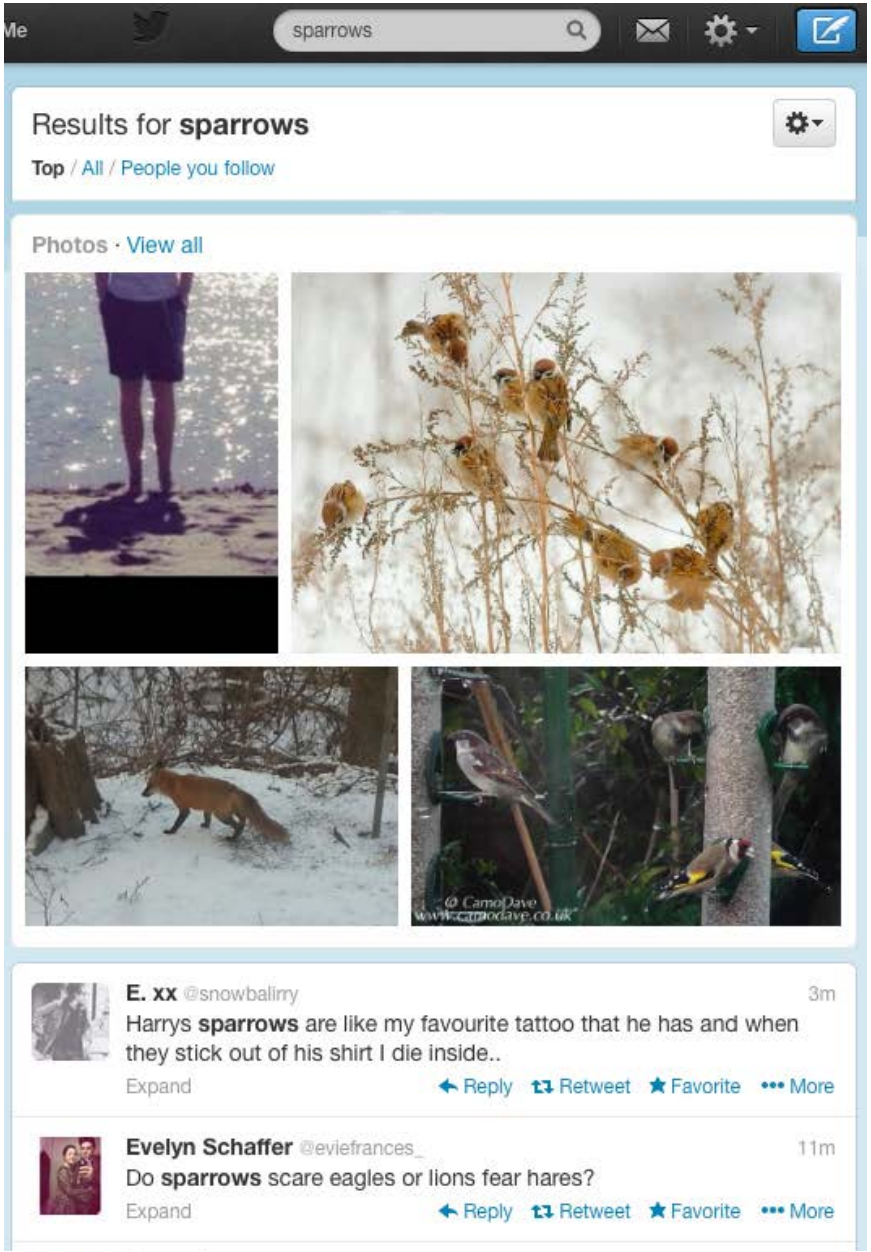
CS6200
Information Retrieval

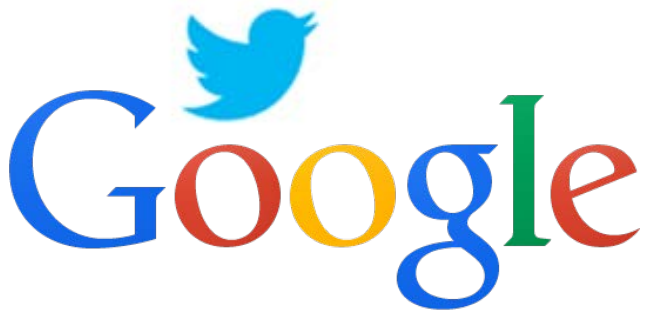
Jesse Anderton
College of Computer and Information Science
Northeastern University

Commercial Search



Commercial Search

- We have focused so far on a high level overview of Information Retrieval, but how does it apply to specific companies?
- Ranking has many applications:
  - ➡ Document search and filtering
  - ➡ Product recommendation
  - ➡ Suggesting social connections
- Businesses also employ many other IR tasks.



Case Studies

- Let's go through a few case studies to see how we can pull together various ideas into a more complete product.
- The ideas presented here are often somewhat incomplete, and don't necessarily represent how any particular company's system actually works.
- The idea is to show a portion of the product development process.

Ranking a Feed

Ranking a Feed | Making a Suggestion
Data Storage at Scale | Task Distribution

Let’s build a social network!

- Our users create posts with text, pictures, and links, and can subscribe to other users’ feeds.
- We show users a feed of content from other users.
- We make our money when users follow links.
- In order to drive clicks to those links, we want lots of users linked to lots of friends discussing lots of posts. User engagement drives up revenue.
- Our business problem: how should we rank the posts in a given user’s feed to maximize our revenue and their engagement?

Let's build a profitable social network!

- We want our ranking to have the following properties:
  - ➡ Prefer posts from friends
  - ➡ Prefer posts with links – but don't crowd out other posts
- Let's quantify these goals, and then combine them into a ranking function.

Rohit V.: I wasted my whole evening watching this crazy movie!

Amanda S.: I just had the worst day…

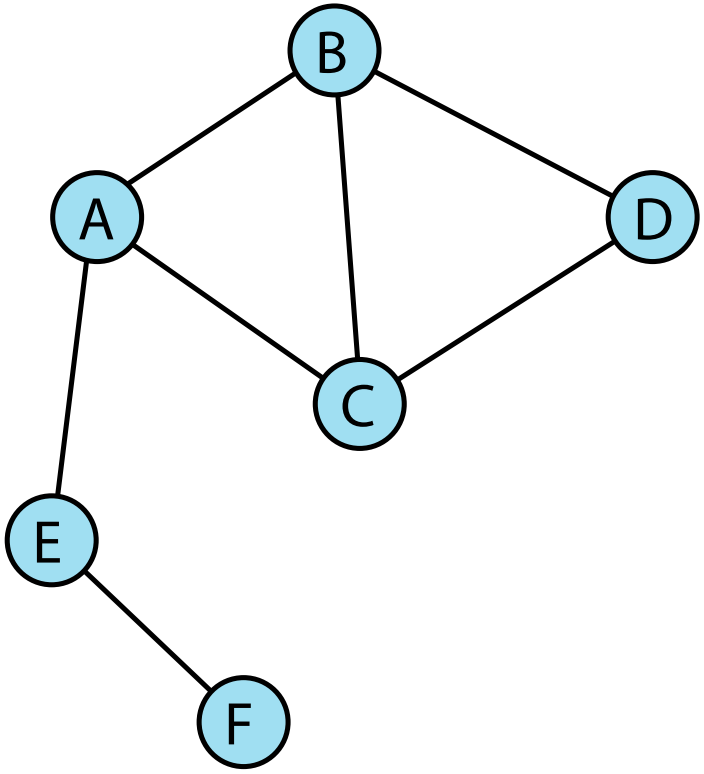
Justin B.: Anyone want to see my new video?

Ranking By Probability

- We have previously ranked by generating a *matching score* between a document and a query, and then sorting by that score.
- The score can be a probability, like the probability this document contains relevant content.
- In this case, we care about the probability a post will maximize some function of our revenue and our users' engagement.
- Our approach will be to choose several different probability functions we believe to be correlated with revenue and/or engagement, and then combine them into a score for ranking.

Posts from Friends

- This looks easy: if user A is following user B, then show B's content in A's feed.
- If A is following 100 users, who wins? Some ideas:
  - ➡ Prefer users who A interacts with more (in terms of comments, clicks, likes…)
  - ➡ Prefer users to whom A is more strongly connected
  - ➡ Decide somewhat randomly, so A has a chance of seeing everyone



Rate of Interaction

- If A interacts more with B, we should have a higher probability of showing B's content.
- We want more recent interactions to count more, so we notice when A's preferences change.
- Let's use the number of interactions on each particular day for the last 90 days:

$$Pr(user = b) \propto \frac{\sum_{t=1}^{90} interactions(b, t)}{\sum_{u \in users} \sum_{t=1}^{90} interactions(u, t)}$$

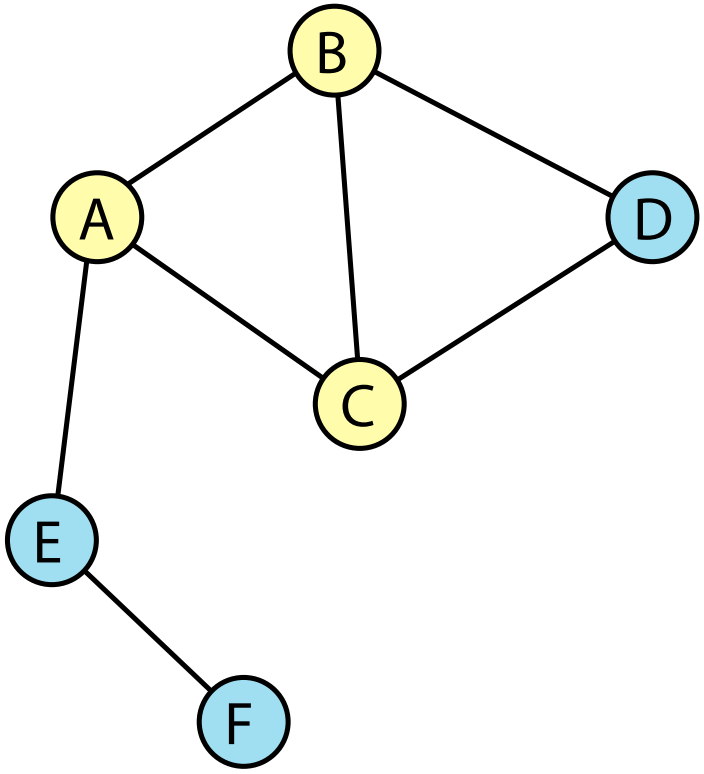
Connection Strength

- Three nodes A, B, and C are members of a *triangle* if they form a 3-clique (see diagram).
- A and B are more tightly connected if they are jointly members of more triangles.

$$Pr(user = b) \propto strength(a, b)$$

$$strength(a, b) = |\{v \in V : (a, v) \in E \text{ and } (b, v) \in E\}|$$

- This is a simplified form of the *clustering coefficient*, which measures a node's influence.



Clustering Coefficient

- A node is considered more influential if more of its outgoing links form triangles.

$$Pr(user = b) \propto cc(b)$$

$$cc(v) = \frac{|\{(u,w) \in E : u \in \Gamma(v) \text{ and } w \in \Gamma(v)\}|}{\binom{d_v}{2}}$$

$$\Gamma(v) \text{ is the set of nodes reachable from } v$$

$$d_v \text{ is the out-degree of } v$$

- Counting triangles in a large social network is difficult, and many papers have been written to refine the algorithms.

- See, e.g. Counting Triangles and the Curse of the Last Reducer, by Suri and Vassilvitskii, 2011.

Popular Links

- We make our money from clicks on user-posted links, so we want to show links to everyone.
- How can we choose links which a user is likely to click? Some ideas:
  - ➡ A user may click links which are more popular among that user's friends, or among all users.
  - ➡ A user may click links which are similar to other links the user has posted or clicked on.
  - ➡ These can be combined: a user may click links which are similar to links which are popular among similar or related users. See Collaborative Filtering, later on.

Link Similarity

- Let's focus on links which are similar to others the user has posted.
- We have already studied ways of measuring the similarity between pages in detail:
  - ➡ Use a vector space representation and measure cosine similarity
  - ➡ Train a topic model on the collection of documents, and treat documents as more similar when their distribution over topics is more similar

Link Similarity

- Topic models were covered in the Ranking 2 lecture.
- Each document is treated as a mixture of topics:

$$|\vec{d_i}| = \#\text{ topics}$$
$$d_{i,j} = Pr(topic = j | doc = d_i)$$

- We can measure the difference between two documents as KL-divergence between their topic distributions:

$$dist(\vec{d_1}, \vec{d_2}) = D(\vec{d_1} || \vec{d_2})$$
$$= \sum_i d_{1,i} \log \frac{d_{1,i}}{d_{2,i}}$$

Arts	*Budgets*	*Children*	*Education*
new	million	children	school
film	tax	women	students
show	program	people	schools
music	budget	child	education
movie	billion	years	teachers
play	federal	families	high
musical	year	work	public
best	spending	parents	teacher
actor	new	says	bennett
first	state	family	manigat
york	plan	welfare	namphy
opera	money	men	state
theater	programs	percent	president
actress	government	care	elementary
love	congress	life	haiti

Example LDA Topics

Link Crowding

- We want to show links, so we can generate revenue, but we don't want to *only* show links because that's bad for engagement.

- One simple way to accomplish this is to choose weights for each post type so that, all else being equal, we will have an "interesting" mix of post types.

$$Pr(d_i) \propto t_{type(d_i)}$$

$$\vec{t} = [t_{links}, t_{text}, t_{images}]$$

$$\sum_i t_i = 1$$

Combining Signals

- Our ultimate goal is to combine all of these things into a ranking score we can use to sort posts.
- We have three fundamental types of evidence:
  - ➡ The user *u* who posted the content
  - ➡ The type *t* of content posted
  - ➡ The user's engagement *e* with the content itself (e.g. similarity to previously-engaging content)

Combining Signals

- Let's combine the evidence in a Bayesian fashion:

$$Pr(d_i|u_i, e_i, t_i) = \frac{Pr(u_i, e_i, t_i|d_i)Pr(d_i)}{Pr(u_i, e_i, t_i)}$$
$$\propto Pr(u_i, e_i, t_i|d_i)Pr(d_i)$$

- If we assume a uniform prior and make the Naive Bayes assumption that the variables are independent, we get:

$$Pr(d_i|u_i, e_i, t_i) \propto Pr(u_i|d_i) \cdot Pr(e_i|d_i) \cdot Pr(t_i|d_i)$$

Combining Signals

- $Pr(u_i|d_i)$ is the probability we'd want to highly rank a post from this user, given that they wrote this document.

- We combine the user's overall influence, connectedness to the feed's owner, and rate of interaction with the feed's owner using a similar Bayesian formula.

Combining Signals

- $Pr(t_i|d_i)$ is the probability we'd want to highly rank a post of this type, given that this document has this type.

- Here we will simply use the overall mixture probability we use to show an appropriate number of links.

Combining Signals

- $Pr(e_i|d_i)$ is the probability the user will be engaged, given that they read this document.

- We combine the document's similarity to documents the user previously found engaging (possibly measured in multiple ways), the document's popularity, the number of clicks (if the document is a link), etc.

What's Missing?

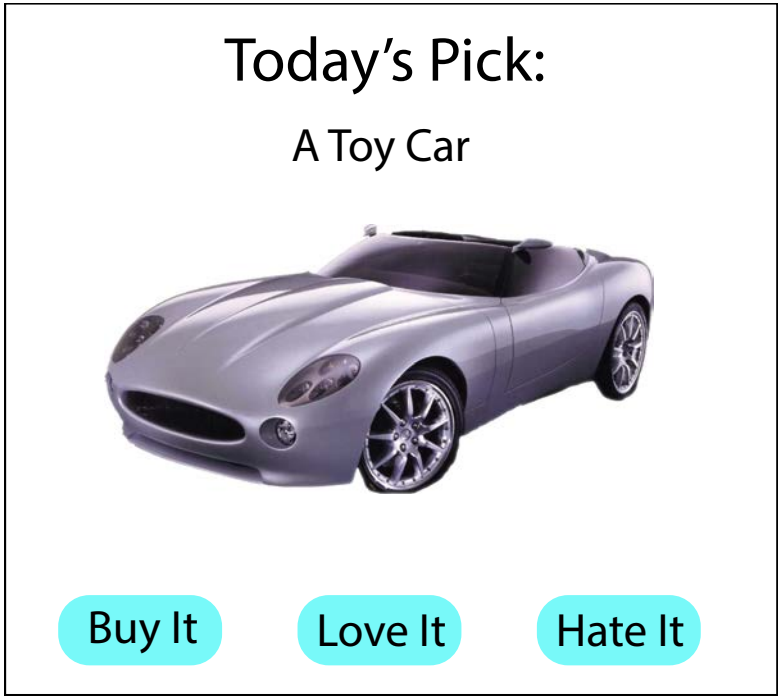
- In practice, we probably don't want a Naive Bayes assumption. Many of these signals are highly correlated.
- We would also like to have parameters we can tune over time, such as the mixture of links to show or how much influential users are preferred over users the feed owner interacts with.
- Two of the many alternatives are to train an Inference Network, discussed in the Retrieval 2 lecture, or to employ Learning to Rank, covered there and in more detail next week.

Making a Suggestion

Ranking a Feed | **Making a Suggestion**
Data Storage at Scale | Task Distribution

Let's Sell Things

- Let's imagine we work for a large online retailer, and are asked to create a new site.
- The site will present one product recommendation per day, based on the user's history with the retailer.
- We want to find the one best product per day, and show something new every day.
- To keep things interesting, let's say that users can interact in four ways: by purchasing the item, giving a thumbs up, giving a thumbs down, or ignoring the recommendation.



Collaborative Filtering

- How can we tell what a person might like before they've seen the product?
- The answer from collaborative filtering is: other users who have expressed preferences similar to our user's preferences can give us evidence about the new product.

			
Susan	👍	👍	👎
Hamid	👍	?	👎
Cheng	👍	👎	👍
Paula	👎	👎	👍

Collaborative Filtering

- We will represent what we already know about user preferences as a matrix:

$$U \in \mathbb{Z}^{m \times n} \text{ for } m \text{ users and } n \text{ items}$$

$$U_{i,j} = \begin{cases} 1 & \text{if user } i \text{ likes item } j \\ -1 & \text{if user } i \text{ dislikes item } j \\ 0 & \text{otherwise} \end{cases}$$

- Instead of 1, you could put a rating value. For instance, use 2 if they bought the item and 1 if they just gave it a “thumbs up.”

Collaborative Filtering

- We will predict a user's rating for an item as the average rating given by the *k* most similar users.

- We will use cosine similarity to compare users:

$$sim(x, y) = \frac{\sum_{i=1}^{n} U_{x,i} \cdot U_{y,i}}{\sqrt{\sum_{i=1}^{n} U_{x,i}^2} \cdot \sqrt{\sum_{i=1}^{n} U_{y,i}^2}}$$

- Our predicted rating, then, is:

$$prediction(x, i) = \frac{\sum_{j=1}^{k} U_{S_j,i}}{k}$$

where S_j is a sorted list of users most similar to *x.*

User Similarity

- This is just the most basic way to computer similarity between users.
- You sometimes know much more about users, and can build a more sophisticated similarity model.
  - ➡ You may have access to user-specific features: age, location, price range of purchased items, interest in items from particular cultures (e.g. books in languages other than English), and so on.
  - ➡ You could build a more sophisticated probabilistic model predicting whether users *x* and *y* will agree on this particular item.
  - ➡ More simply, you could replace cosine similarity with the Pearson correlation coefficient or some other distance function.

Item Similarity

- You can also build in information about item similarity.
  - ➡ Instead of just using user ratings for the particular item you're considering, also include weights for similar items.
  - ➡ Features here might include item category and subcategory, price, overall popularity, popularity by demographic, date released, etc.
  - ➡ This helps with data sparsity – perhaps the product is new or unpopular, and few people have bought it.
  - ➡ You have to be careful of noisy item similarity predictions

What's Missing?

- Collaborative Filtering is an intensely-studied topic with many variations and applications.

- In addition to the rating predicted by collaborative filtering, you will probably want to build a probabilistic model based on the user you're choosing a product for and the item's similarity to other items the user has purchased or rejected.

Data Storage at Scale

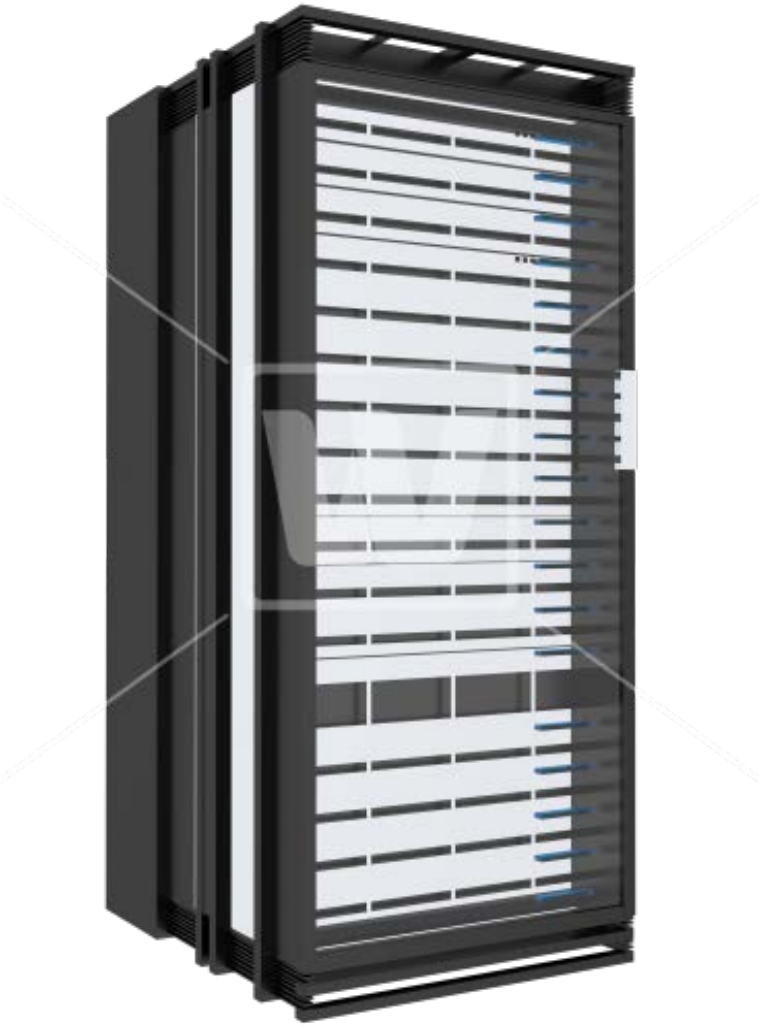
Ranking a Feed | Making a Suggestion
Data Storage at Scale | Task Distribution

Internet Scale

- How many servers does an online company need?
- It varies wildly, changes constantly, and companies generally don't report how many servers they have. A few estimates:
  - ➡ The biggest companies (Google, Microsoft) have over 1,000,000 servers across all their data centers.
  - ➡ Large companies such as Facebook and Amazon generally have hundreds of thousands of servers.
  - ➡ "Smaller" companies such as eBay are estimated to have around 50,000 servers.
- These servers are spread around the world with thousands, or tens of thousands, in a single data center.

Data Storage Paradigms

- Traditionally, businesses have used a SQL database such as MySQL or Oracle.
- Data is stored in tables with fixed schemas. A given row has a clearly-defined set of fields with fixed data types.
- Data access is made efficient by creating indexes on particular fields in a table.
- An index is generally stored in a single file (or is part of a larger file) and is optimized for rapid lookups of particular values, and rapid merging with other indexes and tables.



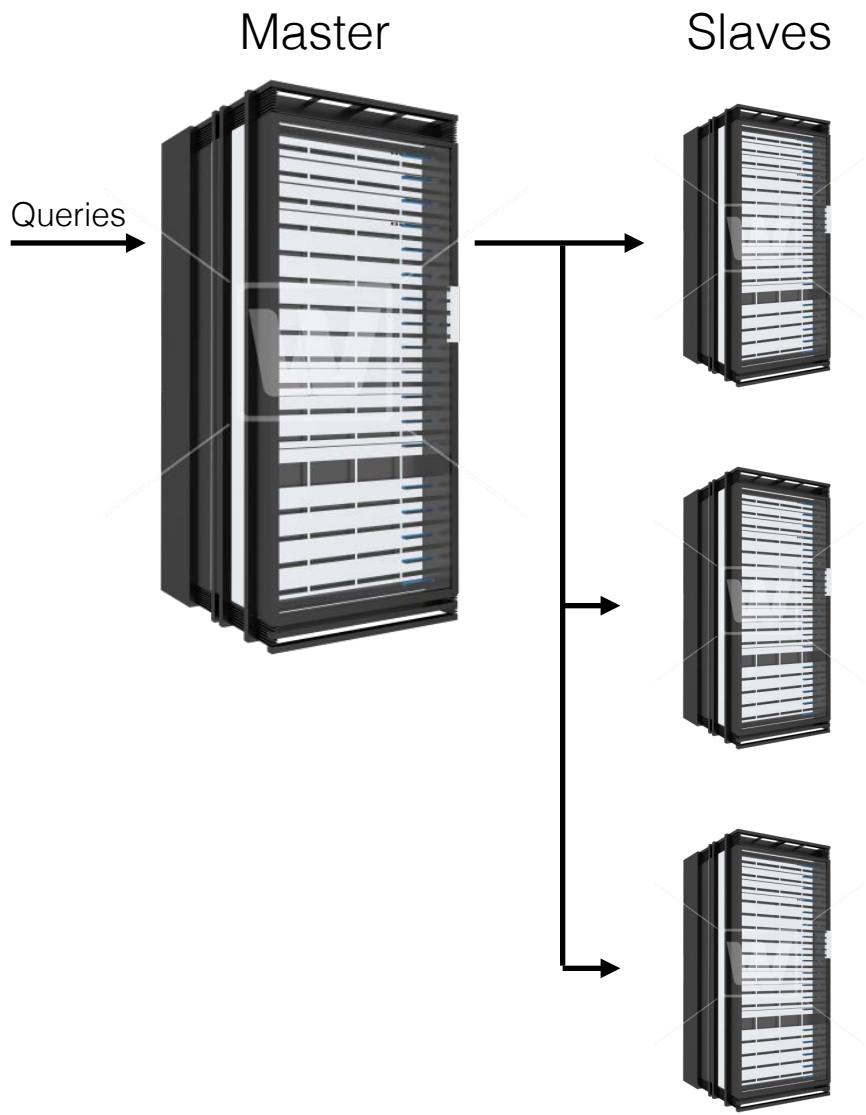
Data Storage Paradigms

- All queries go to a central master server, which can distribute large queries across slave systems.

- As your needs grow, you have two major options:

1. Scale up: buy bigger, more expensive computers with higher capacity.

2. Scale out: buy more slave systems, and carefully design your schema, indexes, and queries to distribute the work efficiently.

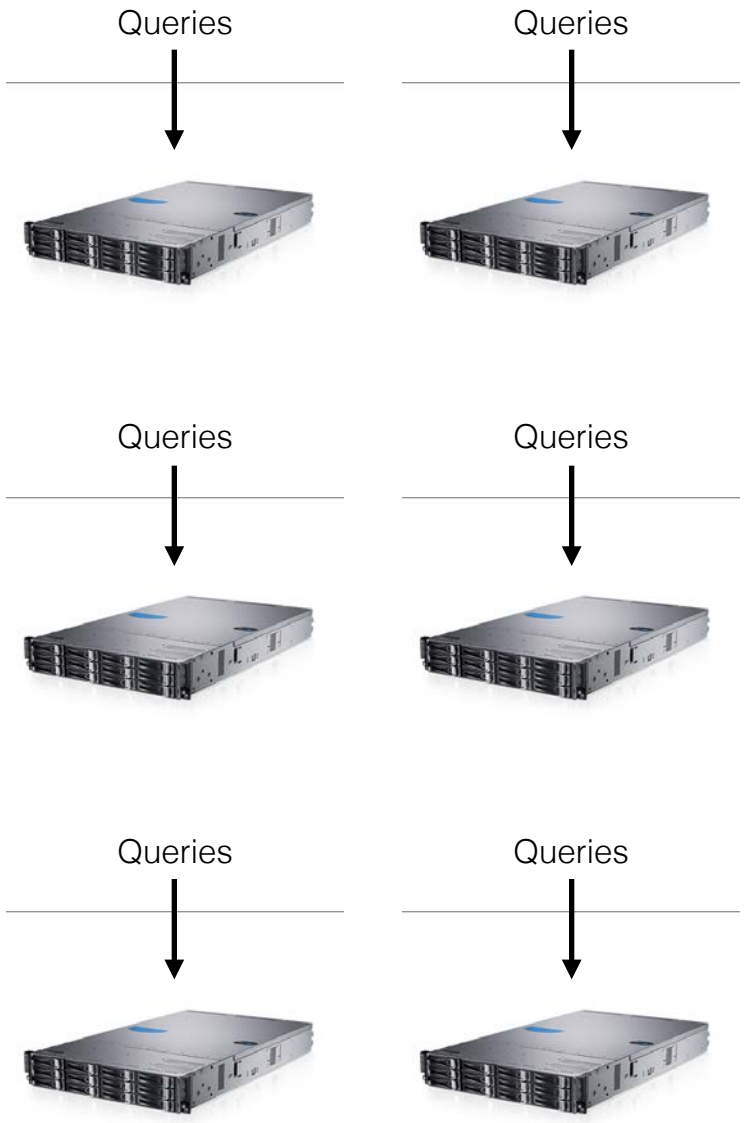


Data Storage Paradigms

- This doesn't work very well with massive data volumes and query throughput.
  - ➡ An index like Google's is simply too big for a standard DBMS to keep up.
  - ➡ Focusing your company around a few large master machines is risky – those machines will fail, and take your whole company offline while you change to a replacement.
  - ➡ Spending millions of dollars per machine on large scale hardware buys you less processing and storage per dollar than commodity hardware.

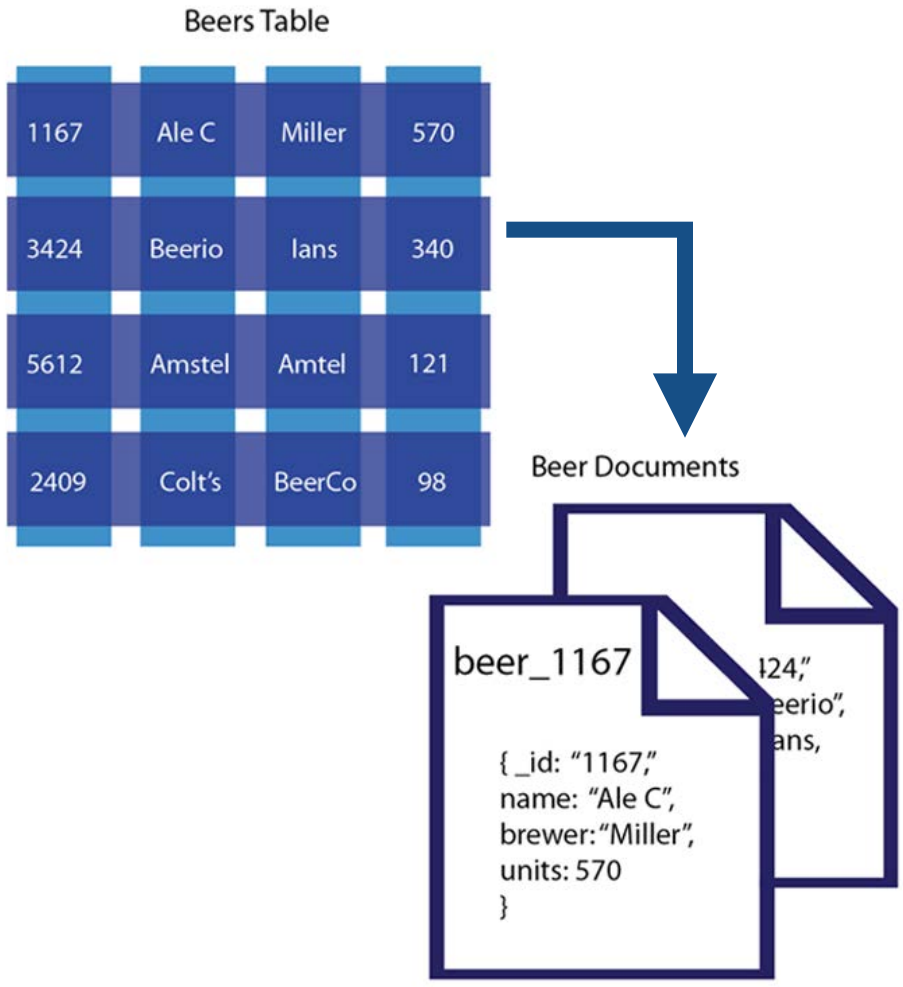
Data Storage Paradigms

- Internet companies tend, instead, to buy huge numbers of cheap, unreliable machines.
- This requires a new type of database software, of which Google's BigTable was an early example.
- The publicly-available products are known as NoSQL, and include MongoDB, Couchbase, and others.
- NoSQL storage is a more natural fit for MapReduce jobs.



Couchbase

- Let's focus on Couchbase as our example.
- Instead of records stored in tables, Couchbase stores a single large, distributed set of (key: value) pairs, which they call "documents."
- There is a single namespace for all keys, so the key typically has a prefix to indicate the type of data stored.



NoSQL, No Schema

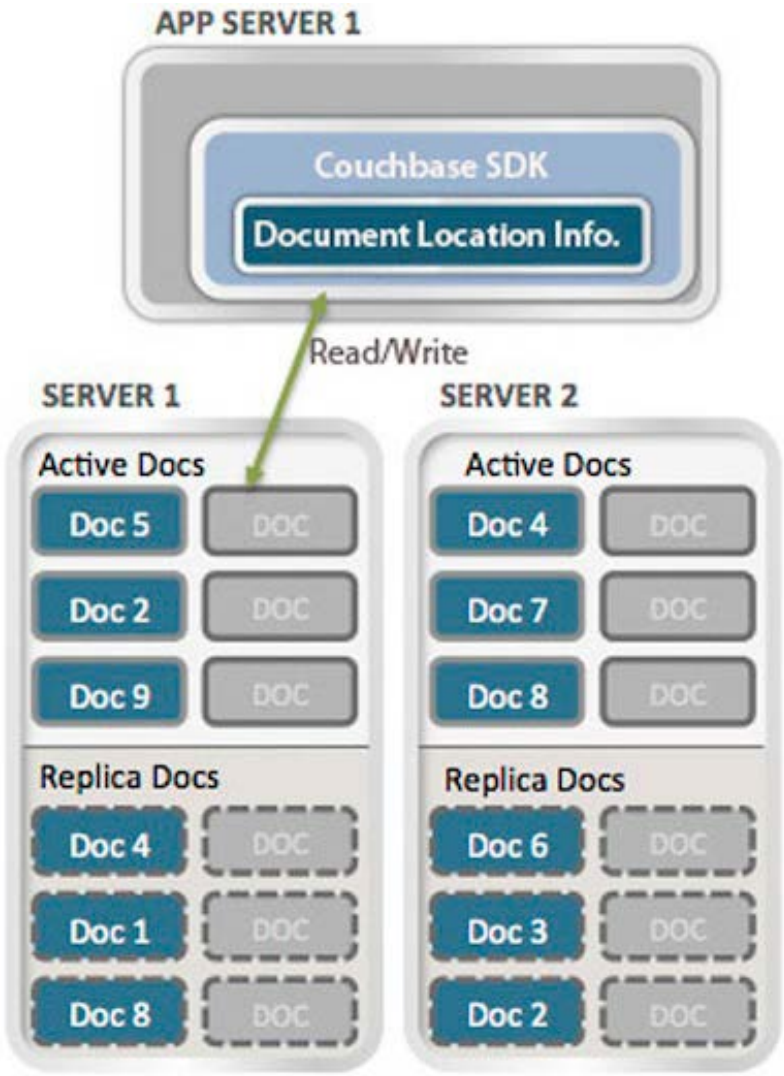
- There is no schema, so different documents can contain different fields. This provides a lot of flexibility, but requires some defensive coding.

- For instance, just add fields when more information, such as translations, is available.

```
{"_id": "post_143",
 "type": "post",
 "author": "Jesse",
 "content": { "en": "I love Google Translate",
              "es": "Me encanta Google Translate",
              "ur": "میں گوگل کے ترجمہ سے محبت",
              "zh": "我爱谷歌翻译"}}
```

Data Replication

- Each document is stored on multiple servers, so if one server fails the system can simply read it from another.
- The client can typically figure out which servers host a given document: the server number used is some deterministic function of the hash code of the key.
- The client connects directly to one of the servers which hosts the document needed, instead of addressing a master server.



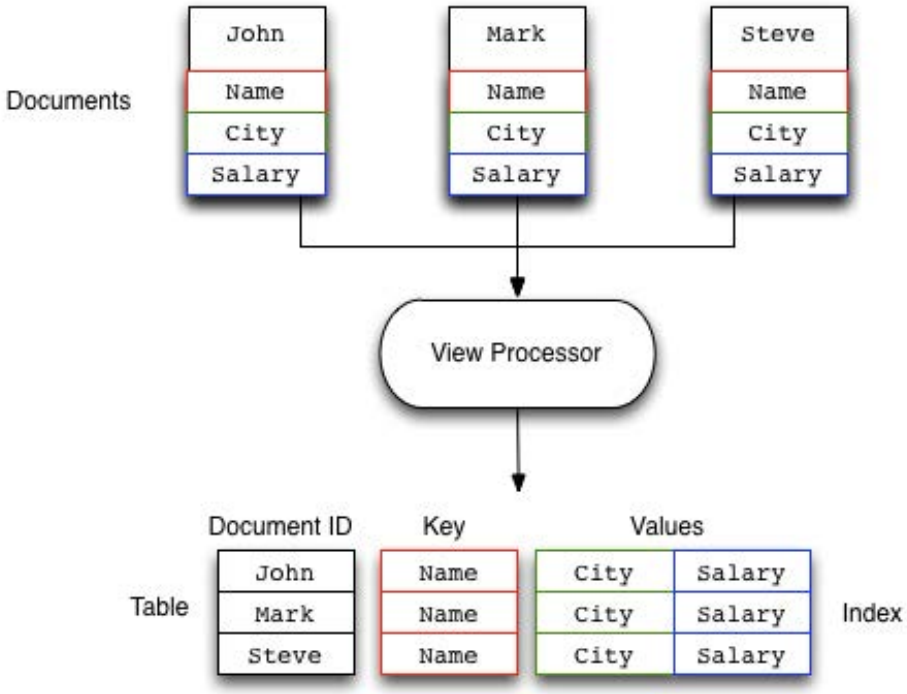
Finding Your Data

- You can get your data by key, and it is common for one document to contain the keys of other documents. This is similar to foreign keys in SQL.
- It is also common for a document to contain another document inside it, as an optimization trick to minimize document read operations.
  - ➡ For instance, you might have a “Top 10 Comments” document, which contains the entire contents of the 10 items listed.
  - ➡ This might make sense if, for instance, you wanted to show the top ten comments every time someone loads your main page: you want to minimize the number of documents read to display the pages with highest load.
  - ➡ This would be strictly forbidden in traditional SQL databases, where data is expected to be normalized.

Couchbase Views

- You can also index your documents for rapid search by arbitrary fields.
- In Couchbase, a view/index is defined by a MapReduce job, written in JavaScript.
- map() transforms stored documents into new rows with different fields and values. It calls emit() to report an output document.
- reduce() can be used to calculate aggregate values (e.g. sum, count, min, max, etc.)

```
// map() – find city and salary by name
function(doc, meta) {
  emit(doc.name, [doc.city, doc.salary]);
}
```



NoSQL for IR

- Say you wanted to store an inverted index in Couchbase. You might do something like this:
  - ➡ Store each crawled document's raw content and properties in a "raw_DOCID" document.
  - ➡ Once the document has been normalized and tokenized, store the result in a "doc_DOCID" document. These documents would also contain document-level features, such as PageRank, a spamminess score, etc.
  - ➡ Store the inverted list for each term in a series of documents, sorted from highest to lowest matching score.

```
"raw_doc23452": {
  "url": "http://www.facebook.com/...",
  "crawled": "2014-11-13 10:34:23 UDT",
  "content": "<html><head><title>...",
  ...
}

"doc_doc23452": {
  "length": 253,
  "terms": ["cats", "are", "eating", ...],
  "terms_stemmed": ["cat", "are", "eat", ...],
  "pagerank": 13.44652143,
  ...
}

"term_cat_0": {
  "docs": {"doc_doc23452": 0.2304,
           "doc_doc23412": 0.00123,
           ...}}
```

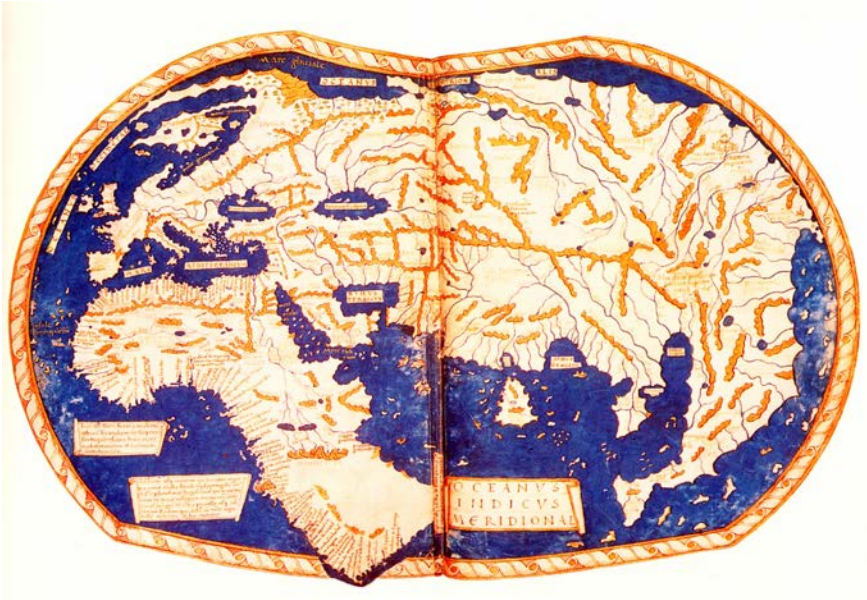
Task Distribution

Ranking a Feed | Making a Suggestion
Data Storage at Scale | **Task Distribution**

MapReduce

- When your data is distributed across many hosts, you also want your software to be similarly distributed.

- MapReduce is a popular software paradigm for distributing your work effectively across machines. It pairs especially well with NoSQL style data storage systems.



MapReduce Roots

- The MapReduce framework has its origins in functional programming, where the map and reduce functions are standard tools.
- map transforms each item in a list, and reduce combines the results into some aggregate value.
- An example in Python:

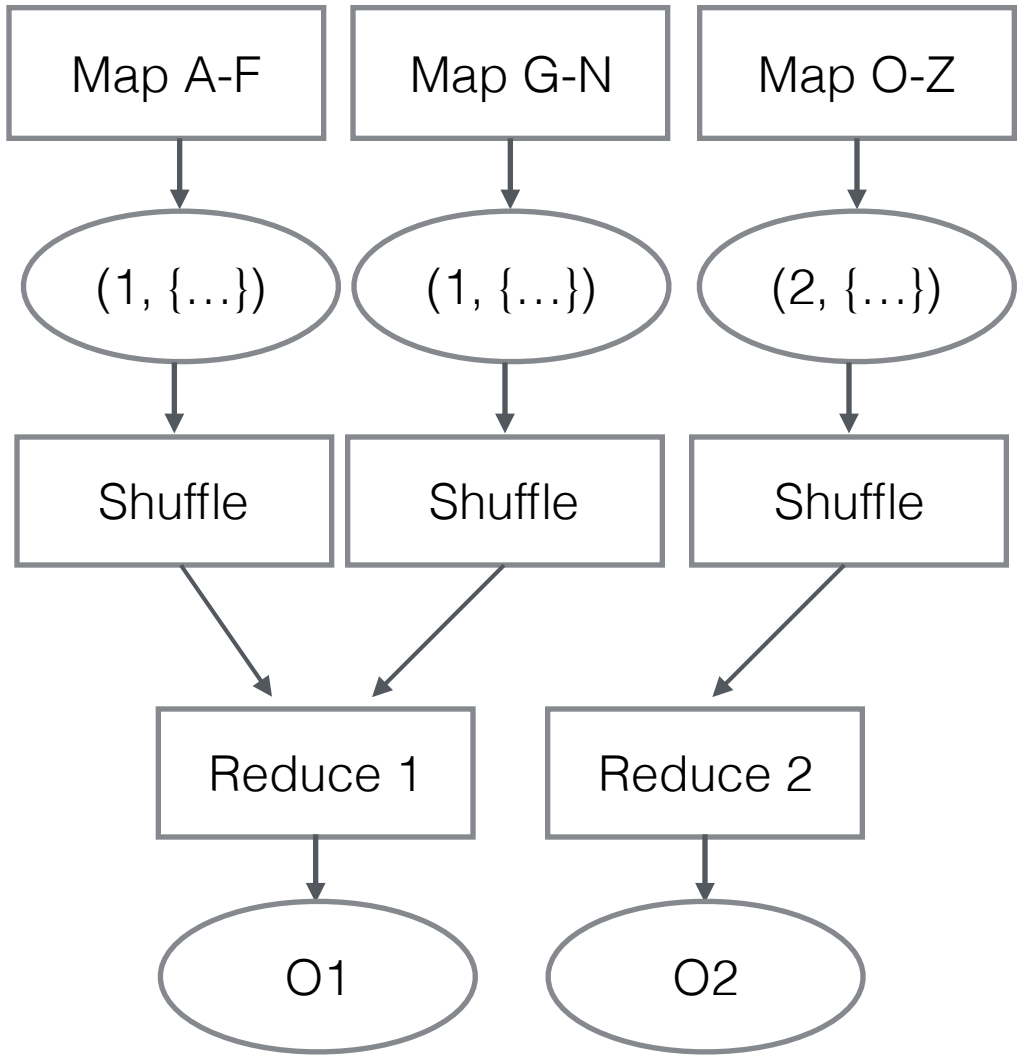
```
In [1]: x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [2]: map(lambda i: i*i, x)
Out[2]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

In [3]: reduce(lambda x,y: x+y, map(lambda i: i*i, x))
Out[3]: 385
```

MapReduce Workflow

- The main program will send a job to the MapReduce system, identifying the map, shuffle, and reduce functions as well as the data to operate over.
- The MapReduce system creates Map jobs on servers in your data center, ideally choosing servers close to the data the jobs will read.
- Each Map job emits (key, value) pairs, which are sent by a Shuffle algorithm to the appropriate reduce job.
- Each Reduce job gets a sequence of all the records with a particular key. The job combines the data and stores the result.



Ex: Simple Indexing

- You could create simple inverted lists with the following jobs.

```
def map(docid, document):
  # docid: document ID
  # document: document contents
  for word, position in tokenize(document):
    emit(word, (docid, position))
```

```
def reduce(word, positions):
  # word: a word
  # positions: a list of (docid, pos) tuples
  invList = []
  for docid, position in positions:
    invList.append((docid, position))
 emit (word, invList)
```

Further Details

- Of course, it's often more complicated than that.
- How do we partition data across map jobs?
  - ➡ An *input reader* is told the entire data set you wish to operate on.
  - ➡ The input reader then divides the data set into smaller chunks, ideally with each chunk living on a single host in your data farm.
  - ➡ The input reader will create the map jobs, read the data from storage, and invoke map as needed.
- Similarly, an output writer stores the results of the reduce jobs.

Managing State

- Remember that we are running on a large number of cheap computers. They will break, at times, while your job is running on them.
- When a machine fails, we want to be able to restart any Map or Reduce jobs that were running on it without affecting the correctness of the program.
- Map and Reduce jobs should not modify any external resources, because they may be run many times on the same data.
- Ideally, they shouldn't even read external resources. They should just express a deterministic transformation from input to output.

What can be MapReduced?

- The MapReduce framework relies on the divide and conquer approach.
- It works best for problems that are neatly separable, such as document indexing.
- It is harder to use it for programs that use large amounts of shared state, such as many dynamic programming tasks.
- It is also vulnerable to network latency issues, so it's important that an individual map or reduce job be large enough to be worth the cost of distributing the task.

Summary

- MapReduce is a good choice for large scale programs that fit the divide and conquer paradigm.
- NoSQL data storage systems were designed with MapReduce in mind.
- The key is to think of each Map and Reduce job as a deterministic transformation from input to output. Experience with a functional language is helpful to learn the paradigm.