EECS-317 Data Management and Information Processing

Lecture 10 – Indexes

Steve Tarzia

Spring 2019

Northwestern

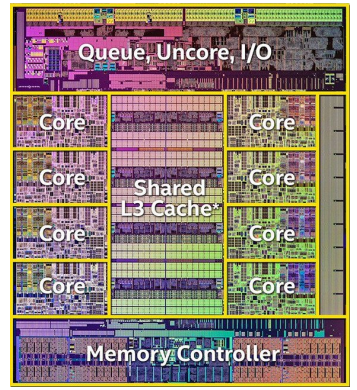
Announcements

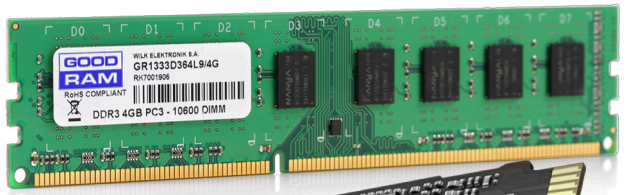
- HW4 was posted yesterday, and it's due on Monday.
- Midterm grades will be posted soon.
- You can pick up your exam from my office hours.
- Friday is the last day to drop.

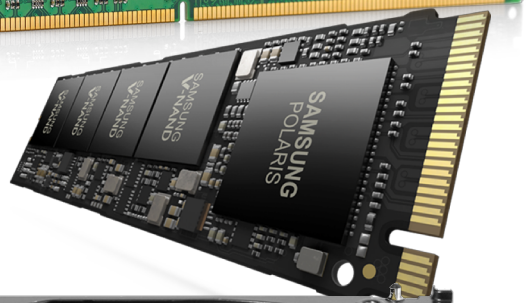
Computers have a hierarchy of storage

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





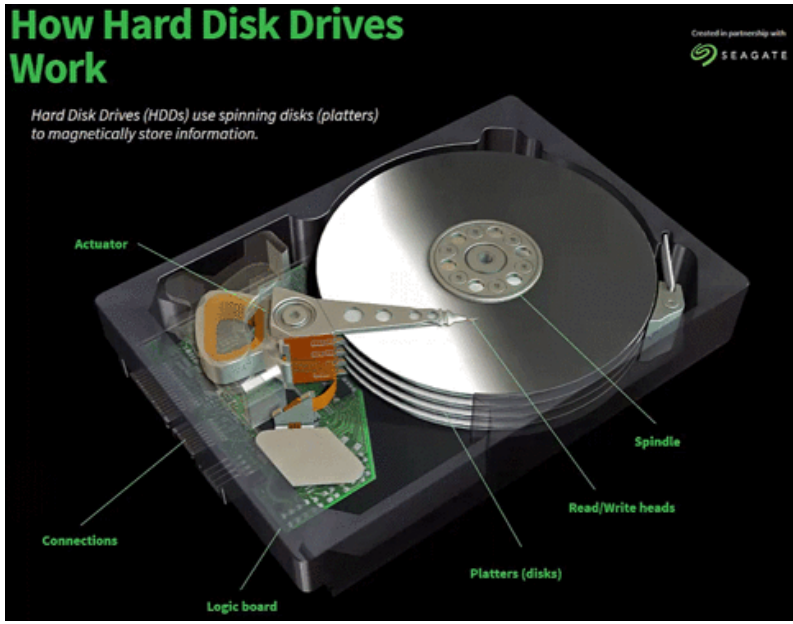


- Disk is about *ten billion* times larger than registers, but has about *ten million* times larger delay (latency).
- Goal is to work as much as possible in the top levels.
- Large, rarely-needed data is stored at the bottom level

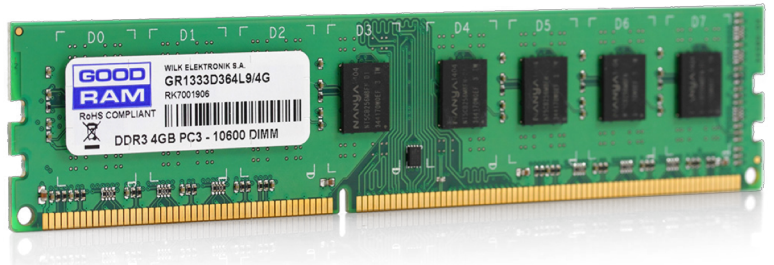
Storage has limited bandwidth

- 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.

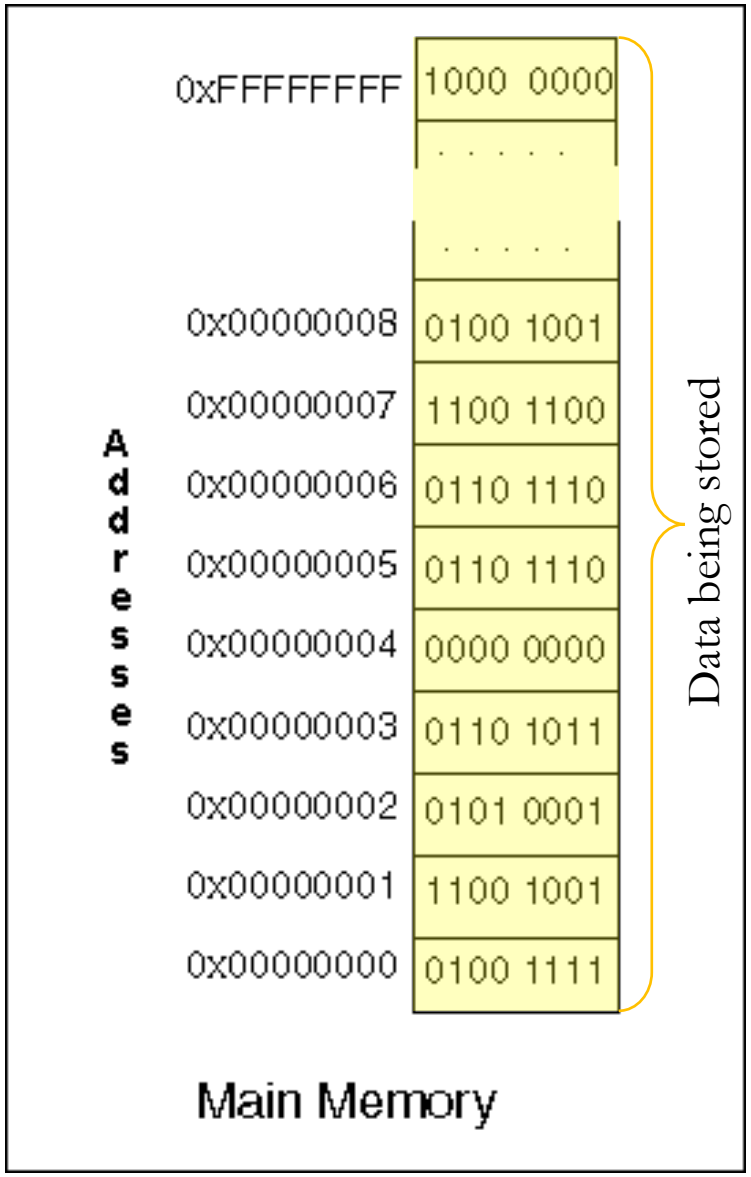


https://animagraffs.com/hard-disk-drive/

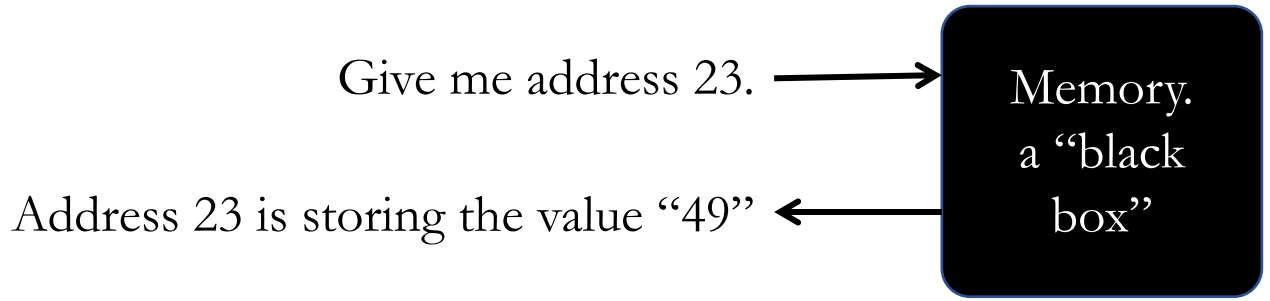


Just a couple hundred electrical connections at the edge of a RAM card.

Abstract view of computer storage

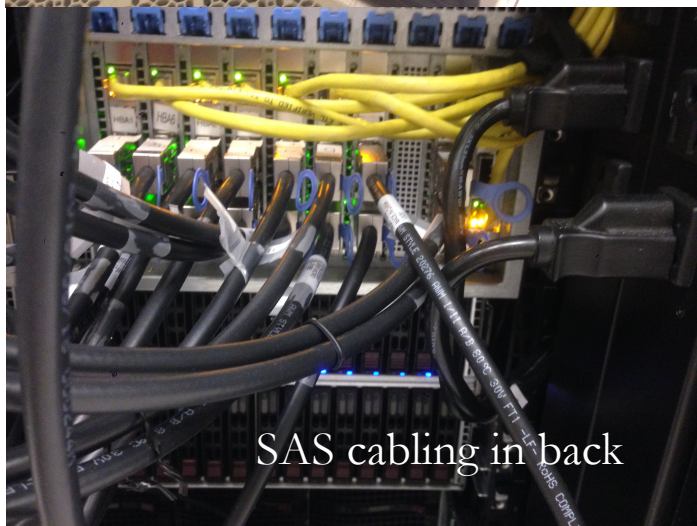


- Storage is a big array of bytes (numbers).
- Computer can read or write from one location (address) at a time.
- The picture at left is misleading because a human observer can *see all the data* at once. The computer cannot!
- Computer requests one address at a time:



A Database Server @ NU

- 264 fast (10k RPM) magnetic disks (for production)
- 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.



Stack Overflow database

- Questions and Answers from a popular programming help website
  - 150 GB of data
  - 29M posts
  - 55M comments
- Reading through all the data takes about 1,000 seconds (17 minutes).

- We don't want to wait 17 minutes for an answer.
- It's impractical to scan through all the data to find what we need.
- We need a way to quickly find the data of interest *(indexing!)*

Indexing

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

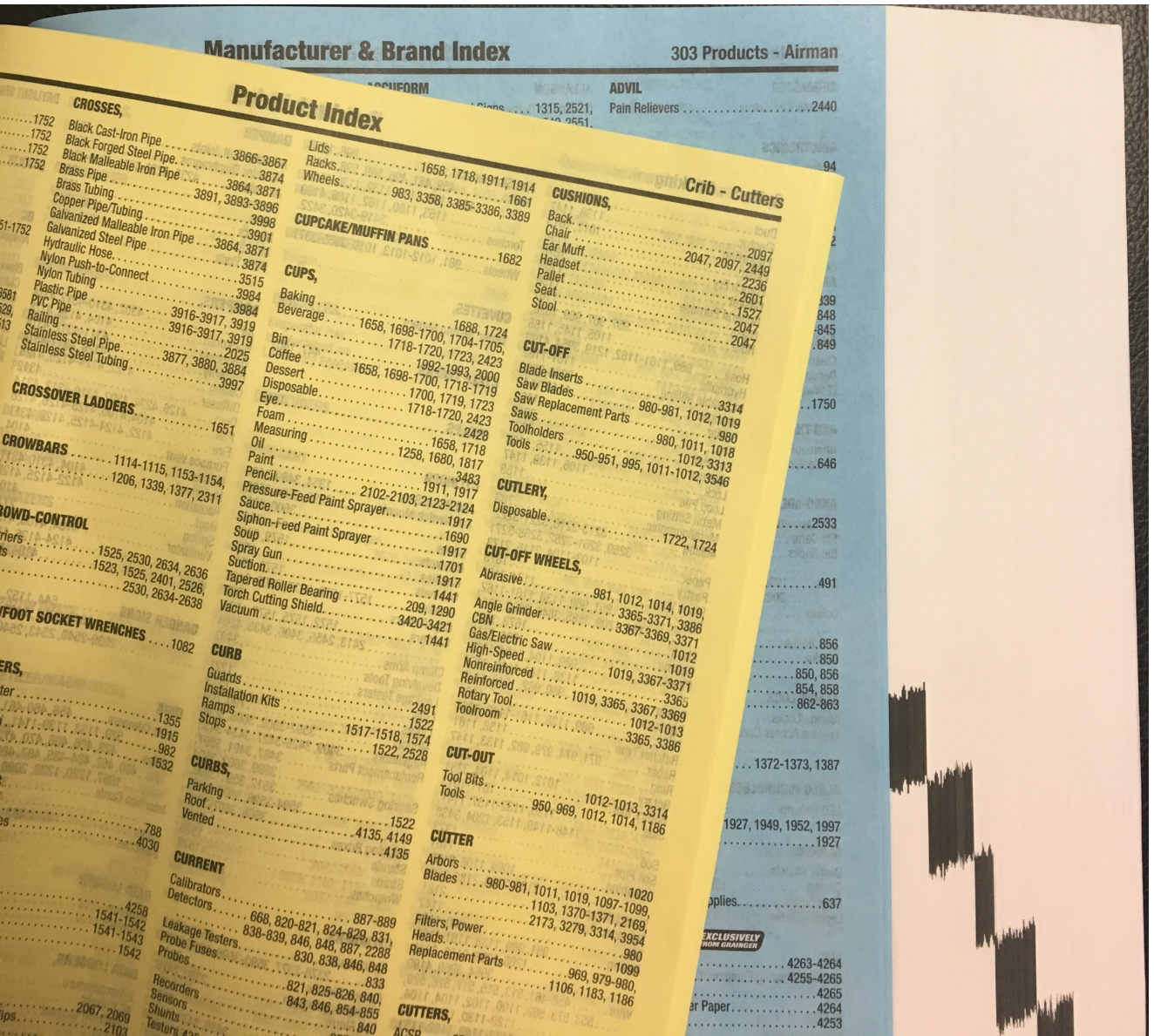
Sorting and Binary Search

- We know it's easy to find data if it's in a ***sorted*** list.
  - That's why printed dictionaries and phone books are alphabetical.
- **Binary Search** is how computers find entries in a sorted list.
  - Let's say you're looking for the word "key" in a list of 10,000 words
    1. Compare "key" to the word in the middle position (5,000th word).
    2. If you're lucky and that middle word is ***equal to*** "key", then you're done!
    3. If the middle word is ***greater than*** "key" then go back to step 1, but refine your search to just the left half of the list (words 0 through 4,999).
    4. If the middle word is ***less than*** "key" then go back to step 1, but refine your search to just the right half of the list (words 5,001 through 10,000).
  - At most is will take $\log_2 N$ steps to find the entry, where N is the list size.
    - Eg., 32 steps for binary search in a list of 4 billion entries (because $2^{32} \cong$ 4 billion)

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).

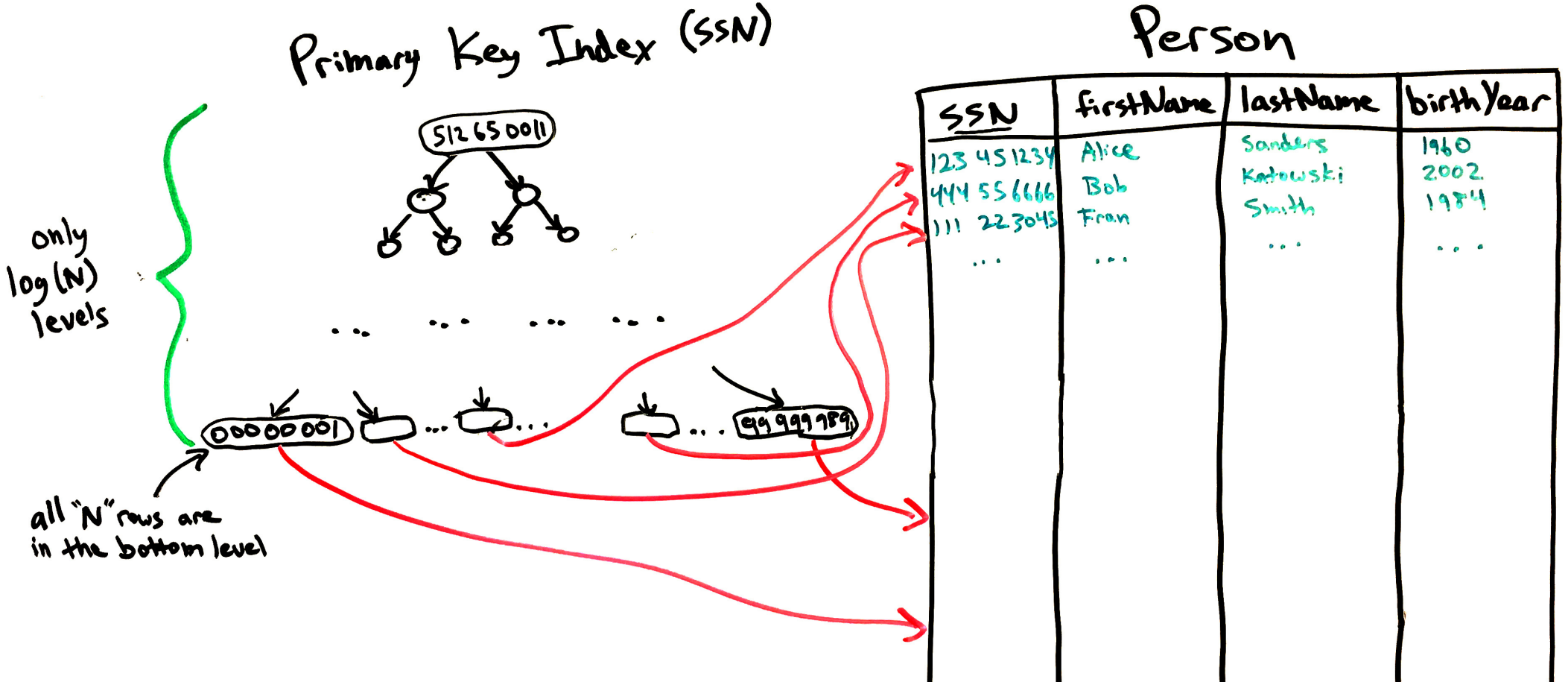
A printed catalog can add multiple indexes



- 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.

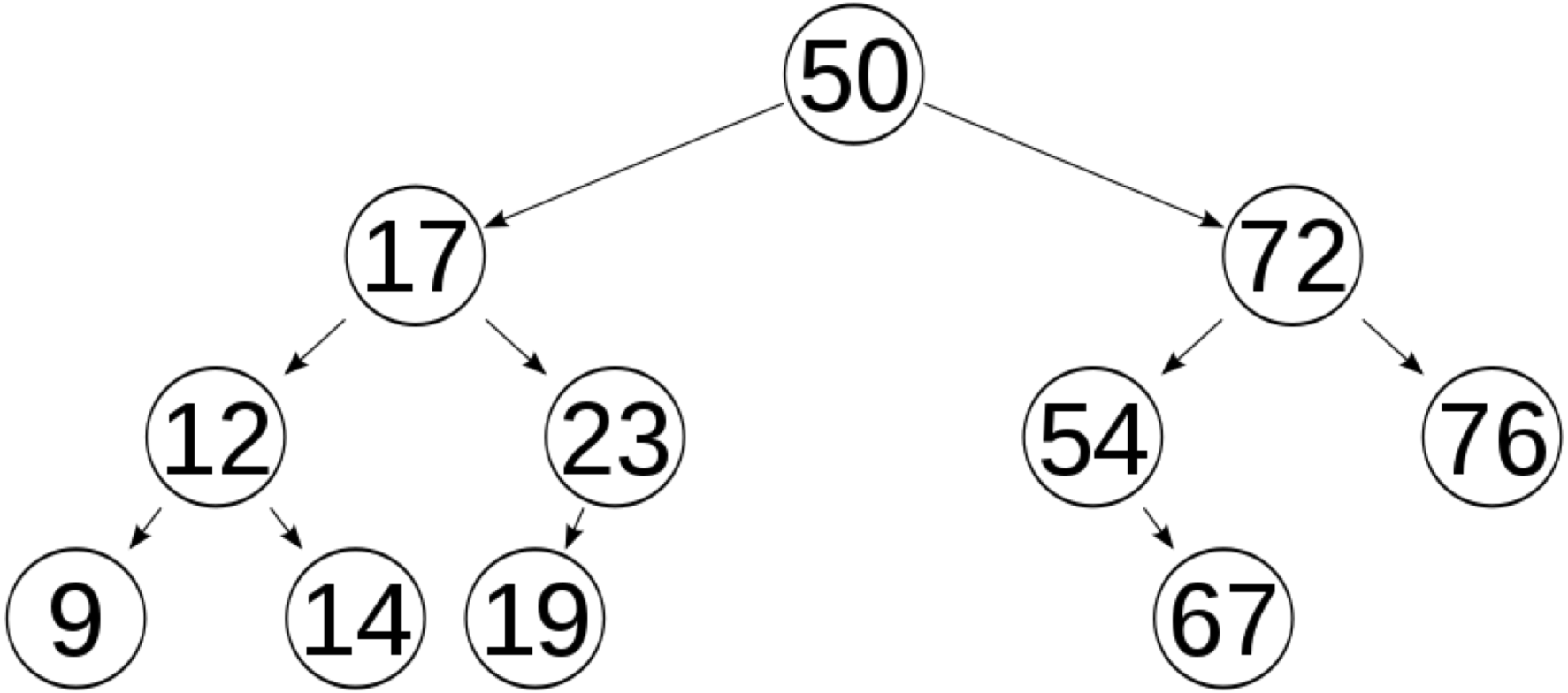
DB indexes use a *tree* or *hashtable* instead of sorting

- Self-balanced binary trees give the log(N) speed of a binary search, while also allowing entries to be quickly added and deleted.
- The details are beyond scope of this class (covered in 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

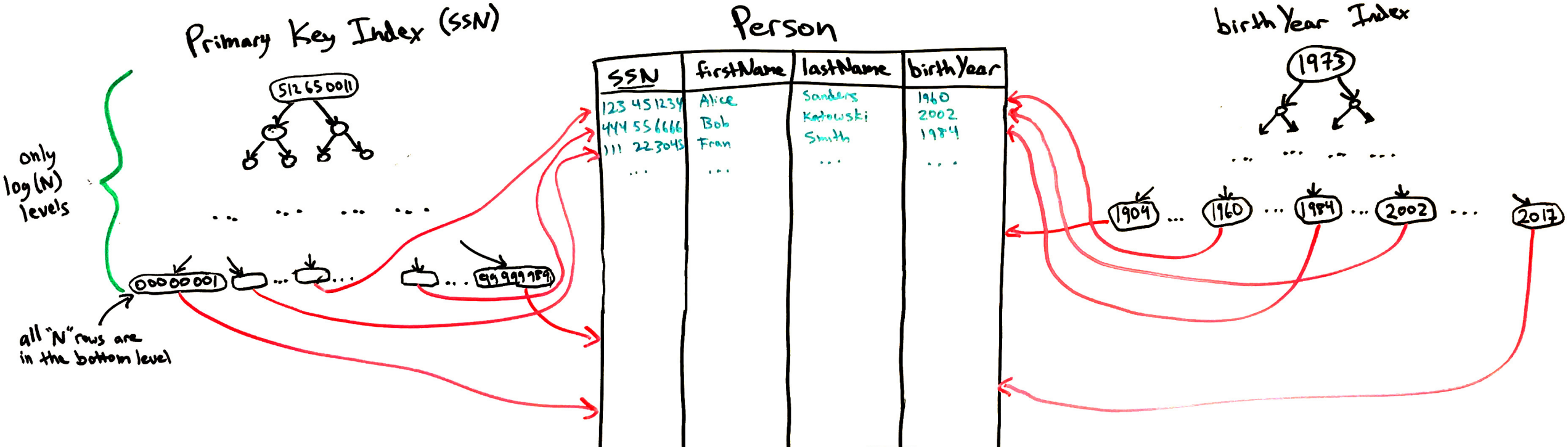
- Indexes are usually defined when the table is created
  - ***Primary key*** must be unique for each row.
  - We must be able to quickly check that new value does not already exist.
  - Thus, unique/primary keys are indexed.
- But you may later realize that certain queries are too slow
  - Without proper 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)
```

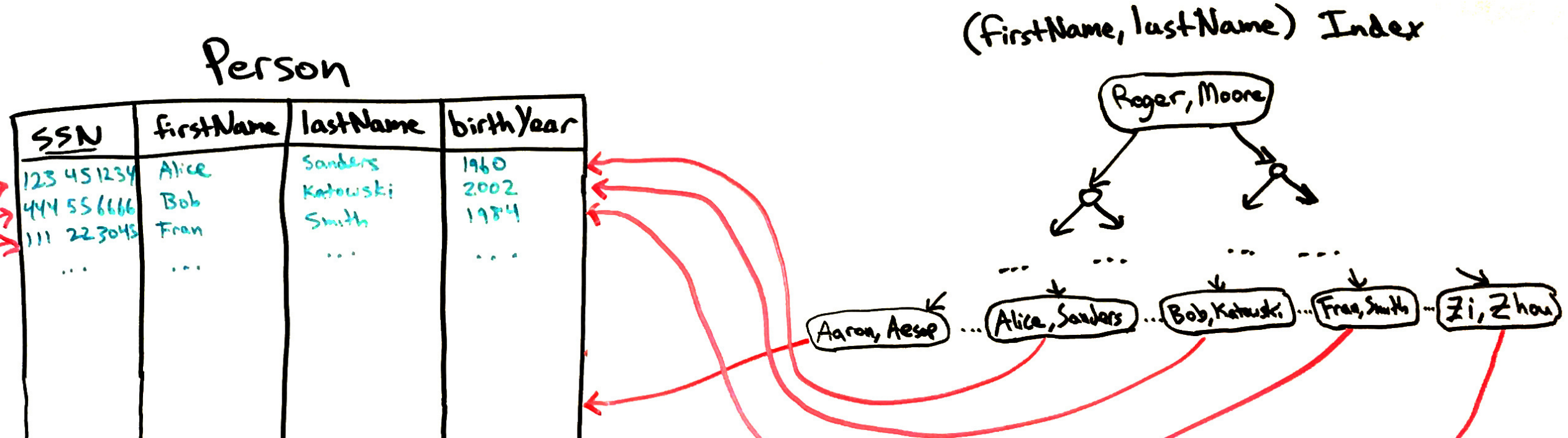
Multiple indexes in one table are possible

- 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

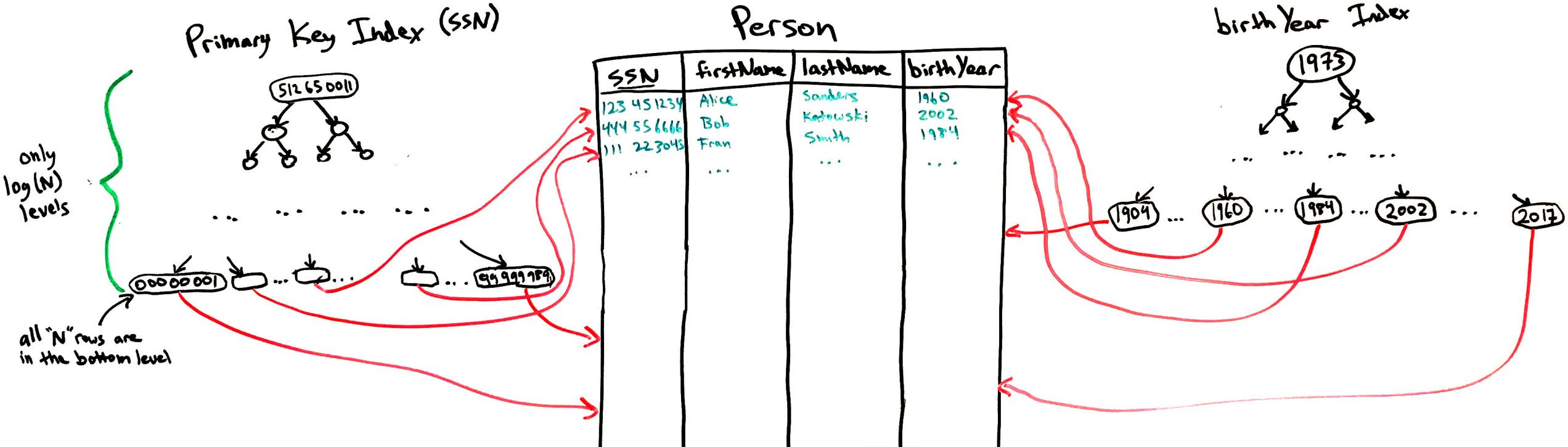
- The DBMS must translate your SELECT query into a series of table lookups.
- A complex query has many choices about what to do first, and it will try to make the most efficient choice.
  - For example, if a JOIN is used, either of the two tables can be examined first.
  - The presence of indexes make 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. *(examples follow)*

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).

Recap

- Large relational databases can store many terabytes of data.
- However, it can take hours days to scan through all the data.
- **Column Indexes** allow row with particular values to be quickly found.
  - If we have N rows, an index allows data to be found in log(N) time.
- Tables can have **multiple (secondary) indexes**.
  - These should be added for columns whole values are often tested in queries.
- **Composite indexes** operate on multiple column values.
- Indexes (a.k.a. keys) are classified as:
  - *Primary Keys*
  - *Unique Keys* (similar to above)
  - *Foreign Keys* (referring to keys in another table)