Excel or other spreadsheet software:



- A **table** in a database is like a worksheet within one of these files, with rows and columns.
- To represent a customer, for example, we might want their:

  # email

  # name (first and last)

  # unique identifier

  # postal address

  # credit card info

  # age

  # password

  # transactions

  # username

  # phone

- So we might create this in Excel with column headers and typing in our data without much consideration. But in databases, where we might not have hundreds or thousands of rows but many, many more, we need to be more careful about how we store our data.

- Going back to our list of information, we might want to think about the **type** and **size** of data we want for each **field**. This will help our database store and search for things more efficiently. The types we could have include:

  # CHAR

    # Some characters of exactly a fixed length.

  # VARCHAR

    # Some character of up to some length.

  # INT

    # An integer of 32-bits (or about 4 billion values).

  # BIGINT

    # An integer of 64-bits, which is much larger, but now we need to allocate more space in case we need to hold that many bits.

  # FLOAT

    # An approximated decimal number.

  # DECIMAL

    # Since financial markets and other applications might need the high precision, there is a special data type that stores decimal numbers exactly, using a specific number of bits to store each part of the number and combining them in software later.

  # DATE

    # Stored in the format of YYYY-MM-DD, since it allows for easy sortability.

  # TIME

    # Stored as HH:MM:SS.

  # DATETIME

    # A combination of DATE and TIME.

  # ENUM

  # …

- So for our customer database, we might want:

email - VARCHAR - 255

    # As a database designer, we'd have to choose some maximum length, and historically 255 was the maximum value, so that tends to be a go-to value.

name (first and last) - VARCHAR - 255

unique identifier - INT

    # Someone might have multiple emails, or even eventually split our data into multiple tables but be able to link them. And comparing ints are also much faster than entire strings of text.

postal address

    # This probably includes many fields, which allows us the opportunity to design our database in greater detail:

    # street - VARCHAR - 255

    # city - VARCHAR - 255

    # state - CHAR - 2

        - Assuming our customers are all in the US, we know that all the states have two-character fields. And setting this allows the database software to allocate exactly two bytes for each item in that column, which helps us with random access and jumping between rows.

        - Alternatively, there's another data type called an **ENUM**, a list of pre-specified strings that we choose from for each value.

    # postal code - CHAR - 5

        - We don't want to use an int here, because the leading 0s in a postal code would be lost!

credit card info

age - DATE

    # We wouldn't want to store ages as an int, since we wouldn't know when to increase them, but a date for their birthday would work.

password

    # We would most likely store our passwords as hashes, so the value actually stored is hard to reverse.
  # transactions
  # username
  # phone

- Choosing the right types help us with validating data and optimizing future access.
- Another major feature of databases is **indexes**, which just uses fancy data structures to optimize the layout of the data on disk and memory.
- We can designate some column as **PRIMARY**, which means that we will uniquely identify rows with this column and use it most frequently to do so.
- A column can be **UNIQUE**, too, but not be used to identify users. For example, we might require that usernames be unique in our example.
- An **INDEX** means that the database should optimize the column for searching or analysis. But the cost would be that insertions and deletions might be slower, since the database would have to rearrange our data each time.
- Going back to our example, we have an inefficiency where we store the city, state, and postal code of each of our customers, even though customers with the same postal code must live in the same city and state. So we might have one table that stores information about our users, and one table that stores geographic information, where postal codes are mapped to cities and states, so we don't need to store that information over and over again.
- The process of combining information from different tables is called **joining** them.
- Next time, we'll talk about some problems we might run into as we implement websites in HTML and CSS.