CS-310 Scalable Software Architectures

Lecture 08: Relational Databases

Steve Tarzia

Last Time: Load balancers

We have 2/3 of the *end-to-end* view of a basic scalable architecture!

(for *services*, at least)

- *Frontend:* Client connects to "the service" via a **load balancer**.
  - Really, the client is being directed to one of many copies of the service.
  - Global LBs (DNS and IP anycast) have no central bottlenecks.
  - Local LBs (Reverse Proxy or NAT) provide mid-level scaling and continuous operation *(health checks & rolling updates)*.

Today's topic

- *Services:* Implemented by thousands of clones.
  - If the code is **stateless** then any worker can equally handle any request.
- *Data Storage??*
  - The next big topic!

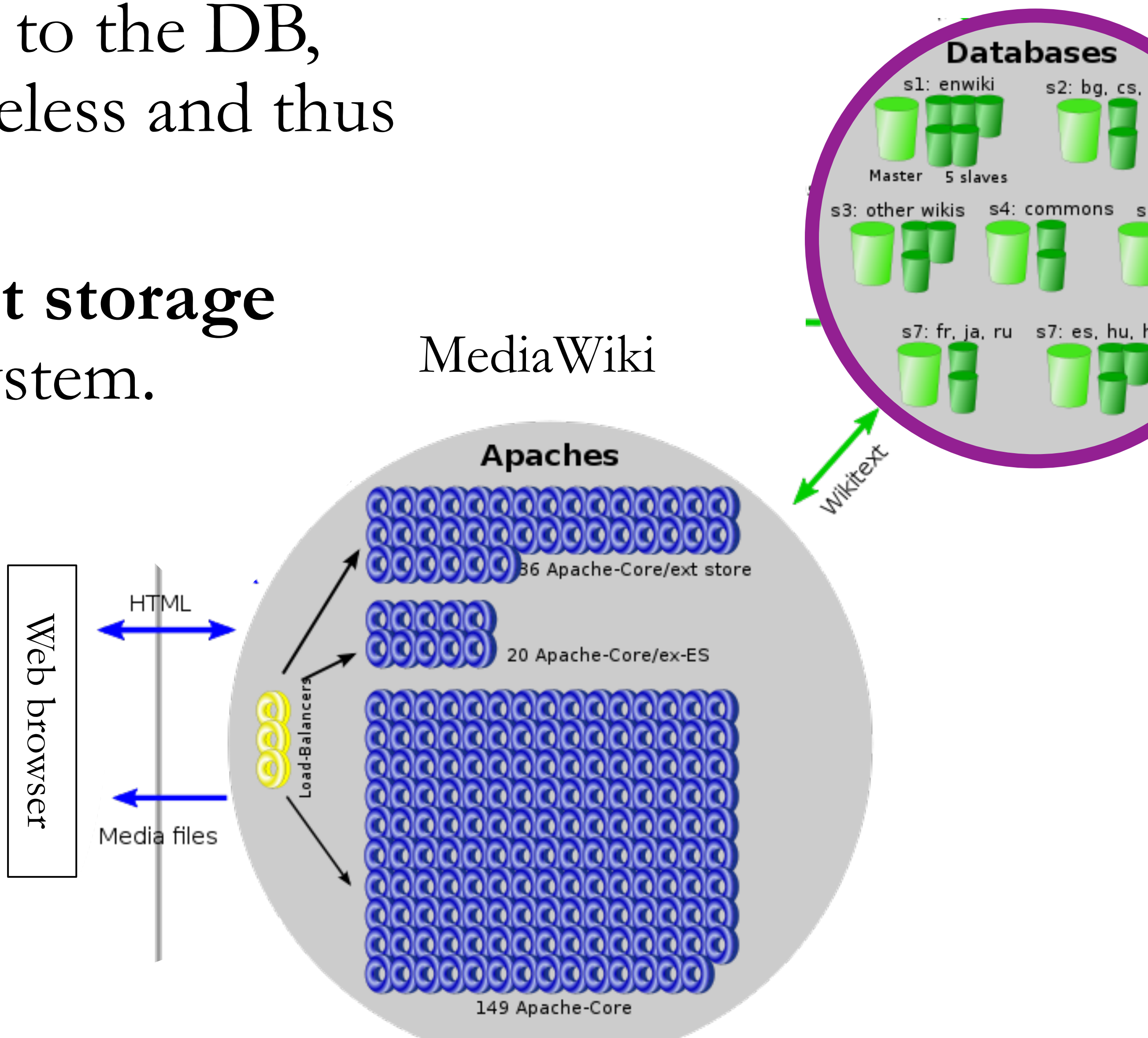
On to Databases

Will summarize the most important parts of CS-339 or CS-217 in one lecture.

Back to Wikipedia

- Recall that we pushed all app state to the DB, allowing MediaWiki app to be stateless and thus trivially parallelizable.
- Databases provide both **persistent storage** and **coordination** in large-scale system.

In general, these are our two biggest scalability challenges and both are the concerns of databases.



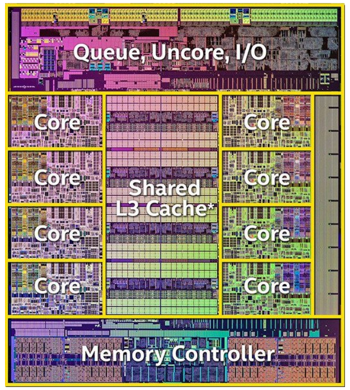
Computers have a hierarchy of storage

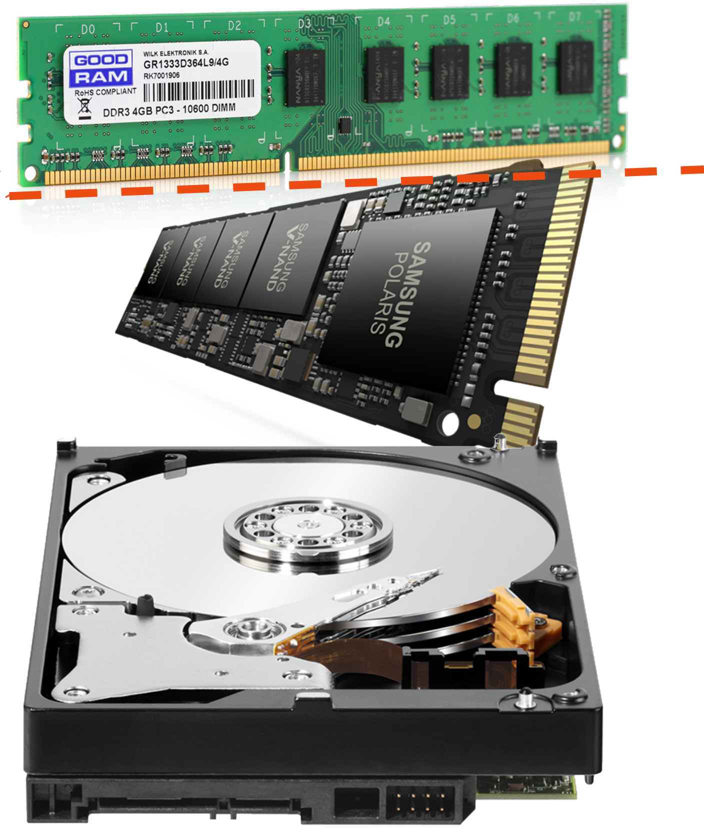
Larger, but slower

delay		*capacity*
0.3ns	CPU Registers	1 kB (kilobyte)
5ns	CPU Caches (L2)	16 MB
50ns	Random Access Memory (RAM)	16 GB
100µs	Flash Storage (SSD)	1 TB
5ms	Magnetic Disk	8 TB

volatile memory

persistent storage



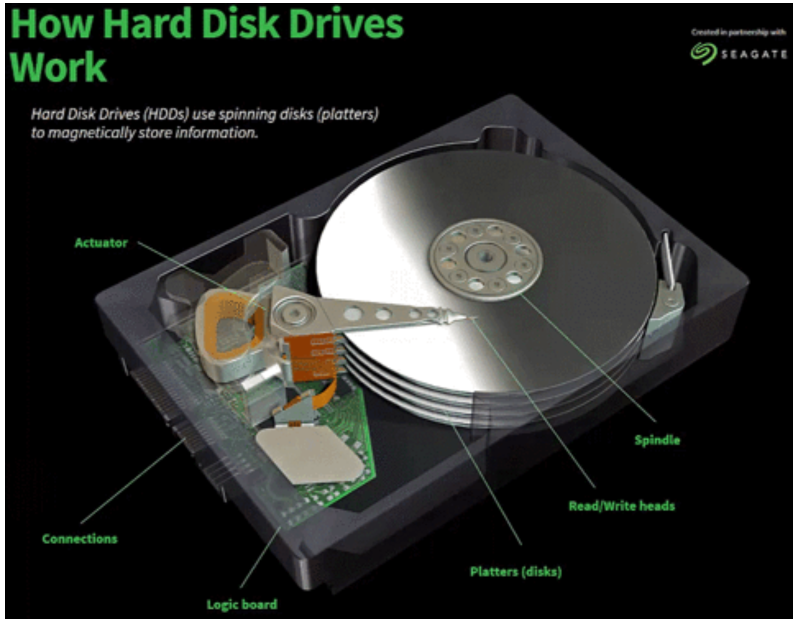


- Disk is about *ten billion* times larger than registers, but has about *ten million* times larger delay (latency).
- Goal is to work as much as possible in the top levels.
- Large, rarely-needed data is stored at the bottom level

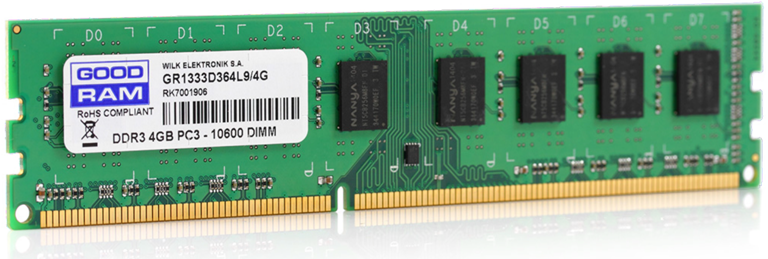
Storage has limited bandwidth

- All types of computer storage are limited to reading/writing just a small fraction at once.
- **Magnetic disks**:
  - The read/write head can read the charges on a tiny portion of the magnetic disk.
- **RAM (memory)**:
  - Memory and flash chips store lots of data, but only a few bytes can be transferred at once, because there are only a couple hundred electrical connections at the edge.
  - SSDs (flash) is similar, with even fewer electrical connections.

Magnetic disk's data can only be read at current location of the read/write head.



https://animagraffs.com/hard-disk-drive/



Just a couple hundred electrical connections at the edge of a RAM card.

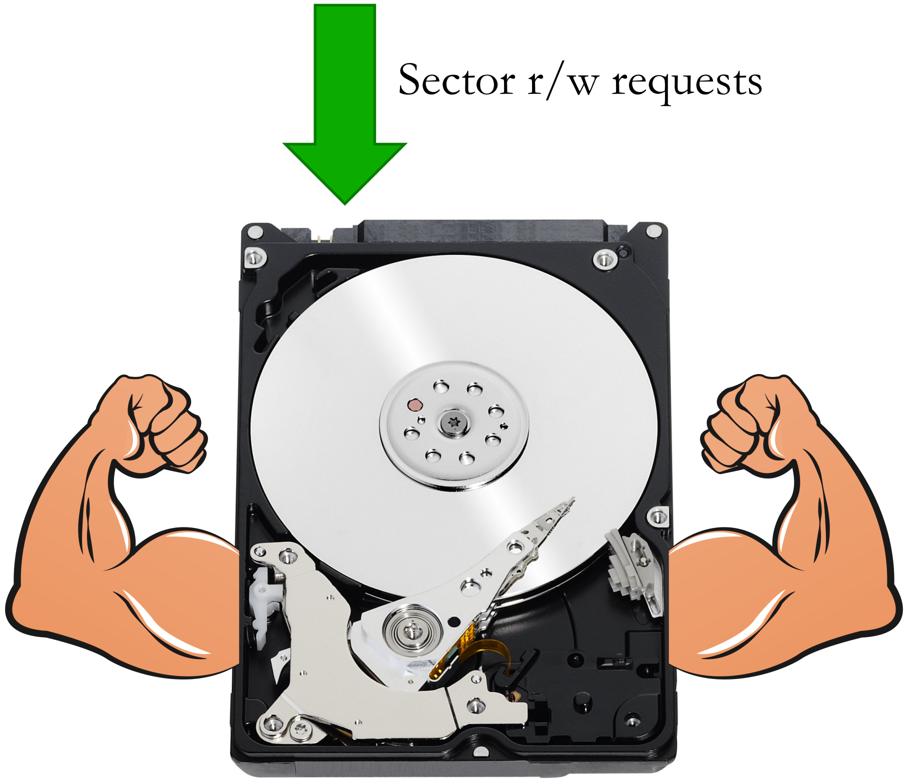
Redundant Array of Independent Disks (RAID)

- Disks have a few shortcomings:
  - ***Limited capacity*** (~12TB)
  - ***Limited throughput*** (~150MB/s)
  - ***Likelihood of failure*** (especially for magnetic/rotating disks)
- RAID uses multiple disks to solve these problems
  - Many different variations of RAID, depending on your budget and which of the above three problems are most important.
- Basic ideas are:
  - Increase ***capacity*** by making multiple disks available to store data.
  - Increase ***throughput*** by accessing data in *parallel* on multiple disks.
  - Reduce impact of a disk ***failure*** by storing data redundantly on multiple disks.
- Disk interface is very simple (just an array of sectors), so it's easy to create a **logical/virtual disk** made of sectors from multiple physical disks.

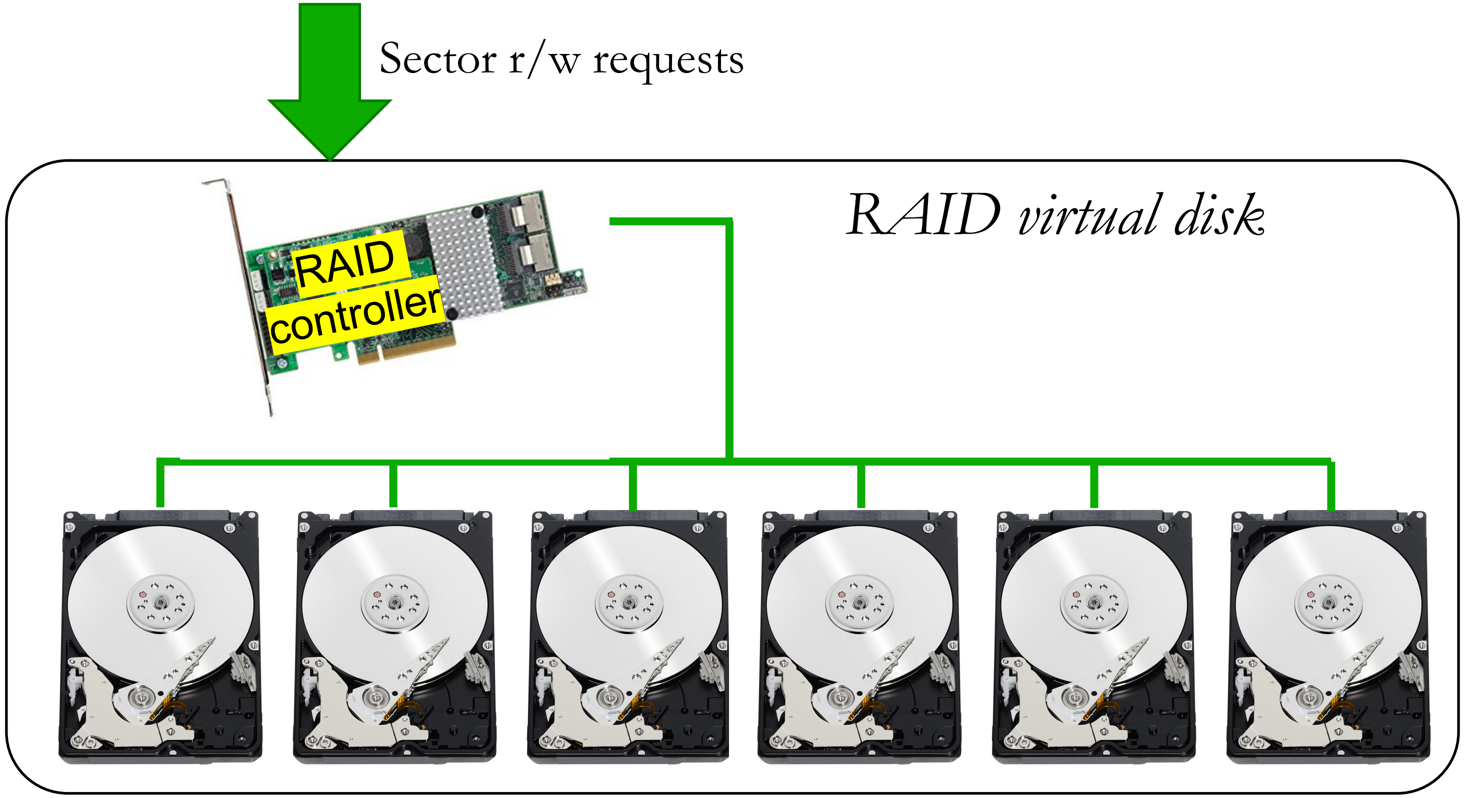
Basic idea of RAID

- Combine many disks to create one *superior* virtual disk.
- The RAID array provides the same interface as a single disk.

OS thinks it's dealing with this:



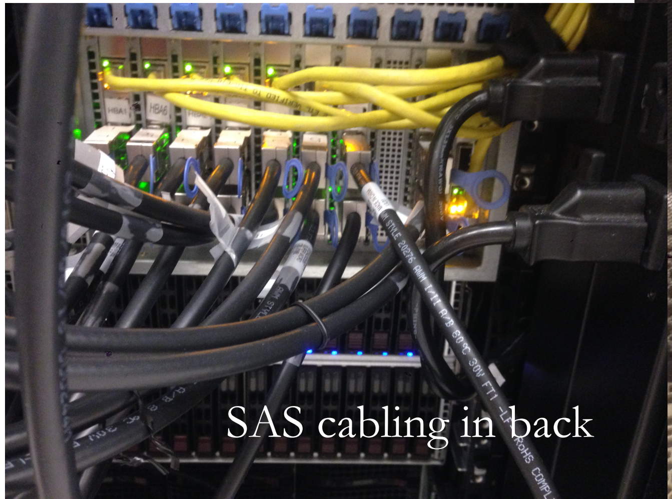
But it's just an illusion. The reality is:



A Database Server @ NU

- 264 fast (10k RPM) magnetic disks
- 56 slow (7200 RPM) magnetic disks (for backup)
- ~150 TB storage capacity
- Comprised of 6 physical chassis (boxes) in one big cabinet, about the size of a coat closet.

Front view



SAS cabling in back

Large database servers

- Capacity is practically unlimited, but a single computer has:
  - Limited compute power.
  - Limited I/O bandwidth (theoretical max of ~80 GB/s on PCI-express v4).
- This single-machine design can actually scale pretty well!

One big computer

Database SW running on one machine

Limited bandwidth to storage (80GB/s)

Many disks can be connected to the computer

- Relational (aka SQL) DBs use this design

Persistent (disk) storage has always been different

- Programming languages rarely deal with storage directly (except SQL).
  - Programmer must write code to move data from memory to disk.
  - Disks are slow, so making the programmer think before using it is OK.
  - But it's also really tedious to operate on large data sets, where data is constantly shuffled between disk and RAM.
- Normal way of accessing persistent storage:
  - Pass data into and out of **files** using system's open/read/write functions.

- **Databases** let the programmer use persistent storage w/out worrying about file-level transfers, with advanced performance optimizations.
- Usually interact with DB using a special query language (eg., SQL).

Relational DBs

- **Relational** databases store data in multiple **tables.**
- Each table has pre-defined set of columns (schema).
- Rows are added over time.
- Rows can refer to other rows through **foreign keys.** *(the arrows)*
  - "Philanthropy" is defined once, but referenced by the *industry* id 131 in many user rows.
- The final LinkedIn page may be generated by **JOIN**ing rows from many tables.

http://www.linkedin.com/in/williamhgates

Bill Gates

Greater Seattle Area | Philanthropy

Summary

Co-chair of the Bill & Melinda Gates Foundation. Chairman, Microsoft Corporation. Voracious reader. Avid traveler. Active blogger.

Experience

Co-chair • Bill & Melinda Gates Foundation
2000 – Present

Co-founder, Chairman • Microsoft
1975 – Present

Education

Harvard University
1973 – 1975

Lakeside School, Seattle

Contact Info

Blog: thegatesnotes.com
Twitter: @BillGates

users table

user_id	first_name	last_name	summary
251	Bill	Gates	Co-chair of … blogger.

… line wrapped

region_id	industry_id	photo_id
us:91	131	57817532

regions table

id	region_name
us:7	Greater Boston Area
us:91	Greater Seattle Area

industries table

id	industry_name
43	Financial Services
48	Construction
131	Philanthropy

positions table

id	user_id	job_title	organization
458	251	Co-chair	Bill & Melinda Gates F…
457	251	Co-founder, Chairman	Microsoft

education table

id	user_id	school_name	start	end
807	251	Harvard University	1973	1975
806	251	Lakeside School, Seattle	NULL	NULL

contact_info table

id	user_id	type	url
155	251	blog	http://thegatesnotes.com
156	251	twitter	http://twitter.com/BillGates

Why so many tables?

- Regions:
  - Allows many users to refer to shared region data without repetition or inconsistency.
- Positions:
  - Allows a user to have an arbitrary number of positions (zero to infinity).
- Industries? Education? Contact_info?

In summary:

A "relation" is a **table**

- A multi-table (relational) DB allows **many-to-one** and **many-to-many** relationships while keeping columns finite and clearly defined.

users table

user_id	first_name	last_name	summary	region_id	industry_id	photo_id
251	Bill	Gates	Co-chair of … blogger.	us:91	131	57817532

… line wrapped

regions table

id	region_name
us:7	Greater Boston Area
us:91	Greater Seattle Area

industries table

id	industry_name
43	Financial Services
48	Construction
131	Philanthropy

positions table

id	user_id	job_title	organization
458	251	Co-chair	Bill & Melinda Gates F…
457	251	Co-founder, Chairman	Microsoft

education table

id	user_id	school_name	start	end
807	251	Harvard University	1973	1975
806	251	Lakeside School, Seattle	NULL	NULL

contact_info table

id	user_id	type	url
155	251	blog	http://thegatesnotes.com
156	251	twitter	http://twitter.com/BillGates

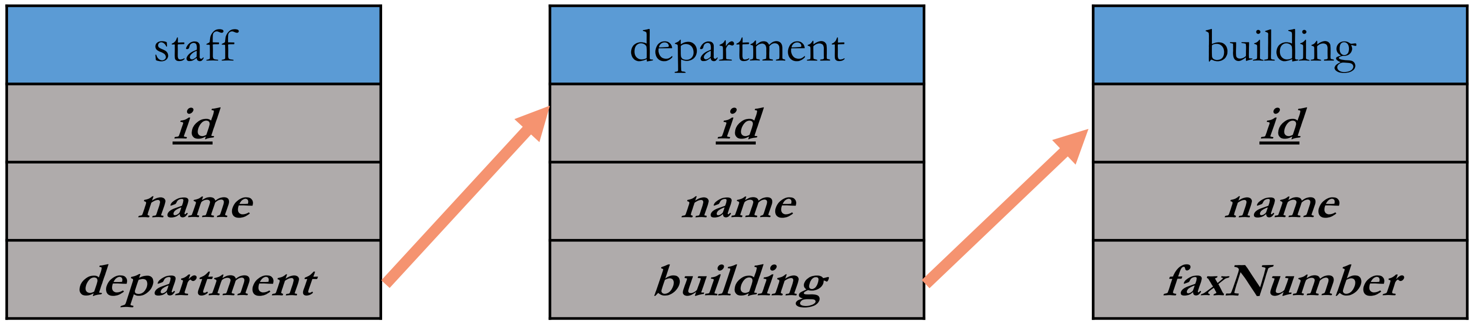
staff			
id	***name***	***room***	***depart-ment***
11	Bob	100	1
20	Betsy	100	2
21	Fran	101	1
22	Frank	102	4
35	Sarah	200	5
40	Sam	10	7
54	Pat	102	2

department		
id	***name***	***building***
1	Industrial Eng.	1
2	Computer Sci.	2
4	Chemistry	1
5	Physics	4
7	Materials Sci.	5

building		
id	***name***	***faxNumber***
1	Tech	1-1000
2	Ford	1-5003
4	Mudd	1-2005
5	Cook	1-3004
6	Garage	1-6001

DB Design diagram: *(my style)*

- A graphical representation of the DB **schema**.
- Defines the tables, columns, primary and foreign keys



For more coverage of relational DB schema design:

- https://youtu.be/kqNpwL14nns?t=267
- Or search Youtube for "Tarzia 317 Lecture 07"
- This is **highly recommended** if you have not taken a database course, and probably helpful even if you *have* taken CS-339.

SQL Query language

```
SELECT staff.id, staff.name, staff.room,
       department.name, department.buildingId
  FROM staff JOIN department
       ON staff.departmentId=department.id
```

staff.*id*	staff.*name*	staff.*room*	department.*name*	department.*buildingId*
11	Bob	100	Industrial Eng.	1
20	Betsy	100	Computer Sci.	2
21	Fran	101	Industrial Eng.	1
22	Frank	102	Chemistry	1
35	Sarah	200	Physics	4
40	Sam	10	Materials Sci.	5
54	Pat	102	Computer Sci.	2

Why a Relational Database?

Most importantly:

- **Scalability** – work with data larger than computer's RAM.
- **Persistence** – keep data around after your program finishes.
- **Indexing** – efficiently sort & search along various dimensions.
- **Concurrency** – multiple users or applications can read/write.
- **Analysis** – SQL query language is concise yet powerful.

And also:

- **Integrity** – restrict data type, disallow duplicate entries, transactions.
- **Deduplication** – save space, keep common data consistent.
- **Security** – different users can have access to specific data.

Can we just read/write **files** to disk to achieve these?

- **Scalability** – work with data larger than computer’s RAM.
- **Persistence** – keep data around after your program finishes.
- **Indexing** – efficiently sort & search along various dimensions.
- **Concurrency** – multiple users or applications can read/write.
- **Analysis** – SQL query language is concise yet powerful.

- **Integrity** – restrict data type, disallow duplicate entries, transactions.
- **Deduplication** – save space, keep common data consistent.
- **Security** – different users can have access to specific data.

Filesystem is like a basic database, it gives:

- **Scalability** – work with data larger than computer's RAM.
- **Persistence** – keep data around after your program finishes.
- **Indexing** – efficient access in just one dimension – the path/filename.
- **Concurrency** – multiple apps can read/write, but lacks **transactions**.
- ~~**Analysis** – SQL query language is concise yet powerful.~~

- ~~**Integrity** – restrict data type, disallow duplicate entries, transactions.~~
- ~~**Deduplication** – save space, keep common data consistent.~~
- **Security** – different users can have access to specific data.

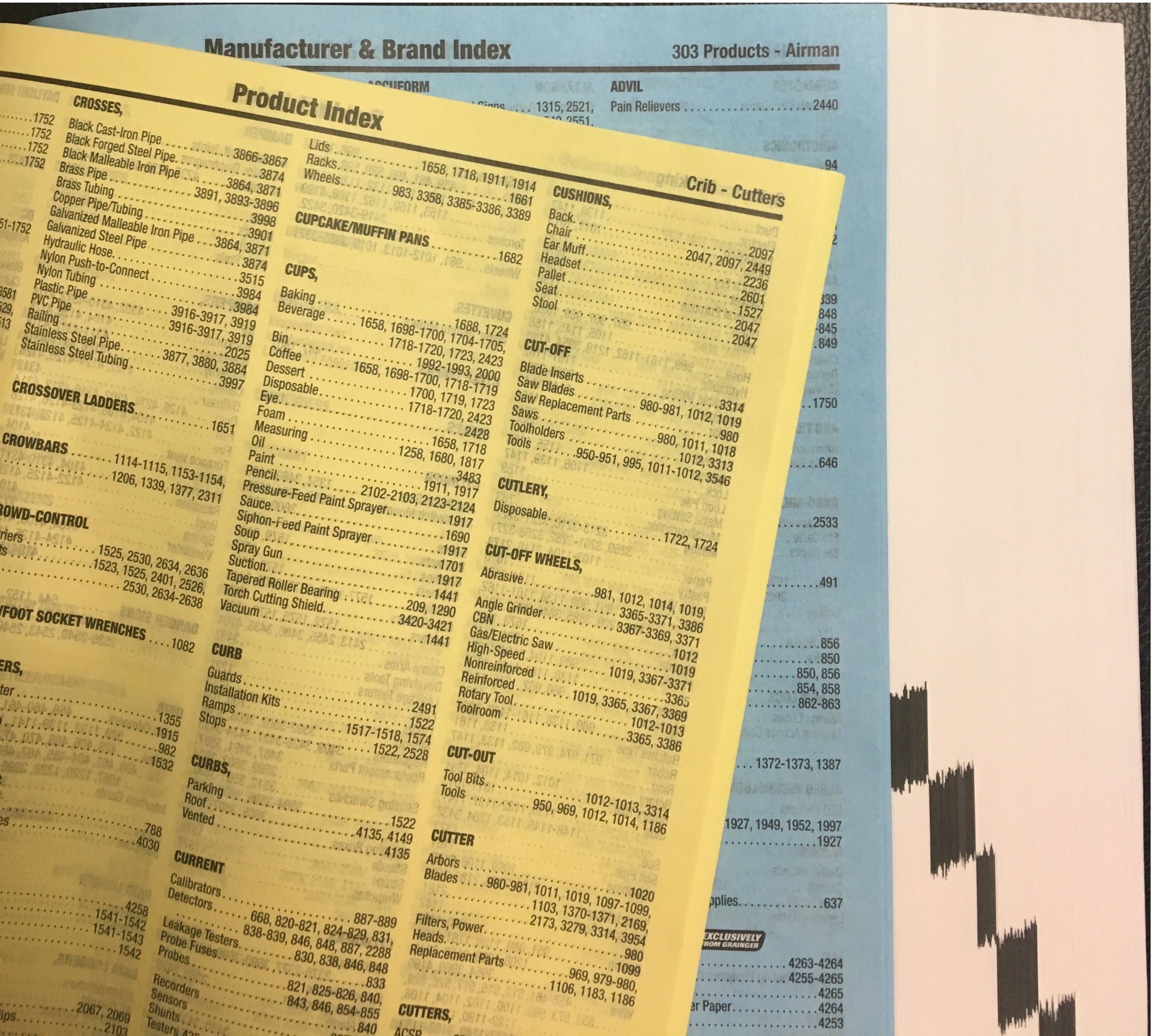
Indexing

- When working with large amounts of data it can be a challenge to find an item of interest.
- We don't want to read every storage address to find what we're looking for.
- ***Sorting*** the data can help tremendously, because it allows *binary search*.

Why sorting is not enough

- You can't sort in **multiple dimensions**
  - Let's say you want to find a product quickly according to either it's name, manufacturer, or price. You can only sort by one of the there three columns.
- Can't **insert new data** without *shifting* everything over to make room.
  - Shifting data in storage would require rewriting about half of it (on average).
  - That's incredibly amount of work to accommodate just one tiny addition.
- Sorting doesn't take advantage of the hardware's storage hierarchy.
  - The binary search will have to access the disk in every step because the index is distributed over the full data set.
  - It would be better to put all the index data close together (spatial locality).

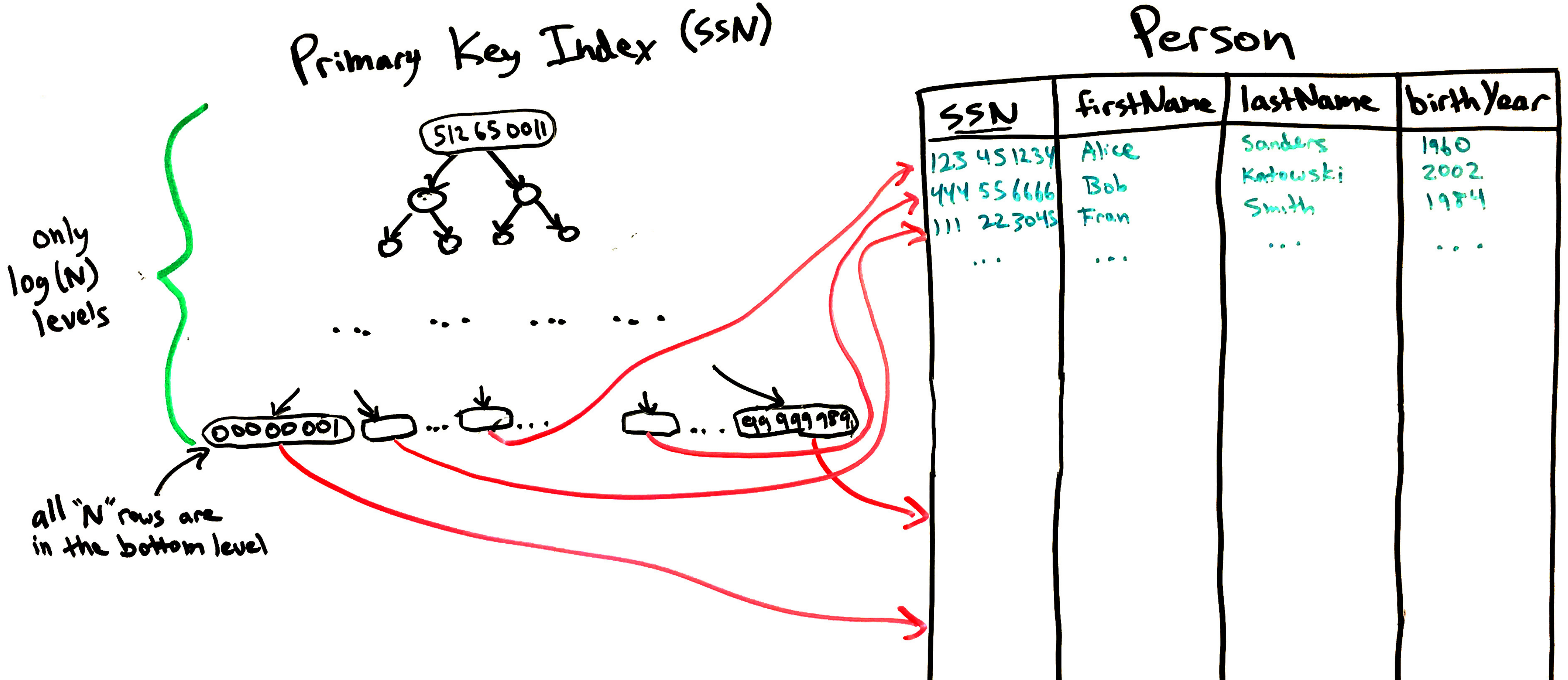
A printed catalog can add multiple indexes



- Grainger catalog is sorted according to high-level *product categories*.
- It has both yellow and blue index pages.
- These allow efficient lookup by:
  - *product type names*
  - *manufacturer names*
- In total, products can be efficiently found in three ways.
- Simple sorted lists are effective here because data is never added.

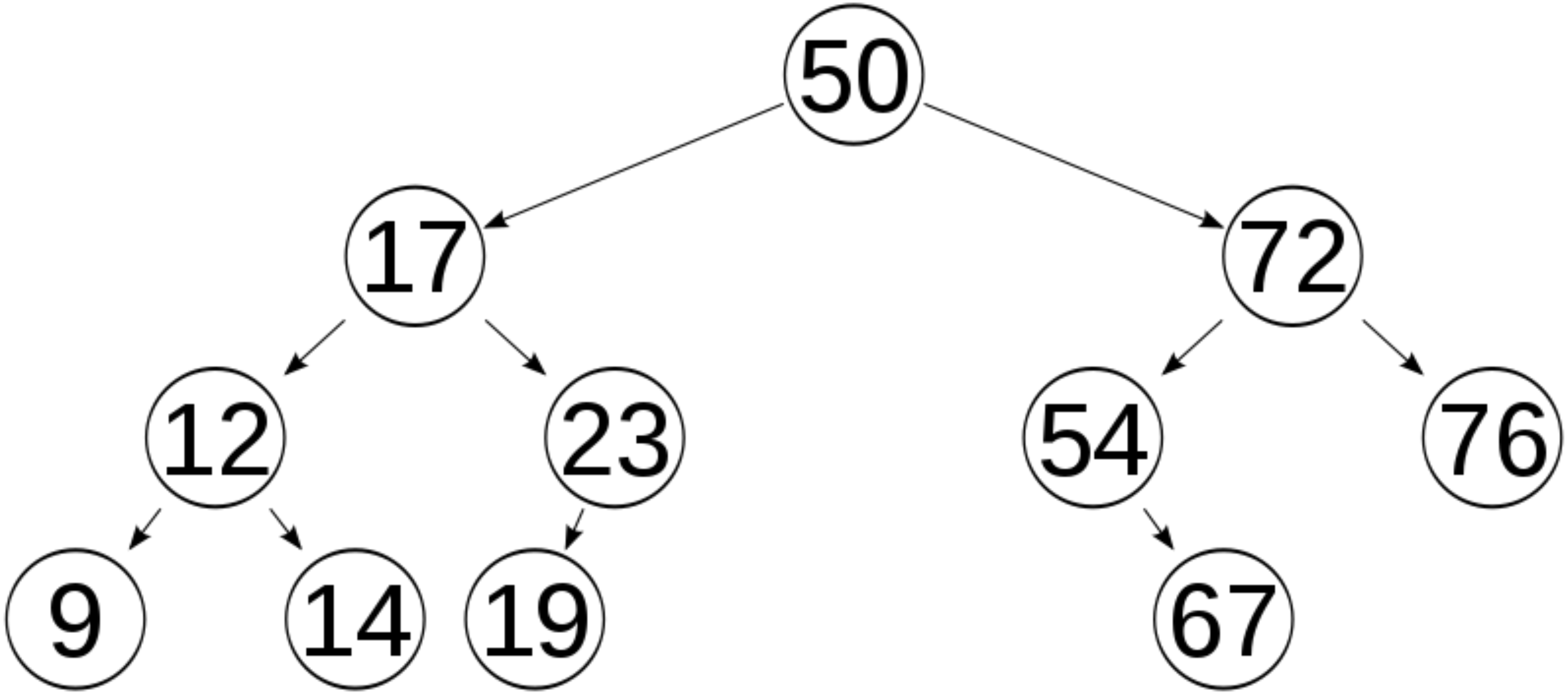
DB indexes use a *tree* or *hashtable* instead of sorting

- Self-balanced binary trees give the log(N) speed of a binary search, while also allowing entries to be quickly added and deleted.
- This is all review of CS-214 Data Structures.



Balanced binary search tree

- Finding an element is very similar to binary search of a sorted list.
- Start from the root. Move to the **left subtree** if the value you're looking for is smaller, otherwise move to the **right subtree**.
- Repeat.



Creating indexes/keys

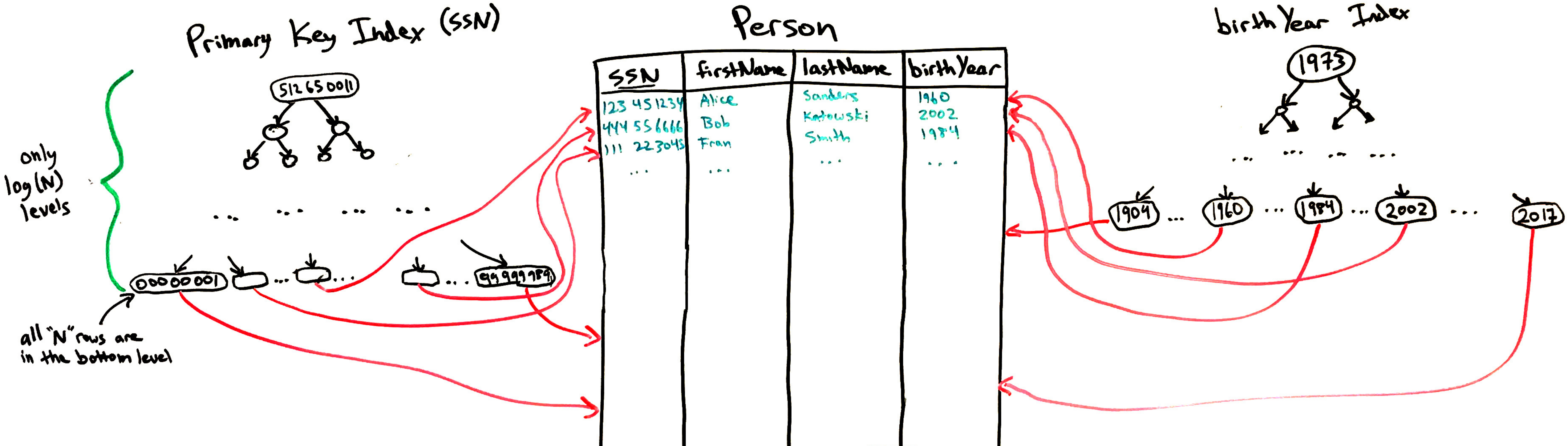
- Indexes are usually defined when the table is created
  - ***Primary key*** must be unique for each row.
  - We must be able to quickly check that new value does not already exist.
  - Thus, unique/primary keys are indexed.
- But you may later realize that certain queries are too slow
  - Without proper indexes, DBMS will have to examine every row in the table to find the relevant rows.
  - Adding one or more indexes may dramatically speed up a query.

Basic syntax:

```
CREATE INDEX index_name ON table_name (column_name)
```

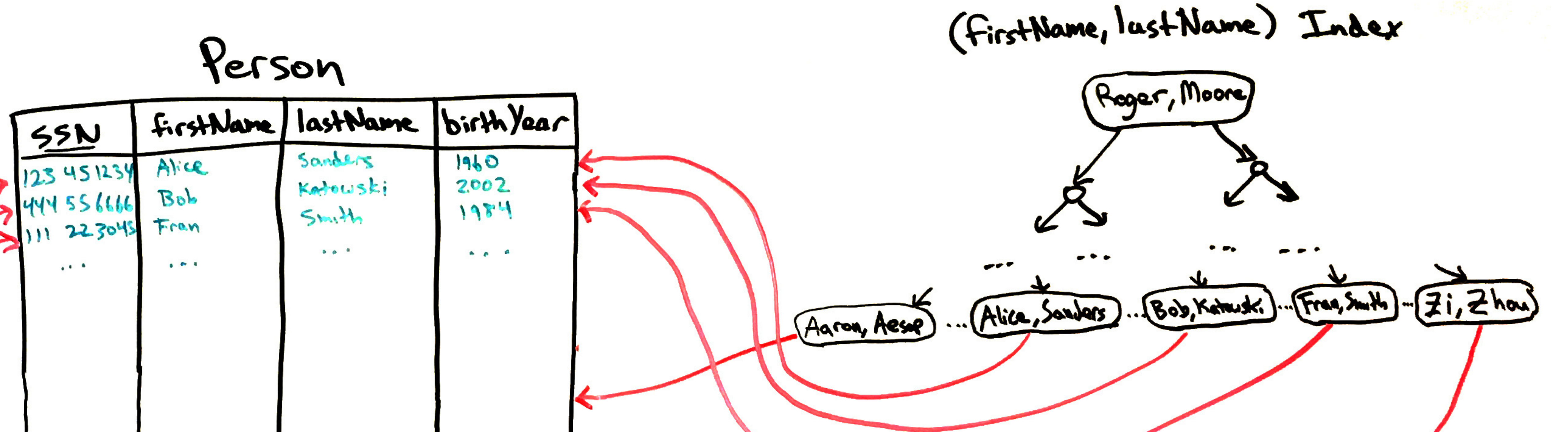
Multiple indexes in one table are possible

- Allow finding rows quickly based on multiple criteria
- Need two indexes to quickly get results for both:
  - `SELECT * FROM Person` **`WHERE SSN=543230921`**
  - `SELECT * FROM Person`
    `WHERE birthYear BETWEEN 1979 AND 1983`



Composite indexes involve multiple columns

- Useful when WHERE clauses involves pairs of column values:
  `SELECT * FROM Person WHERE firstName = "Alice" AND lastName = "Sanders"`
- Unlike two separate indexes, you can find the *matching pair* of values with one lookup.
- Otherwise, would have to first find results for `firstName = "Alice"` and scan through all the Alices checking for `lastName = "Sanders"`
- However, example below does not allow you to quickly find rows by lastName



Query execution plans

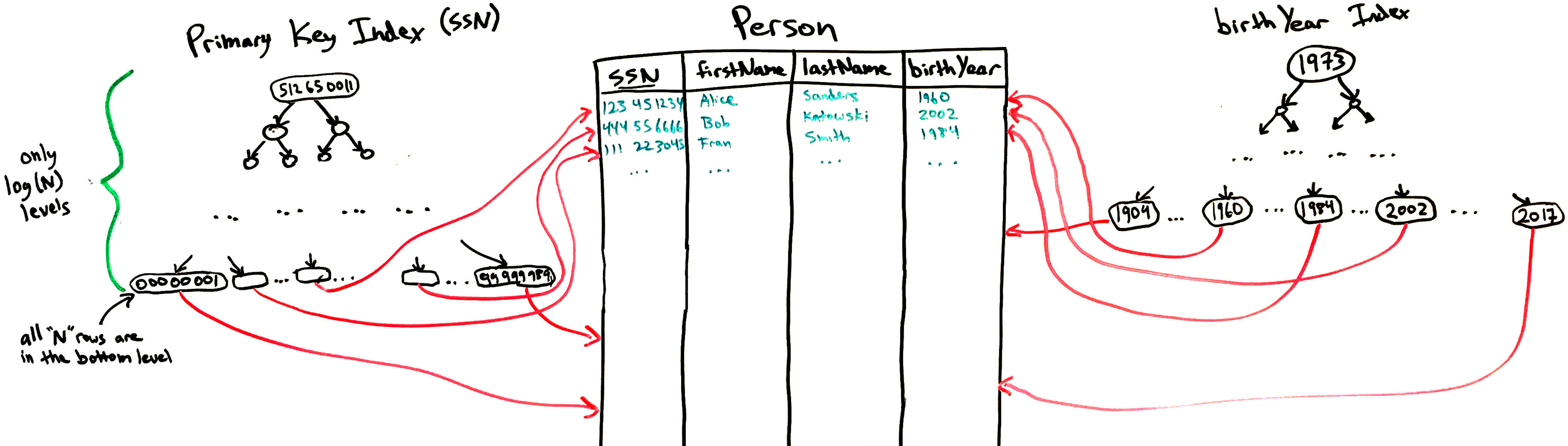
- The DBMS must translate your SELECT query into a series of table lookups.
- A complex query has many choices about what to do first, and it will try to make the most efficient choice.
  - For example, if a JOIN is used, either of the two tables can be examined first.
  - The presence of indexes make some choices more efficient than others.
- DBMSs have special commands that explain the query execution plan:
  - SQLite: `EXPLAIN QUERY PLAN SELECT …`
  - MySQL: `EXPLAIN SELECT …`
  - This usually tells you how many rows will be examined, and adding indexes can reduce these numbers.

When to index columns?

- When a query is slow!
- Generally, add an index if the column is:
  - Used in `WHERE` conditions, or
  - Used in `JOIN … ON` conditions, or
  - A foreign key refers to it.
- Also helpful if the column is:
  - In a `MIN` or `MAX` aggregation function

Indexes are not free!

- Don't add indexes unless you need them.
- Rookie mistake is to index every column "just in case."
- Indexes consume storage space ***(storage overhead),***
- Indexes must be updated when data is modified ***(performance overhead).***



Key and Index terminology in SQL

- Plain **key** or **index** is just a way to find rows quickly
  - Just creates a search tree.
- **Unique key** is an index that prevents duplicates
  - Bottom level of search tree has no repeated values
  - DBMS can use the tree to quickly search for existing rows with that value before allowing a row insertion (or column update) to proceed.
- **Primary key** is just a unique key, but there can only be one per table
  - We think of the primary key as the *most important* unique key in the table
- **Foreign key** makes a column's values match a column in another table
  - The referenced column in the other table should be indexed (usually it's the primary key).

Summary

- **Persistent** storage requires special consideration due to slow performance and lack of language-level support.
- Databases solve lots of problems:
  - **scalability, persistence, indexing, concurrency,** etc.
  - Filesystems can solve some, but not all, of these problems.
- **Relational (SQL) databases** store data in tables.
- Developer defines the DB **schema** first (tables, columns, keys).
  - Rows are added during DB operation, and they must fit the schema.
- **Indexes** let us find rows quickly with value of one or more column.
- SQL query language lets us run analysis code "close to" data storage (filtering, aggregation – sum, count, min, max, avg, etc.).