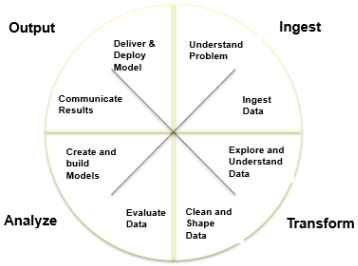
CS4225 Cheatsheet

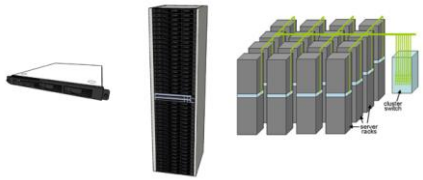
Data Science Challenges
- Volume, velocity (speed), variety (different structures of data), veracity (uncertainty of data (data precision, uncertainty)), value (insights from data)

Data Lifecycle



Main infrastructure for Big Data? **Cloud Computing**

Building Blocks of Cloud Computer Infrastructure: Commodity Machines



- Blade server: Commodity hardware
- Rack: Consists of dozens of servers
- Data centers: Consists of hundreds of racks

Bandwidth vs Latency

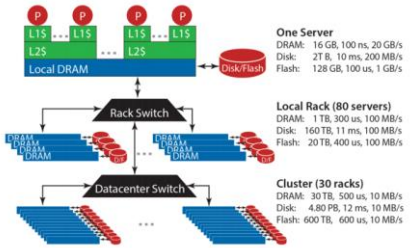
Bandwidth: Max amount of data that can be transmitted per unit time (Gb/s for e.g.)

Latency: Time taken for 1 packet to go from src to destination (one-way) or from source to destination back to src (round-trip), e.g. in ms

→ Bandwidth tells us roughly how long the transmission will take.

→Latency tells us how much delay there will be.

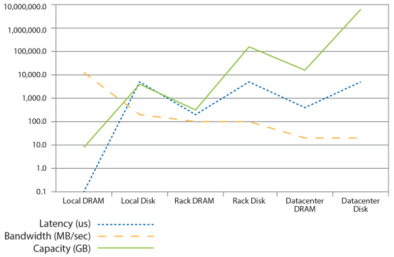
Storage Hierarchy



→ Observe the speed of the dram/disk/flash in different hierarchy levels, at rack level and datacenter level, speed is bounded by the network bandwidth

→ Note: Rack dram/disk is referring to transferring bytes from machine a to b in same rack

Datacenter dram/disk is referring to transferring bytes from machine a to b in different rack



→ Look at yellow dashed line, bandwidth. Network is the bottleneck, Which is why we see a decreasing trend. (Look at storage hierarchy figure and the transmission speeds). Even though bandwidth of a rack > bandwidth of one machine, many machines are sharing one data center switch so it doesn't just add up across all machines

→ For solid green line, capacity, Disk > RAM at each level, and since the capacities are aggregated at each level, capacities have upward trend

→ For dotted blue line, latency, disk latency < dram latency at every level. Local RAM latency very low, but increases a lot once at rack level

→ Latency: DRAM < FLASH < DISK

→ Ops/Sec: DRAM > FLASH > DISK

→ Cost: DRAM > FLASH > DISK

4 Big Ideas of Data Science
- Scale "out" not "up"

→ Large shared-memory machines have limitations, so increase number of machines instead

- Move processing to the data

→ Don't move the data so much as bandwidth is low across machines/racks and therefore costly

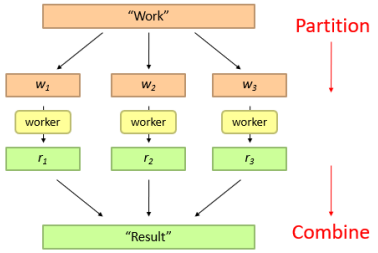
- Process data sequentially, avoid random access

→ Reduce i/o cost, seeks are expensive

- Seamless scalability (2x more machine → 2x more performance)

MapReduce Introduction

→ Idea: Divide and Conquer in big data to increase parallelization and improve processing speed

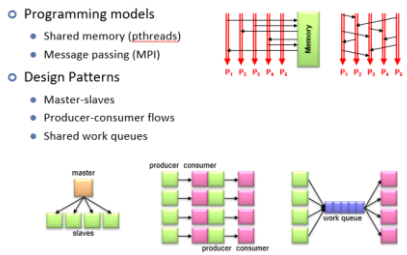


Parallelization Challenges
- How do we assign work units to workers?
- What if we have more work units than workers (Scheduling)
- What if workers need to share partial results (communicate between workers)
- How do we aggregate partial results
- How do we know all workers have finished?
- What if workers die/fail?

→ Need synchronisation mechanism to communicate between workers/access shared resources

Managing multiple workers

Current Tools



Challenges with concurrency
- Difficult to reason, even more so at scale of datacenters and across datacenters, with the presence of failure, with multiple interacting services
- Need to debug not just for correctness, but performance

MapReduce

Map: (k1,v1) -> [<k2,v2>]

Reduce: (k2, [v2]) -> [<k3,v3>]

MapReduce framework/runtime
- Handles scheduling

→ Assigns workers to map and reduce tasks

- Handles data distribution

→Moves processes to data

- Handles Synchronisation

→ Gathers, sorts, shuffles intermediate data

- Handles errors and faults

→ Detect worker failures and restarts

- Distribute file system

→ HDFS

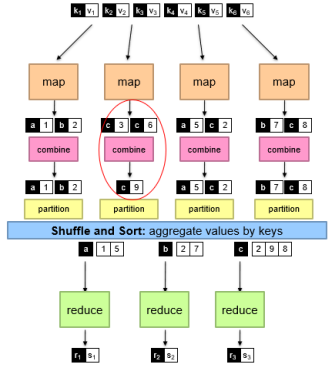
- Programmers specify 2 functions: map and reduce
- Specify 2 other optional functions: Partition and combine

Partitioner: (k', number of partitions) -> partition key by hashing to divide key space to several buckets for parallel execution

Combiner: (k',v') -> <k',v'>

→ Mini-reducers that run **in memory** after map phase

→ Used as optimization to reduce network traffic



→ Local combiner that is applied in the local machine before being sent to be partitioned.

→ **Important:** Return <k,v> of mapper must be the same as return <k,v> of combiner, because there may be instances where combiner is not run, and we need consistent input into the reducer! (double check tutorial 1)

→ With combiner, we send instead of the 2k/v pairs that need to be sent, we can send 1 aggregated k/v

Features of Hadoop
- Keys arrive at each reducer in sorted order, **no enforced ordering across reducers,** values are also not guaranteed to be sorted

Distributed File System

→ Move workers to the data by storing data on the local disks of nodes in the cluster. Start up the workers on the node that has the data local alrd

→ Why? Data center network is slow!

→ It needs to be distributed since it is very possible that entire data set > size of dram

→ While disk access is slow, its aggregated bandwidth is reasonable

DFS/GFS (Google File System): Assumptions
- Commodity hardware over exotic hardware, scale "out", not "up"
- High component failure rates

→Inexpensive commodity hardware fail a lot

- "Modest" number of huge files

→ Multi-gigabyte files common

- Files are write-once, mostly appended to, perhaps concurrently, throughput > latency
- Large streaming reads preferred over random-access (?) → High sustained throughput over low latency

GFS/HDFS: Design Decisions

1. File stored as chunks

→ Fixed size (64MB). Why chunks? Large chunks reduce clients' need to interact with master (reads/writes to same chunk only require 1 initial request, reduce network overhead by keeping persistent connection with 1 chunkserver).

2. Reliability through replication

→ Each chunk replicated across 3+ chunkservers

→ Why 3? We can use majority (2 out of 3) to determine which file had a bitflip, increasing robustness.

3. Single master to coordinate access, keep metadata

→ Simple centralized management

4. No data caching

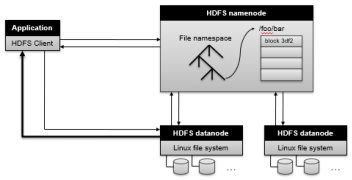
→ Little benefit since we have streaming reads on large dataset

5. Simplify the API

→ Push some of the issues onto client (e.g. data layout)

→ GFS master/chunkservers = Hadoop namenode/datanode

HDFS Architecture



→ Master & Slave architecture

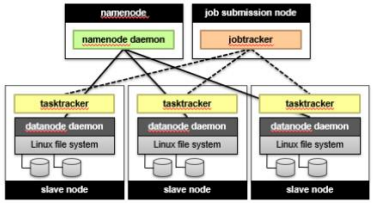
→ Client requests for file from namenode, namenode will tell client which datanode to query. Interaction is from client to name node and client to datanode.

→ Datanode and namenode communication includes health check, synchronisation

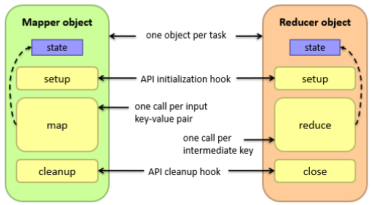
Namenode Responsibilities

→ Manage filesystem namespace: Hold file/directory structure, metadata, file-to-block mapping, access permissions

→ Coordinate file operations: Direct clients to datanodes for reads/writes. **No data is moved** through the namenode
→ Maintaining Overall health: Periodic communication with datanodes, block re-replication and rebalancing, garbage collection



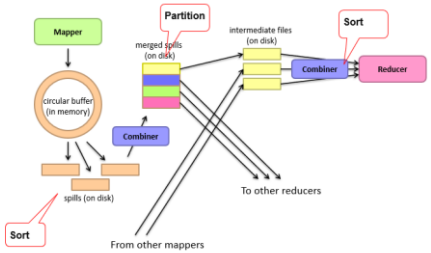
Tools for synchronization in MapReduce
- Combine partial results together
- Sort order of intermediate keys. Control order in which reducers process keys
- Partitioner: Control which reducer processes which keys ("Secondary sorting" done here I think)
- Preserving state in mappers and reducers – capture dependencies across multiple keys and values



→ Basically means we can use data structure/variable as attribute in the mapper/reducer class, it will update with each function call

Performance Guideline in Designing MapReduce job
- Linear scalability: More nodes = more work in same time, linear on data size and computer resources
- Minimize the amount of I/Os in hard disk and network.
→ Minimize disk I/O; prefer sequential over random
→ Minimize network I/O; bulk send/receive over many small sends/receive
- Memory working set of each task/worker
→ Lower memory working set, since larger memory working set → high probability of failure

Importance of local aggregation
- Ideally, we want linear scaling: Twice the data, twice the running time, twice the resources, half the running time
- However, synchronization/communication kills performance
- Reduce communication by reducing intermediate data via local aggregation (Combiners)



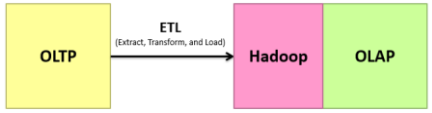
→ Circular buffer in memory to maintain partial results before writing to disk.
→ Map: Before map writes to disk, the thread divides the data into partitions based on which reducer it goes to. Within each partition, background thread performs in-memory sort by key
→ Shuffle: Guarantees input to every reducer is sorted by key
→ Sort: Sort happens in various stages of MapReduce, can exist in map and reduce phases
→ Reduce: Sort phase which merges the map outputs and maintain sort ordering before reduction is performed I think?

Design Pattern for local aggregation
- "In mapper" combining
→ Fold functionality of combiner into mapper by preserving state across multiple map calls (Use global data structure in class)
+ Speed, faster than combiners
- Requires sufficient memory from local machine
- Explicit memory management required
- Potential for order-dependent bugs

Combiner Design
- Combiners and reducer can share same method signature, return <k,v> of mapper must be the same as return <k,v> of combiner
- Combiners are optional optimizations and **should not** affect algorithm correctness. They may not even be run

MapReduce – Relational Data
OLTP – Online transaction processing – facilitate and manage transaction-oriented applications (like processing e-commerce transactions)

OLAP – Online analytical processing, answers multi-dimensional analytical queries (like doing reporting/ranking/prediction)



Relational Algebra operations in MapReduce
Projection/Selection
- Map over tuples, emit new tuples with appropriate attributes/meets condition, no reducers unless we want to sort/regroup tuples
- Can also perform picking in reducer
- Limited by HDFS streaming speed (encoding / decoding of tuples). I/O is the key bottleneck. Can use compression to reduce I/O (???)

Group by.. Aggregation
- E.g. for question "What is the avg time spent per URL" i.e. SELECT url, AVG(time) from visits group by url
- Map over tuples, emit <url, time>
- Compute average in reducer, optimize with combiners.
→ Can also emit <url, timeSum count> and compute average

Order By
- By single column – automatically done by mapreduce. Just put it as key in mapper
- Multiple column – secondary sorting by creating composite key of <k, v> in key
→ Scalable solution, does not suffer from scalability bottleneck, better than doing in memory sort which could cause reducer to run out of memory. Especially so if the dataset is skewed and most of the data has same key value!
→ E.g. instead of (B) as key, (C, sale) as value, do (B,C) as key, sale as value, used customized comparator function to perform secondary sort

Joins
Reduce-Side Joins
- Basic Idea: Group by join key
- Map, emit tuple as value with join key as intermediate key, framework brings tuples sharing same key together
- Perform join operation in reducer
→1 to 1 joins
- **For 1 to many** (R to S), by using join value as key, we don't know among all tuples, which is the R tuple. We can do in memory sort to get the R tuple out, then cross with every tuple in S, but it creates scalability bottleneck
- Solution? Create composite key, which consists of join key and tuple id, defining sort order for tuple from R to come first. Next, we **define the partitioner** to pay attention to only the join key so that they all arrive at the same reducer.
→No need to buffer tuples except the single one from R to cross with tuples from S. Eliminate scalability bottleneck

Many-to-many reduce-side join
Buffer all tuples with R in memory and join with tuples in S. Not very efficient solution, and would require smaller dataset to come first for it to be more efficient

Map-side Join
- Datasets already sorted by join key, perform join by scanning through both datasets simultaneously
- Partition and sort both datasets in same manner, then scan simultaneously

In-memory join
- Load one dataset into memory, stream over other dataset
→ Works only if R << S, R can fit into memory
- Distribute R to all nodes (each node has full R dataset)
- Map over S, load R in memory, hashmap by join key??
- For every tuple in S, look up join key in R and join tuples. Sounds abit like hash join (with the hashtable strategy in memory)

In-memory join variants
- If R is too big to fit into memory, we can split each R value into separate buckets that all fit into memory. Then perform in memory join for each bucket. Each bucket is 1 mapreduce job. Very expensive, each job is 1 pass of 1/o!
- Take union of all results

Memcached Join
- Load R into Memcached
- Replace in-memory hash lookup with Memcached lookup
- Memcached capacity >> RAM of individual node, Memcached scales out with cluster, latency is speed of network

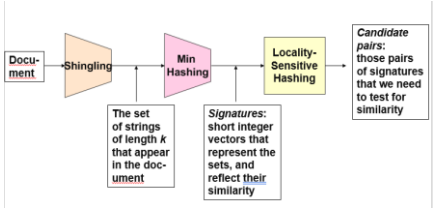
In-memory vs map-side vs reduce-side
- In-memory join > map-side join > reduce-side join in terms of performance
Limitations: **In-memory join**: memory, **map-side join**: datasets must be in sort order and partitioned according to key, otherwise, **reduce-side join**, which is in general the slowest.

Data Mining
- Find similar items using locality sensitive hashing or clustering
- Find "similar" sets: Find near-neighbors in high-dimensional space
- E.g. Pages with similar words, customers who purchased similar products, images with similar features, users who visited similar websites

Distance Measures – Jaccard Similarity
- Size of their intersection divided by the size of their union: sim(C1,C2): $|C1 \cap C2|/|C1 \cup C2|$
Jaccard Distance: $d(C_1, C_2) = 1 - |C_1 \cap C_2|/|C_1 \cup C_2|$

Finding Similar Documents
- Goal: Given N = millions/billions of documents, find near duplicate pairs.
- Challenge: pieces of documents can appear out of order but are talking about the same story. Number of documents can be large, pair-wise comparison is time consuming. Documents large or so many that they cannot be handled efficiently on a single machine

3 Essential steps:
1. Shingling: Convert docs to sets
2. Min-Hashing: Convert large sets to short signatures, while preserving similarity
3. Locality-sensitive hashing: Focus on pairs of signatures likely to be from similar documents (Candidate pairs)



Shingles
- A k-shingle or k-gram is a sequence of k tokens that appear in the document
→ Tokens can be characters, words, etc depending on application
E.g. k=2; document D1 = abcab => set of 2-shingles S(D1) = {ab, bc, ca}
Note: Shingles are sets, ab is only counted once. Can also be used as a bag (multi-set, double count ab)

Compressing Shingles
- Storage overhead and computation overhead with shingles
- Compress by hashing them to (say) 4 bytes
- We can now represent a document by a set of hash values of the k-shingles

- Most likely, 2 different values wont hash to same position. We have 4 bytes, which is already 2^{32} buckets!!
- Better to hash 9-shingles than 4-shingles, else most documents will have many common shingles

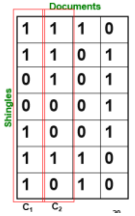
Similarity Metric for Shingles
- Use jaccard similarity to compare
- Must pick large enough **k** value, else most documents will have many common shingles.
→ k = 5 ok for short documents, k = 10 better for long documents

Minhashing/Locality Sensitive hashing
- With all the k-shingles, we have to compute pairwise jaccard similarities for every doc, it is going to take forever
- Convert large sets to short signatures, while preserving similarity

1. Encode sets using 0/1 (bit,Boolean) vectors, 1 dimension per element in the universal set → Interpret set intersection using bitwise AND and set union using bitwise OR (O(1) operations!)

2. Arrange vectors in Boolean matrices
From Sets to Boolean Matrices

- **Rows** = elements (shingles)
- **Columns** = sets (documents)
  - 1 in row *e* and column *s* if and only if *e* is a member of *s*
  - Column similarity is the Jaccard similarity of the corresponding sets (rows with value *1*)
  - Typical matrix is sparse!
- **Each document is a column:**
  - **Example: sim(C_1 ,C_2) = ?**
    - Size of intersection = 3; size of union = 6, Jaccard similarity (not distance) = 3/6
    - **d(C_1,C_2) = 1 – (Jaccard similarity) = 3/6**



→ One line down belongs to a document (Each document is a column)

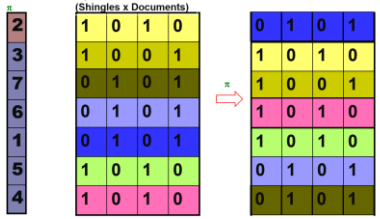
3. Goal: Find similar columns while computing small signatures: Similarity of columns == similarity of signatures
- Naïve approach: Comparing all pairs will take a lot of time. Perform **LSH**
- Idea: Hash each column C into a small signature h(C), such that:
→ h(C) is small enough that the signature fits in RAM
→ sim(C1,C2) is the same as similarity of signatures h(C1) and h(C2)
→ We want a hash function such that
- If ***sim(C_1,C_2)*** is high, then with high prob. ***h(C_1) = h(C_2)***
- If ***sim(C_1,C_2)*** is low, then with high prob. ***h(C_1) ≠ h(C_2)***

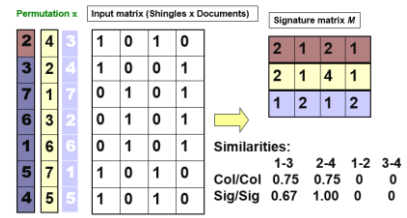
Min-Hashing
- Permute rows of the Boolean matrix under random permutation π
→ ***π(i)=j means that the ith row of input matrix will become the jth row of the permuted matrix.***
- Define a **"hash" function $h_\pi(C)$** = the index of the **first** (in the permuted order π) row in which column ***C*** has value **1**:

$$h_\pi(C) = min_\pi \pi(C)$$

- One permutation is a hash function, use several independent hash functions (i.e. several permutations) to create signature of a column.

Min-Hashing Example





Min-Hash Property
Claim: **Pr[$h_\pi(C_1) = h_\pi(C_2)$] = *sim*(C_1, C_2)**
(Proof is not so important in this mod, but idea is that the ratio of rows that determine sim(C1,C2) is the probability that h(C1) = h(C2))

→ Expected similarity of two signatures is equal to the Jaccard similarity of the columns. The longer the signatures, the lower the error
→ The more permutations you use, the accurate the estimate (Refer to image above).
→ sig(C) = column vector
→ sig(C)[i] = index of the first row that has a 1 in column C of the ith permutation
With k = 100 random permutation of rows,

$$sig(C)[i] = \min(\pi_i(C))$$

- **Note:** The sketch (signature) of document *C* is small **~100 bytes!**

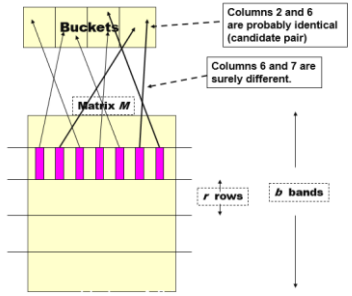
→ Compressed long bit vectors into short signatures.

Locality Sensitive Hashing
→ Goal: We want a function f(x,y) that tells us whether x and y is a candidate pair: a pair of elements whose similarity must be evaluated.
→ With the new columns of the signature matrix M that we have created, hash bands of columns, and bands that fall into same buckets are candidate pairs

Candidates from Min-Hash
- Pick a similarity threshold s (0<s<1)
- Columns x and y of M are candidate pair if their signatures agree on at least fraction s of their rows: M(I,x) = M(I, y), where i is all rows, then they are candidate pairs if the equation above holds for at least fraction s of the I rows
→ We expect documents x and y to have the same (Jaccard) similarity as their signatures

Partitioning Signature matrix M into b bands

Hashing Bands

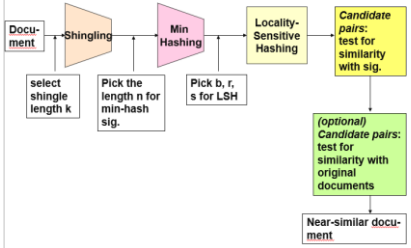


→ After u split into bands (split columns), we compare the rows pairwise in each band by hashing each row (i.e. it isn't really comparing). We know that if they are in different buckets, they are definitely different, if they are the same, they are most likely identical

Assumptions in LSH
- There are enough buckets that columns are unlikely to hash to the same bucket unless they are identical in a particular band (assume same bucket = identical in that band)

Putting it together:

Put it all together



MapReduce implementation
Shingling/Min-hashing:
Map: Parse document into shingles, perform row hashing, emit <document id, min hash signature)
Reduce: none

LSH:
Map: Perform band hashing, emit <band id + bucket id, document id>
Reduce: Emit candidate pairs. All similar document ids are alr in the same bucket

Parallelism limited by #bands if we had emitted <bandid, documentid + min hash signature> instead, before performing LSH in each bucket.

Clustering
Problem Domain: We have a cloud of high dimensional data points, where we want to understand its structure.
We might want to visualize the data points. We want to put the points into clusters and study the details of each cluster.
Similarity can be defined using a distance measure such as L2 norm (Euclidean), cosine, jaccard..

Problem with clustering
→ Aim: We want to group a points into some number of clusters. These points have a notion of distance between the points, and we want members of a cluster to be close/similar to each other, while members of different clusters should be dissimilar.

Distances – Jaccard

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$
$$d(A,B) = 1 - J(A,B)$$

Euclidean Distance
- Euclidean distance (L_2-norm)

$$d(x,y) = \sqrt{\sum_{i=0}^{n} (x_i - y_i)^2}$$

- Other Norms: Manhattan distance (L_1-norm), L_r-norm

Cosine Distance
- Idea: measure distance between the vectors

$$\cos\theta = \frac{x \cdot y}{|x||y|}$$

- Thus:

$$sim(x,y) = \frac{\sum_{i=0}^{n} x_i y_i}{\sqrt{\sum_{i=0}^{n} x_i^2}\sqrt{\sum_{i=0}^{n} y_i^2}}$$
$$d(x,y) = 1 - sim(x,y)$$

Edit Distance
- How many edits are needed to match the target with the pattern
E.g. Target: TCGACGTCA, Pattern : TGACGTGC
By Deleting C and A from the target, and by Deleting G from the Pattern – Distance of 3

Outliers
Some points might not belong to any clusters and be outliers. Moreover, clustering is a hard problem and some clusters might even overlap with each other.
Why is clustering hard
- Dimensionality issue – applications might have 10,000 dimensions. Curse of dimensionality – almost all pairs of points might have approximately same distance in high distance.
- High volume of data points to cluster.

Example problems:
Galaxies – Catalog of 2 billion sky objects, represents objects by their radiations in 7 dimensions (freq bands). Problem: Cluster into similar objects (galaxies, nearby stars, quasars)
Music CDs – Music can be divided into categories, customers usually prefer a small subset of categories. We can represent a cluster as a "CD" by a set of customers who bought the set of music. → Similar set of customers will be in 1 cluster, and similar clusters will have similar sets of customers

Space of all CDs:
- Think of a space with one dim. for each customer
  - Values in a dimension may be 0 or 1 only
  - A CD is a point in this space (x_1, x_2,..., x_k), where x_i = 1 iff the i^{th} customer bought the CD
- For Amazon, the dimension is tens of millions
- **Task:** Find clusters of similar CDs

Clustering Problem: Documents
Represent document as vector (x_1, x_2,..., x_k), where x_i = 1 iff the i^{th} word (in some order) appears in the document.
→ Documents with similar sets of words/shingles may be about same topic

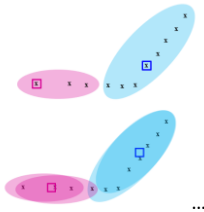
Which distance metric to use?
Depends on how we represent the points we use different distance metrics.
- **Sets as vectors:** Measure similarity by the **cosine distance**
- **Sets as sets:** Measure similarity by the **Jaccard distance**
- **Sets as points:** Measure similarity by **Euclidean distance**

Methods of Clustering
Hierarchical:
1. Agglomerative (cluster from the bottom up): Each point is a cluster initially, repeatedly combine the two "nearest" clusters into one
2. Divisive (clustering from the top down): Start with 1 cluster, recursively split it.

Point Assignment:
- We maintain a set of clusters and assign points to the "nearest" cluster. Typically, this process is preceded by a phase where clusters are initiated.

K-means clustering
- Use Euclidean distance to determine distances.
- Pick k, the number of clusters.
- Initialize clusters by picking one random point per cluster (k points).
2 ways to pick points:
1. We pick each point to be as far away from each other as possible.
2. Cluster **a sample** of the data until there are k clusters, pick a point from that cluster (e.g. centroid of the cluster)

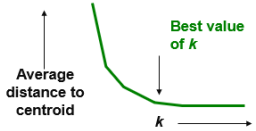
K means clustering algorithm
1. For each point, place it in the cluster whose centroid is the nearest to the point.
2. Once we assign all points, we update the new centroid of the clusters by taking the average value of all points in that cluster.
3. Reassign the points to their new nearest centroid.
4. Repeat step 2 and 3 until convergence (the points don't move between clusters and centroids stabilize).
E.g.:



to…

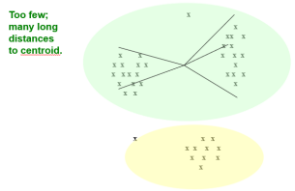
… and so on.

Selecting the best k value
- Try different k, look at chance in average distance to centroid as k increases. Elbow method: The average falls rapidly until the right k before only changing a little.

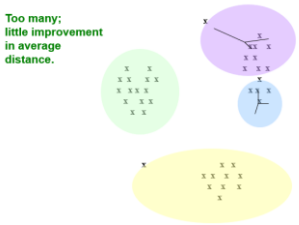


→ Possible strategy: Run k-means for k = 1,2,4,8… and eventually we find 2 k values in which there is very little decrease in the average diameter.

Too little k:



Too many k:



MapReduce Implementation for K-means clustering

```
1: class Mapper
2:   method Configure()
3:     c ← LoadClusters()
4:   method Map(id i, point p)
5:     n ← NearestClusterID(clusters c, point p)
6:     p ← ExtendPoint(point p)
7:     Emit(clusterid n, point p)
1: class Reducer
2:   method Reduce(clusterid n, points [p1, p2, ...])
3:     s ← InitPointSum()
4:     for all point p ∈ points do
5:       s ← s + p
6:     m ← ComputeCentroid(point s)
7:     Emit(clusterid n, centroid m)
```

Network traffic!

→ For each point, find out which is the nearest cluster.
→ ExtendPoint will extend the point by 1 dimension, to store the count?
→ Then we can emit based on the cluster id as key, which mapreduce will group together. We can then calculate the new centroid value in the reduce phase. Repeat with multiple jobs until convergence.

Problem? The parallelism is bounded by the number of clusters only in the reduce phase! Moreover, what if there is a skew and all the points go to one cluster? If it is a big K, it will be fine, but small K is a problem.

MapReduce Implementation using In-mapper Combiner

```
1: class Mapper
2:   method Configure()
3:     c ← LoadClusters()
4:     H ← InitAssociativeArray()
5:   method Map(id i, point p)
6:     n ← NearestClusterID(clusters c, point p)
7:     p ← ExtendPoint(point p)
8:     H{n} ← H{n} + p
9:   method Close()
10:    for all clusterid n ∈ H do
11:      Emit(clusterid n, point H{n})
1: class Reducer
2:   method Reduce(clusterid n, points [p1, p2, ...])
3:     s ← InitPointSum()
4:     for all point p ∈ points do
5:       s ← s + p
6:     m ← ComputeCentroid(point s)
7:     Emit(clusterid n, centroid m)
```

→ Use a hashmap in memory with cluster id as key and an accumulated cluster point as the value (I assume we use the extended point which has a count, and the point coordinates itself to continually calculate a streamed average which can be used to return an "aggregated point" in the close method()?
→ Compute new centroid in the reducer method.
→ **All in all,** mapreduce does not support this algorithm very well, we might need better support for iterations as multiple jobs are needed to complete this algorithm, which requires a lot of I/O.
→ Also, must keep centroids in memory, and with large k/feature space, this might be an issue.

NoSQL systems
- MapReduce is a step backward to databases, because schemas, separation of schema from the application and high-level access languages (SQL) are all good
- MapReduce is largely brute force without cost saving data structures like indexes
- Incompatible with DBMS tools, missing indexes, updates transactions, bulk loader…
- Major bottleneck in Hadoop – i/o and data format parsing

Why are schemas good?
- Parsing fields out of flat text files is slow
- Schemas define contract and decouple logical from physical, underlying implementation can vary across systems. Without schema, we need to manually parse files for data which is much slower

Thrift – Developed by FB
- Provides a data definition language (DDL) with numerous language bindings, compact binary encoding of typed structs, compiler automatically generates code for manipulating messages
- Provides RPC (Remote procedure call a remote procedure call is when a computer program causes a procedure to execute in a different address space) mechanisms for service definitions
Alternatives: Protobuf and Avro

Column Stores
- Defacto storage format in DBMS OLAP (Online analytical processing). Because usually, a query does not touch all columns, so compared to row stores, column stores will only read necessary columns.

Advantages of Column Stores
- Read efficiency – Read columns necessary for the query
- Better compression – Each column has same data type
- Vectorized processing
- Opportunities to operate directly on compressed data

Indexes in Hadoop
- Not invasive, don't require changes to Hadoop infrastructure, useful for speeding up selections or joins, indexing building itself can be performed using mapreduce.

Cons of MapReduce?
- Iterative, java verbosity
- **High Overhead at startup,** designed for batch processing
- Stragglers might slow down entire job. They occur because some commodity machine might have lower computing power
- Needless data shuffling, checkpointing at each iteration
- Proposal to add a looping mechanism to mapreduce that the job can be aware of (so that we don't have to do 1 iteration → 1 job)

Why is high-level language like SQL good?
- While Hadoop is great for large-data processing (batch processing), writing java programs is too verbose, need a language that is already well known
- SQL

Hive, Pig
- Hive: Data warehousing app in Hadoop, query language is HQL (variant of SQL), tables stored on HDFS with different encodings.
- Pig: High level language (I assume its declarative like sql?) For developers to focus on data transformations, which compiles down to Hadoop jobs

Hive: Example
- Similar to SQL DB

```
SELECT s.word, s.freq, k.freq FROM shakespeare s
  JOIN bible k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1
  ORDER BY s.freq DESC LIMIT 10;

the     25848   62394
I       23031   8854
and     19671   38985
to      18038   13526
of      16700   34654
a       14170   8057
```

→ Compiles into mapreduce jobs
→ Metastore holds metadata, such as DB, tables, schemas, access control/permission info
→ Hive data is stored on HDFS (Tables stored in directories, partitions on tables in sub directories, actual data in files)

Yarn (Yet another resource negotiator)
API to develop any generic distribution application
Handle scheduling and resource requests
- Provide resource management and a central platform to deliver consistent operations, security and data governance tools
- Allows support of other distributed frameworks

Spark
For large-scale batch processing, supports generalized dataflows (java, python, scala), heavily relies on memory for computation

Other Requirements of NoSQL systems different from SQL
- Able to horizontally scale "simple operations"
- replicate/distribute data over many servers
- Have a simple call interface
- Don't need such a strict concurrency model like ACID
- Use of distributed indexes and RAM, flexible schemas.
- E.g. KV stores (Redis), column oriented DBs (Cassandra), document stores (MongoDB), graph databases (neo4j)

Key-Value Stores: Data Model
- Keys usually primitives: int, string, bytes
- Values can be primitive or complex: ints, strings, json, html
- Simple API**:** get – fetch value of key, put: set value associated with key
→ Other operations like multi-get, multi-put, range queries
Consistency Model: Atomic puts

Implementation of key value stores
→ Non-persistent: Big in-memory hashtable (Memcached, redis)
→ Persistent: Wrapper around traditional RDBMS, rocksDB, dynamo, riak.

Document Stores
- Database can have multiple collections, collections have multiple documents. Document is like a JSON object, has fields and values. Different documents can have different fields.
- Can be nested: JSON objects as values
→ MongoDB
→ Document stores allow some querying based on content of a document.

MongoDB Examples

CRUD: Read

In MongoDB

```
db.users.find(                          ← collection
    { age: { $gt: 18 } },               ← query criteria
    { name: 1, address: 1 }             ← projection
).limit(5)                              ← cursor modifier
```

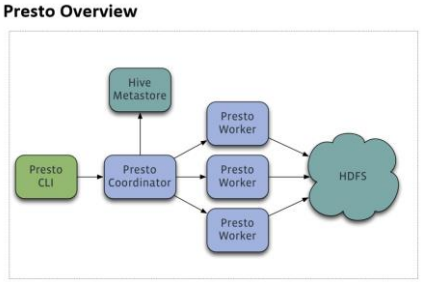
In SQL

```
SELECT _id, name, address   ← projection
FROM   users                ← table
WHERE  age > 18             ← select criteria
LIMIT  5                    ← cursor modifier
```

Wide Column Stores
- Related groups of columns grouped as column families
- rows describe entities
- E.g Cassandra, BigTable

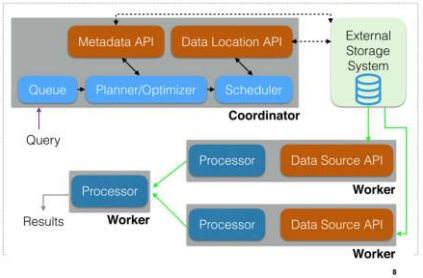
Presto
- Open source distributed SQL query engine. Designed to be extensible by providing a versatile plugin interface.

Overview:



→ Good for long running batch ETL (extract-transform-load) jobs.
→ Provide SQL interface to multiple internal NoSQL systems.

Presto Architecture:



Introduction to Spark

Hadoop vs Spark: Hadoop was some issues.
→ Network and I/O costs. Hadoop unnecessarily shuffles data and saves intermediate results to local disks, which is slow
→ Not suitable for iterative: Each iteration is done as one separate mapreduce job since mapreduce is not designed to perform iterative tasks, thus there is high overhead
→ Verbose: MapReduce can only be programmed in Java with rigid interfaces.

Spark Advantages over MapReduce
→ Programmability in Java/Scala/Python, allows functional transformation on collections, and requires much less code vs MapReduce
→ Performance: General DAG of tasks, in-memory cache, potentially runs much faster than MapReduce
→ Core Idea: In Spark, most of the intermediate results are stored in memory, making it much faster for iterative processing, and only spills to disk when memory is insufficient

Spark Architecture
→ Master-slave architecture
→ Driver Process: respond to user input, manage spark application, distribute work back to executors (which run the code and return results back to driver)
→ Cluster Manager – Allocates resources when application requests for it. (YARN, Kubernetes)

Spark APIs
→ RDDs, Dataframes, Dataset
RDDs – Resilient Distributed Datasets. Basically a collection of Java objects. We are allowed to perform function operations on it (map, flatmap etc)
DataFrame – Collection of Row objects, row objects can have underlying schema I think?
DataSet – Internally they are rows, externally Java objects, have underlying schema which we can manipulate. Typesafe + Fast

RDDs
Resilient – Achieve fault tolerance through lineages
Distributed Datasets – collection of datasets distributed across multiple machines
→ Why lineages? Improves reliability, because spark mostly stores intermediate data in memory, which is not persistent. If results are lost due to machine failure, we can use the DAGs/lineages to recover the intermediate results
→ For spark, we try to aggregate main memory of multiple machines to achieve **memory pool**

RDDs Types and Operations
2 Types of RDDs: Mainly based on where they come from.
1. Parallelized collections – An existing collection from a single node, and we parallelize that
2. Hadoop datasets – files from HDFS or other compatible storage

Transformations f(RDD) => RDD
- Lazy (not computed immediately), e.g. map
→ All transformations in spark are lazy. The transformations applied to some base dataset is remembered, and the transformations are only computed when an action requires a result. → Spark runs more efficiently → We can just return the final result to the driver, don't need to save intermediate results anywhere on disk or return to driver.

Actions: Triggers computation. E.g. "count", "saveAsTextFile"

How to distribute RDDs over partitions

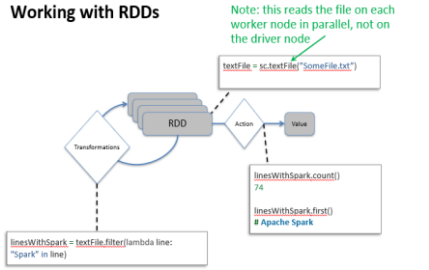
→ RDDs are immutable, cannot be changed once created.
→ The above function parallelizes and partitions the rdd into 3 workers

Transformations
→ Turn RDDs into RDDs using functions such as map, filter

Actions – Trigger spark to compute some result from a series of transformations (e.g. .collect(), an action that asks spark to retrieve all elements of the RDD to the driver node (which then can be used to perform in memory operations like throwing into a java arraylist (which also can directly be done using collectAsList()))

Distributed Processing
- Transformations/actions happen in parallel, results are only sent to driver in the final step

Spark example



Caching in Spark
- Do rdd.cache() to cache rdds in spark. This will create and keep the rdd in the machine's memory, which can be used by other operations later
- cache(): saves an RDD to memory of each worker node.
- persist(options): can be used to save an RDD to memory disk or off-heap memory

When to cache/not cache an RDD?
- When it is expensive to compute and needs to be re-used multiple times
- If worker nodes do not have enough memory, it will evict the "least recently used" RDDs. Be aware before caching!

Directed Acyclic Graph (DAG)
- Spark creates a DAG graph internally which represents all the RDD objects and how they will be transformed.
- **Transformations construct this graph and actions trigger computations on it!**

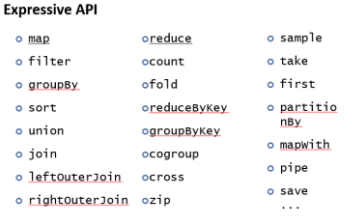
Lineage and Fault Tolerance
Unlike Hadoop, spark does not use replication to allow fault tolerance. Why? Remember that spark tries to store all data in memory, not disk. Memory capacity is much more limited, so duplicating the data is very expensive!

- Lineage approach: If a worker node goes down, we replace it by a new worker node and use the DAG graph to recompute the data in the lost partition. **We only need to recompute the RDDs from the lost portion.**

- DAG: Keep tract of which worker node depends on the computation of the previous worker node, so if the current one goes down, we can simply recompute on whoever depended on it

Narrow and Wide Dependencies
- Narrow Dependencies – operations which only require data from 1 partition e.g. map, filter, contains
- Wide Dependencies – Operations that require multiple partitions e.g. groupBy, orderBy
- **In the DAG, consecutive narrow dependencies are grouped together as "stages".**
- Data only needs to be shuffled (exchange of data, incurring of disk and network i/o) when going between stages!
- **Within stages, Spark tries to perform consecutive transformations on the same machine.**
- Minimizing shuffle is good for performance!

Spark API



DataFrames and DataSets
- A DataFrame represents a table of data, similar to tables in SQL or DataFrames in pandas.
- Compared to RDDs, DFs are a higher level interface e.g. it has transformations that resemble SQL operations.
→ Recommended interface for working with Spark – they are easier to use than RDDs and almost all tasks can be done with them, while only rarely using RDD functions.
→ However, DF operations are compiled down to RDD operations under the hood!

```
flightData2015 = spark\
.read\
.option("inferSchema", "true")\
.option("header", "true")\
.csv("/mnt/defg/flight-data/csv/2015-summary.csv")
```

- Reads in a DataFrame from a CSV file.

```
flightData2015.sort("count").take(3)
```

- Sorts by 'count' and output the first 3 rows (action)

```
Array([United States,Romania,15], [United States,Croatia,...
```

DataFrames: Transformations
- One way is to use SQL queries to transform dataframes. Takes in a DataFrame and returns a DataFrame.

- An easy way to transform DataFrames is to use SQL queries. This takes in a DataFrame and returns a DataFrame (the output of the query).

```
flightData2015.createOrReplaceTempView("flight_data_2015")
maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")
maxSql.collect()
```

- Aside from using SQL, can also use DataFrame interface.
- Take in strings (of the column names), which represents columns, return a column object.

- We can also run the exact same query as follows:

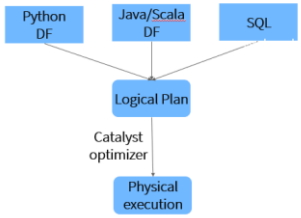
```
from pyspark.sql.functions import desc
flightData2015\
.groupBy("DEST_COUNTRY_NAME")\
.sum("count")\
.withColumnRenamed("sum(count)", "destination_total")\
.sort(desc("destination_total"))\
.limit(5)\
.collect()
```

DataSets
- Similar to DFs but are type-safe. Not available in python/R, which don't support strict typing
- In Java they type signature is Dataset<Flight> for example, so when you do operations such as collect(), it also return objects of Flight class, instead of Row like DataFrames.
- Datasets/DataFrames use much less memory in caching compared to RDDs! Why? Because of the defined schema in DF/DS, compression can happen more efficiently? I think

Spark DataFrame Execution

- Layer between logical plan and physical execution. We have wrappers to create logical plan, and similar to DBMS query optimizer, we optimize physical plan by going through relational tree. (e.g. type of join used)

Spark DataFrame Execution



Performance Guidelines for Basic Algorithmic Design

- Linear Scalability: When you increase number of nodes, you want the performance of your algorithm to scale linearly (computer resources). Performance should also be linear on data size

- Minimize amount of I/Os in hard disk and network
→ Minimize disk I/O. Minimize random I/Os vs sequential I/Os
→ Minimize network I/O; bulk send/receives vs many small send/receives
- Memory working set of each task/worker
→Large memory working set → higher probability of failures

Applicable guidelines to both spark and MR!!

Possible Challenges of computation over big data? Watch out for:
1. Correctness
2. Performance
3. Trade-off between accuracy and performance (??)

Examples with Spark

1. Big Data Variance

Recall formula:

$$Var(X) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$

Second pass

First pass

One pass to calculate mean, then another pass to calculate variance

(Calculate mean by doing a reduce and summing up all xi)

Another possible fast (but inaccurate) solution:

$$Var(X) = E[X^2] - E[X]^2$$

$$= \frac{\sum x^2}{N} - \left(\frac{\sum x}{N}\right)^2$$

☑ Can be performed in a single pass, but

☐ Subtracts two very close and large numbers!

→ if sum(x^2) and sum(x) are stored as integers, given input with huge numbers, we will have overflow problem.
→ Another problem: Because of rounding error, when numbers are added in different order, result can be different despite sum being a commutative operation!

Another Solution: Accumulator Pattern

- An object that incrementally tracks the variance

```
Class RunningVar {
  var variance: Double = 0.0

  // Compute initial variance for numbers
  def this(numbers: Iterator[Double]) {
    numbers.foreach(this.add(_))
  }

  // Update variance for a single value
  def add(value: Double) {
    ...
  }
}
```

Welford's online algorithm, https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's_online_algorithm

→ Welford's online algorithm to incrementally track variance. Just be aware of this pattern

Parallelize for performance

- Distribute adding values in map phase, merge partial results in reduce phase. (Hadoop?)
→ Spark has this idea implemented!

Use the RunningVar in Spark

```
doubleRDD
  .mapPartitions(v => Iterator(new RunningVar(v)))
  .reduce((a, b) => a.merge(b))
```

Or simply use the Spark API

```
doubleRDD.variance()
```

Approximate Estimations

→ Might be good enough if it can be computed faster or cheaper. Trade accuracy with memory or running time.
→ Especially if the case has **an error bound,** then it can be especially useful!

Cardinality Problem

- E.g. Count number of unique words in Shakespeare's work
→ Use of Hashset – requires too much memory!

Linear Probabilistic Counting

1. Allocate bitmap of size m and initialize to zero
- Hash the word to a position in the bitmap, set corresponding bit to 1

Linear Probabilistic Counting (cont')

2. Count number of empty bit entries: v

$$count \approx -m \ln \frac{v}{m}$$

→ The bigger m is, the more accurate the estimation.
→ Requires one pass of the document and one pass of the bitmap to get the answer.

Spark's Implementation: The Spark API

Use the LogLinearCounter in Spark

```
rdd
 .mapPartitions(v => Iterator(new LPCounter(v)))
 .reduce((a, b) => a.merge(b)).getCardinality
```

Or simply use the Spark API

```
myRDD.countApproxDistinct(0.01)
```

rsd - maximum estimation error allowed (default = 0.05)

More Spark APIs:

More Details...

Spark APIs:
- **approxCountDistinct:** returns an estimate of the number of distinct elements
- **approxQuantile:** returns approximate percentiles of numerical data

Google PageRank Algorithm

→ Each page starts with rank 1/N
→ On each iteration:
A. contrib = currRank / |neighbors|
B. currRank = 0.15/N + 0.85sum(contrib of my neighbors)
Algo converges when nodes are all stable. 0.15 and 0.85 are arbitrary numbers.

(not important, prof skipped this)
- Rank of each page is the probability of landing on that page for a random surfer on the web.
- Probability of visiting all pages after k steps $V_k = A^k * V^t$, where V is the initial rank vector and A is the link structure (sparse matrix)

Data Representation in Spark

Each page is identified by its unique URL rather than an index.
- Ranks vectors (V): RDD[(URL, Double)]
- Links matrix (A): RDD[(URL, List(URL))]

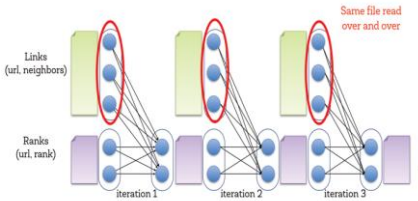
Spark Implementation

```
val links = // load RDD of (url, neighbors) pairs
var ranks = // load RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks).flatMap {
    case (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  ranks = contribs.reduceByKey(_ + _)
    .mapValues(0.15 + 0.85 * _)
}
ranks.saveAsTextFile(...)
```

→ You join links with ranks, now for each tuple, you get the url, with its rank and associated neighbors. Then, for each of these tuples, you create an array of tuples, where each tuple is (neighbor of curr tuple, my contribution). My contribution is basically my rank / |neighbors|.
→ Then, we can reduce by key, which collects all the tuples with the same url together. This consolidates all the contribution received by the current URL's neighbors. We sum these contributions up and apply 0.15/N + 0.85(sum(contrib)) to get new PageRank.

Repeatedly multiply sparse matrix and vector



→ Observe that when we are doing joins, we read the same links file over and over to join with the newly updated ranks. W
→ Cache the links in memory! Also, we don't write intermediate results on disk. Caching prevents us from partitioning the neighbors over and over.

```
val links = // load RDD of (url, neighbors) pairs
var ranks = // load RDD of (url, rank) pairs

links.partitionBy(hashFunction).cache()

for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks).flatMap {
    case (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  ranks = contribs.reduceByKey(_ + _)
    .mapValues(0.15 + 0.85 * _)
}
ranks.saveAsTextFile(...)
```

ACID vs BASE

- BASE = Basically Available, Soft State, Eventually consistent
- ACID = Atomicity, Consistency, Isolation and Durability

The CAP theorem, also known as Brewer's theorem, states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees: Consistency. Availability. Partition tolerance.

→ Basically Available: System guarantees the availability of data with respect to CAP theorem and there will be a response to any request. However, the requested data may not be in a consistent state, and may not be an updated value

→ Soft State: State of system could change over time, even during state when there is no input or requests, there may be changes going on to reach eventual consistency (therefore soft state).

→ Eventual consistency: System will eventually become consistent once it stops receiving input. Data will propagate to everywhere it should sooner or later. As system continues to receive input, it does not ensure and check for consistency after every transaction before it moves on to the next one.

Graphs

Defacto data structure for many applications.
→ Graph Data: Social Networks, media networks, traffic network (roads/maps).
→ Web as a graph (nodes: webpages, edges: hyperlinks)
Why Web Graph? Large scale, well studied yet still emerging area.

Organizing the Web

First Try: Human curated web directories (yahoo, dmoz). Second Try: Web search: find relevant docs in a small and trusted set (newspaper articles, patents). Take note the web is huge and full of untrusted documents

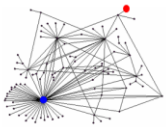
Challenges of web search:

→ Web contains many sources of information (who to trust? – trustworthy pages might point to each other!)
→ What is the best answer to query something like "newspaper"? There may be no single right answer, but pages that actually know newspapers might all be pointing to many newspapers
→ In essence, we are solving issues relating to retrieving trustworthy sources and retrieving topic related answers.

Ranking Nodes on the Graph

All web pages are not equally "important"
www.joe-schmoe.com vs. www.stanford.edu

There is large diversity in the web-graph node connectivity.
Let's rank the pages by the link structure!

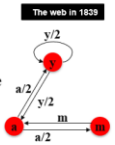


Google PageRank
- Page is more important if it has more in coming links. Links from important pages count more
- Each link's vote is proportional to the importance of its source page. If page j with importance rj has n out-links, each link gets rj/n of its importance.
- Page j's own importance is the sum of the votes on its in-links.

PageRank "Flow" Model
- A "vote" from an important page is worth more
- A page is important if it is pointed to by other important pages
- Define a "rank" r_j for page j
- Can be viewed as a random walk on the graph.

$$r_j = \sum_{i \to j} \frac{r_i}{d_i}$$

d_i ... out-degree of node i



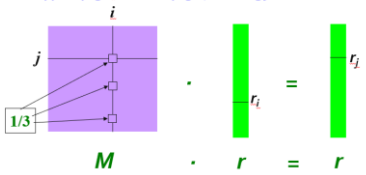
→ We have 3 equations and 3 unknowns. We need additional constraints to enforce a unique solution.
- 3 equations, 3 unknowns, no constants
  - No unique solution
  - All solutions equivalent modulo the scale factor
- Additional constraint forces uniqueness:
  - $r_y + r_a + r_m = 1$
  - Solution: $r_y = \frac{2}{5}$, $r_a = \frac{2}{5}$, $r_m = \frac{1}{5}$
- Gaussian elimination method works for small examples, but we need a better method for large web-size graphs
- We need a new formulation!

PageRank: Matrix Formulation
- Stochastic adjacency matrix M
  - Let page i has d_i out-links
  - If $i \to j$, then $M_{ji} = \frac{1}{d_i}$ else $M_{ji} = 0$
    - M is a column stochastic matrix
    - Columns sum to 1
- Rank vector r: vector with an entry per page
  - r_i is the importance score of page i
  - $\sum_i r_i = 1$
- The flow equations can be written

$$r = M \cdot r \qquad r_j = \sum_{i \to j} \frac{r_i}{d_i}$$

→ Each row represents the rankings flowing into that node j, each column represents the rankings flowing out of the node I (which is why the columns sum to 1)
→ Represent the links in term of this matrix, then calculate the respective values based on the teleport probability and the in coming links.
→ Get the simultaneous equations and solve them to get the final ranks

- Remember the flow equation: $r_j = \sum_{i \to j} \frac{r_i}{d_i}$
- Flow equation in the matrix form $M \cdot r = r$
  - Suppose page j links to 3 pages, including j



$M \cdot r = r$

→ This slide is basically saying that M.r = r basically is the notation on the top right

Power Iteration:

Power Iteration Method
- Given a web graph with n nodes, where the nodes are pages and edges are hyperlinks
- Power iteration: a simple iterative scheme
  - Suppose there are N web pages
  - Initialize: $\mathbf{r}^{(0)} = [1/N,\ldots,1/N]^T$
  - Iterate: $\mathbf{r}^{(t+1)} = \mathbf{M} \cdot \mathbf{r}^{(t)}$
  - Stop when $|\mathbf{r}^{(t+1)} - \mathbf{r}^{(t)}|_1 < \varepsilon$

$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{d_i}$$

d_i .... out-degree of node i

$|x|_1 = \sum_{1 \le i \le N} |x_i|$ is the L_1 norm
Can use any other vector norm, e.g., Euclidean

→ Just use matrix and create simultaneous equations and solve from there, think this slide is just mainly saying power iteration can be done by iterating the middle 2 bullet points until we reach some improvement less than some value epsilon, and we can stop.

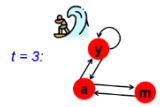
To solve PageRank,
- Power Iteration:
  - Set $r_j = 1/N$
  - 1: $r'_j = \sum_{i \to j} \frac{r_i}{d_i}$
  - 2: $r = r'$
  - Goto 1
- Example:

$r_y = r_y/2 + r_a/2$
$r_a = r_y/2 + r_m$
$r_m = r_a/2$

$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{matrix} 1/3 & 1/3 & 5/12 & 9/24 & 6/15 \\ 1/3 & 3/6 & 1/3 & 11/24 & \ldots\ 6/15 \\ 1/3 & 1/6 & 3/12 & 1/6 & 3/15 \end{matrix}$$

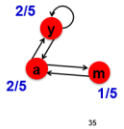
Iteration 0, 1, 2, ...

Random Walk Interpretation
→ Imagine a random web surfer, at time t it could be at any page I, and at time t+1, it follows an outlink from i uniformly at random. This process repeats indefinitely.
e.g. for this diagram, at next step, surfer could either be at y or a (because of the outgoing links)



→ Interpreting page rank scores as a function of time:
- Let:
  - $\boldsymbol{p}(t)$ ... vector whose ith coordinate is the prob. that the surfer is at page i at time t
  - So, $\boldsymbol{p}(t)$ is a probability distribution over pages
- Stationary Distribution: as $t \to \infty$, the probability distribution approaches a 'steady state' representing the long term probability that the random walker is at each node, which are the PageRank scores

$p(t)$ where $t \to \infty$:



35

→ As t goes towards infinity, since we keep calculating the updated page ranks, when they converge (impt: they don't always converge!), where we have a long term probability.

Equivalence between Random Walk and Flow formulations
- For random walk, at time t+1, p(t+1) = M.p(t)
- When the random walk reaches convergence/stationary state Ps, then Ps = M.Ps
- In flow formulation, the rank vector r is defined by r = M.r

→ Same recursive definition

PageRank with Teleports
- PageRank does not always converge

Does this converge?

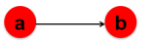
$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{d_i}$$

- Example:

$$\begin{bmatrix} r_a \\ r_b \end{bmatrix} = \begin{matrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{matrix}$$

Iteration 0, 1, 2, ...

Another example:



$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{d_i}$$

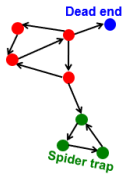
- Example:

$$\begin{bmatrix} r_a \\ r_b \end{bmatrix} = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

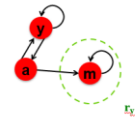
Iteration 0, 1, 2, ...

Problems with vanilla PageRank:
Deadends:
1. Some pages are dead-ends (have no out-links)
→ Random walk has nowhere to go to
→ **Such pages cause importance to leak out**

Spider traps:
All out links are within a group
→ Random walkers get stuck in a trap, and eventually, **the spider traps absorb all the importance.**



Example: Spider Traps
- Power Iteration:
  - Set $r_j = 1$
  - $r_j = \sum_{i \to j} \frac{r_i}{d_i}$
    - And iterate



- Example:

$r_y = r_y/2 + r_a/2$
$r_a = r_y/2$
$r_m = r_a/2 + r_m$

$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{matrix} 1/3 & 2/6 & 3/12 & 5/24 & 0 \\ 1/3 & 1/6 & 2/12 & 3/24 & \ldots\ 0 \\ 1/3 & 3/6 & 7/12 & 16/24 & 1 \end{matrix}$$

Iteration 0, 1, 2, ...

All the PageRank score gets "trapped" in node m.

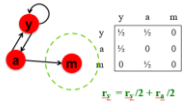
44

→ Spidertrap eventually absorb all the importance

Solution: PageRank with Teleport
Spider traps:
→ At each step, the random surfer can either follow a link with random probability β, or with probability 1-β, jump to a random page.
→ Common values for β are in the range of 0.8-0.9
→ Surfer can teleport of out a spider trap eventually!

Dead Ends:
In dead ends, the matrix is not stochastic (the sum of the values in the column for the node without any outgoing links is 0, not 1).

Problem: Dead Ends
- Power Iteration:
  - Set $r_j = 1$
  - $r_j = \sum_{i \to j} \frac{r_i}{d_i}$
    - And iterate



- Example:

$r_y = r_y/2 + r_a/2$
$r_a = r_y/2$
$r_m = r_a/2$

$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{matrix} 1/3 & 2/6 & 3/12 & 5/24 & 0 \\ 1/3 & 1/6 & 2/12 & 3/24 & \ldots\ 0 \\ 1/3 & 1/6 & 1/12 & 2/24 & 0 \end{matrix}$$

Iteration 0, 1, 2, ...

Here the PageRank "leaks" out since the matrix is not stochastic.

51

→ Solution: **Always teleport.** Follow random teleport links with probability 1.0 from dead-ends!

Why do teleports solve the problem?
→ Spidertraps are okay, but traps pagerank scores which are not what we want. The nodes within a spider trap still reinforce each other, but we give a chance for the surfer to teleport out so that the traps do not absorb all the pagerank scores.
→ However, dead-ends are a problem because it leads to the matrix being not column stochastic, going against our initial assumptions. So, we need to make it column stochastic again so that the PageRank algorithm works.

→ Preprocess the matrix M is remove all dead ends.

PageRank Equation:

Solution: Random Teleports
- Google's solution that does it all:
  At each step, random surfer has two options:
  - With probability β, follow a link at random
  - With probability $1-\beta$, jump to some random page
- PageRank equation [Brin-Page, 98]

$$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1-\beta)\frac{1}{N}$$

d_i ... out-degree of node i

This formulation assumes that M has no dead ends. We can either preprocess matrix M to remove all dead ends or explicitly follow random teleport links with probability 1.0 from dead-ends.

→ Equation: the rank of a page j (rj) is the sum of the ri/di (rank of all neighbors i divided equally among all out degrees) * B (chance of not teleporting) + (1-B) (chance of teleporting) * 1/N (random distribution of every page)
- The Google Matrix A:

$$A = \beta M + (1-\beta)\left[\frac{1}{N}\right]_{N \times N}$$

[1/N]NxN...N by N matrix where all entries are 1/N

- We have a recursive problem: $r = A \cdot r$
  And the Power method still works!

Problems with PageRank:
1. Measures generic popularity of a page, but is biased against topic-specific authorities (solution? Topic-specific pagerank)
2. Uses a single measure of importance, but they might be other factors of importance. (Solution? Hubs-and-authorities)
3. Susceptible to link-spam, where artificial link topographies are created to boost page rank. (Solution? TrustRank)

Topic Sensitive/Specific PageRank
→ Measure popularity within a topic. Evaluate web page not just according to their popularity but how close they are to a particular topic like sports and history

→ Allow search queries to be answered based on interests of user. ("Apple" could be fruit or company)

Algorithm:
- Random walker has a small probability of teleporting at any step. Teleport can go to:
→ Any page with equal probability to avoid dead-ends/spider-traps (Vanilla pagerank), or **a topic set of relevant pages** (teleport set)
→ Idea: When a random walker teleports, it picks a page from a set S, where it contains only pages relevant to the topic. For each teleport set S, we get a different vector rs (rank vector).

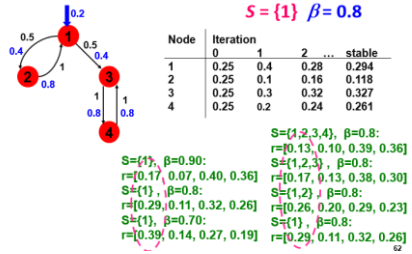
- To make this work all we need is to update the teleportation part of the PageRank formulation:

$$A_{ij} = \begin{cases} \beta M_{ij} + (1-\beta)/|S| & \text{if } i \in S \\ \beta M_{ij} + 0 & \text{otherwise} \end{cases}$$

  - A is stochastic!
- We weighted all pages in the teleport set S equally
  - Could also assign different weights to pages!
- Compute as for regular PageRank:
  - Multiply by M, then add a vector
  - Maintains sparseness

→ (1-B) – probability we teleport, multiply by 1/|S| - |S| is the number of pages(vertices) in the graph
→ Instead of normal uniform page rank where we teleport to any page, but now we only teleport to the relevant topics
→ If some pages are even more relevant than some other pages in the same topic, assign more weight

Example: Topic-Specific PageRank

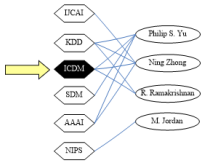


- As beta decreases, the probability of jumping to S becomes larger. Thus, the rank of vertex 1 increases.
- As S set decreases, more weight is given to vertex 1. S is the set to which we can teleport to, if S increases, the probability to each node is (1-B)/|S|
→ Conclusion: We can bias page rank to some specific page in the topic by tweaking value B or by tweaking the teleport set.

Discovering the Topic Vector S
→ Create pageranks for different topics. How to decide which topic ranking to use?
- Users pick from menu
- Classify query into a topic
- Use context of query (query launched from webpage talking about a known topic, use user's search history to figure context of query, user's bookmarks etc)

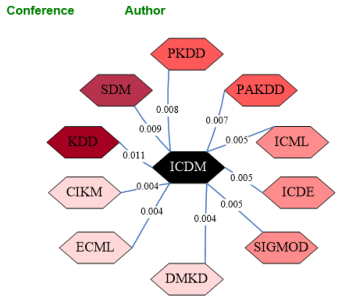
Measuring Proximity in Graphs
- We want to measure similarity between 2 topics.
- Using metrics such as shortest path, network flow is not good as it doesn't reflect the graph structure well.
- A good notion of proximity will be one with multiple connections

SimRank: Idea
→ Perform random walk from a fixed node on a k-partite graph. (There are k types of nodes, e.g. Authors, Conferences, Tags etc).
→ Effectively, SimRank is a measure that says "two objects are considered to be similar if they are referenced by similar objects."
→ To calculate similarity score to node u, we make teleport set = {u} (only contain node u), which allows us the measure the similarity to node u. (maybe this works because we can then see what are the respective page ranks of each comparison, the one that matches more closely will have higher values along the path of node u?)
→ **Problem:** Must be done once for each node u, which is ok on small graphs and bad on large ones



Q: What is most related conference to ICDM?

A: Topic-Specific PageRank with teleport set S={ICDM}



→ Darker the color the more similar.

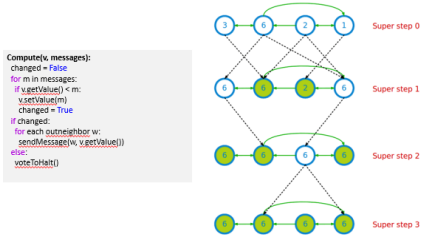
PageRank Summary
→ Vanilla/Normal PageRank: teleports uniformly at random to any node, all nodes have the same probability of surfer landing there.
→ Topic-Specific PageRank: teleports to any page in the set of pages in the topic. May not be uniform, some pages that are more relevant / impt have higher chances of landing there
→ SimRank (aka Random walk with restarts): Always teleport to same node.

Graph Systems
Pregel – Computation consists of a series of supersteps. In each superstep, the framework invokes a user-defined function, compute(), for each vertex, in parallel.
- compute() specifies the behaviour at a single vertex v and a superstep s:
  - `compute()` specifies behavior at a single vertex *v* and a superstep *s*:
    - It can read messages sent to *v* in superstep *s* -1;
    - It can send messages to other vertices that will be read in superstep *s* + 1;
    - It can read or write the value of *v* and the value of its outgoing edges (or even add or remove edges)

Termination:
- A vertex can choose to deactivate itself
- Is woken up if new messages are received.
- Computation will halt when all vertices are inactive.

→ Look at the 3 highlighted bullets, its actually quite similar to page rank, in page rank remember for each iteration and each node has some equation that is determined by the in degree from neighboring nodes + teleport, and the associated values simply depends on the previous iteration. So if I can read messages from my previous superstep, I can compute my current rank, or if I can send messages to vertices in next superstep just means it will help to compute for next iteration.

Pregel Example: Find max value

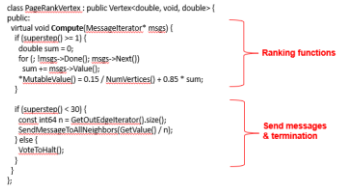


Pregel: Implementation
- Master & slave architecture
→ Vertices are hash partitioned by default and assigned to workers ("edge cut"). Note: Size of partition must be smaller than machine's memory.

Pregel Performance
→ **Each worker maintains the state of its portion of the graph in memory**
→ **Computations happen in memory**
→ Messages from vertices can be sent to same worker or different workers. (buffered locally or sent as a batch to reduce network traffic).
→ **Very important for the partitions to be in memory.** Memory has better performance than disk. On top of that, for graphs, a node can arbitrarily have anyone else as its neighbor. At the storage level, these neighbors are stored in "random" locations and not sequential → disk i/o will all be random i/os which are very expensive.

Pregel Fault Tolerance
- Checkpoints to persistent storage. Failure is detected through heartbeats (which check if machine is alive). Once detected, corrupt workers are reassigned and reloaded from the checkpoints.

Pregel: PageRank



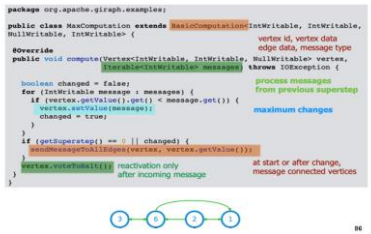
Apache Giraph
- Open source counterpart to pregel, with more features

Giraph Architecture
- Master which synchronizes the super steps and assigns partitions to workers before super step begins.
- Workers that handles computation and messaging. Handles I/O (reading/writing of graph), computation/messaging of assigned partitions.

ZooKeeper – Maintains global app state. ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications.

Giraph Workflow
1. Loading the graph into main memory
2. Compute on the in-memory graph partitions. A worker can work on multiple partitions and perform local reduction on the messages. Explicit iterations are supported for efficiency.
3. Store the graph into distributed file systems, for example, HDFS.

Example (Giraph): Max Value



→ Similar to pregel

GraphX – Spark for graphs
- Spark is more efficient in iterative computation than Hadoop.
- Integration of record-oriented and graph-oriented processing
- Extends RDDs to resilient distributed property graphs.

Property Graphs can present different views of the graph, support map like operations and support distributed pregel-like aggregations.

Summary for Graphs
- Graphs are de facto data structures for many applications.
- PageRank is a fundamental algorithm for Web graph.
- Other than mapreduce, many distributed systems have been built for large scale graph processing.

Streams
- Sequence of items; structured (tuples) or unstructured(documents)
- ordered (implicitly or timestamped)
- Is arriving continuously at high volume
- May not be possible to store entirely due to high volume
- Not possible to examine all items as it will be too slow
→ Challenges are volume and velocity

Applications of streams
- Network traffic monitoring
- Datacenter monitoring
- Sensor networks monitoring
- Credit card fraud detection
- Stock market analysis
- Online mining of click streams
- Monitoring social media streams

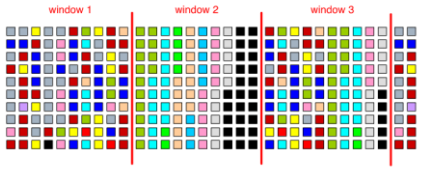
Scale of big data streams
- Single 2 Gb/sec link; say avg. packet size is 50 bytes
  - Number of packets/sec = 5 million
  - Time per packet = 0.2 microseconds
- If we only capture header information per packet: source/destination IP, time, no. of bytes, etc. – at least 10 bytes
  - 50 MB per second
  - 4+ TB per day
  - **Per link!**

→ With such scale, it might be difficult to perform deep-packet inspection because no time

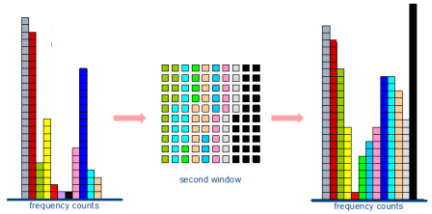
What makes big data problems hard?
→ Intrinsic challenges: Volume, velocity, limited memory/storage, strict latency requirements, out of order delivery

→ System challenges: Load balancing, unreliable message delivery, fault-tolerance, consistency semantics (lossy, exactly once etc)

Common Techniques for Streams
→ We start with a simple hello world stream processing program, counting the frequency of items in a stream.
→ Handle streams by non-overlapping windows.



→ Build histogram based on window, compiling its relative counts



Window Counting
→ If stream volume is too high/velocity is too high, we can approximate (solution is approximate/lossy) instead of getting a exact value, which might require too much computation to achieve.
→ General strategies: Sampling or hashing

Reservoir Sampling
→ Select s elements from a stream of size N with uniform probability. N can be infinite!
→ Store first s elements
→ For the kth element after, keep it with probability s/k (and randomly discard an existing item).
→ e.g. if s = 10, keep the first 10 elements, 11^{th} element keep with prob 10/11, 12^{th} element with probability 10/12
→ Reservoir sampling ensures that every element so far has an equal chance of staying in the pool.
→ Proof: We prove by induction, assume our hypothesis is true that the probability of selecting each item up till now is s/k.
→ When the new item comes in, it has a s/k+1 probability of getting selected. If so, the probability that an element is discarded from the pool is 1/s * s/k+1 = 1/k+1. Which means that the probability that an element is not discarded from the pool is k/k+1.
→ From our inductive hypothesis, the overall probability of some element being in the sample is s/k * k/k+1 = s/k+1 (Proven!)

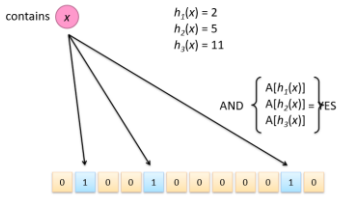
Stream Hashing
→ Common tasks: cardinality estimation, set membership, frequency estimation.

Flajolet-Martin FM counter
→ For cardinality estimation
→ Pick a hash function h that maps each of the N elements to at least $\log_2 N$ bits (So each element is assigned one unique number like 0,1,2,..)
→ Let r(a) be the number of trailing 0s in h(a). r(a) = position of first 1 counting from the right - E.g. If h(a) = 12, 12 is 1100 in binary, so r(a) = 2
→ Store R = the maximum r(a) seen, i.e. R = max(r(a)) over all items a seen so far.
→ Estimated number of distinct elements = 2^R

Intuition to FM Counter
→ h(a) hashes a with equal probability to any of the N values. So, h(a) is a sequence of $\log_2 N$ bits.
→ About 50% of all values hash to ***0, 25% hash to **00…
So, if we saw the longest tail of r = 2, then we have probably seen (expected number of items seen) 1/0.25 = 4 items so far!
→ I assume this only works if the h(a) increments sequentially? Not sure about this though

Bloom Filters
→ Find out if some element is a member of some set
Bloom Filter APIs
put(x) → insert x into the set, contains(x) → yes if x is a member of the set
→ m-bit vector, k hash functions: h1,…,hk
→ When you put, you hash your element with all k hashes, and add 1 to each position in the bit vector.
→ When you check contains(x), you hash your element with all k hashes and check the positions of all relevant slots in the bit vector. If all of the slots store a 1, then the contains method replies true.
→ delete(x) is not supported!!

Bloom Filters: contains

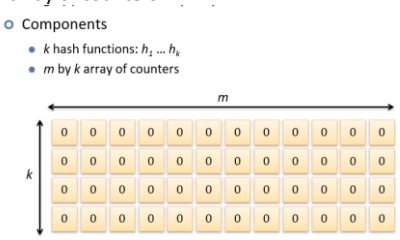


→ With this method, false positives are possible, but false negatives are impossible.
→ We need a bit vector that is large enough such that when enough elements are added, the bit vector wont just be filled with 1s! Else contains(any element) will likely return true.

When do we use Bloom Filters?
→ When we have capacity constraints and allow some error probability/boundary
→ We can tune the size of the bit vector m or the number of hash functions k. Increasing size of bit vector m and increasing number of hash functions should reduce the number of false positives.

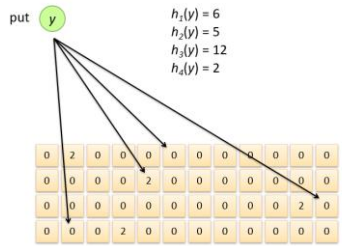
Challenges of using Bloom Filters
→ When doing in parallel, multiple hashes might write to same bit location. We need a lock to ensure atomicity/consistency of operation/bitmap.

Count-Min Sketch (CMS)
Task: Frequency Estimation
→ put(x) → Increment count of x by 1
→ get(x) → Return frequency of x
→ We have k hash functions: h1…hk and an m by k array of counters.



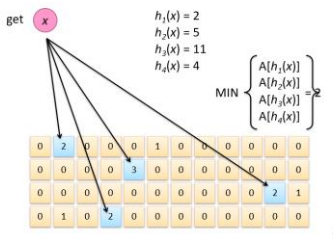
How CMS works
→ When we do put(x), we calculate the hashes of x against all k hashes. Based on which hash we used and the bucket x fall into, we add 1 to the current count accordingly

Count-Min Sketches: put



→ When we get, we once again will calculate the values of all k hashes.
→ Check the value in each of the buckets. The estimate of the frequency is the minimum of all values of every bucket checked.

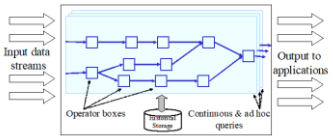
Count-Min Sketches: get



CMS Evaluation
→ It reasonably estimates the heavy-hitters (elements that appear a lot), over-estimation at the tail (Elements that don't appear a lot)
→ **When to use?** To find number of distinct events (???), find distribution of events and when we have high error bound.
→ We can tune the number of buckets m, the number of hash functions k

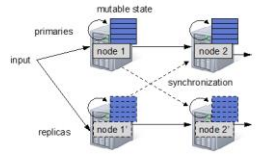
Other stream processing frameworks - Storm
→ Typical architecture: Data flow system, data flows in and goes through series of transformations in a DAG fashion and out put to applications.

Typical Architecture: Aurora



Spark Stream

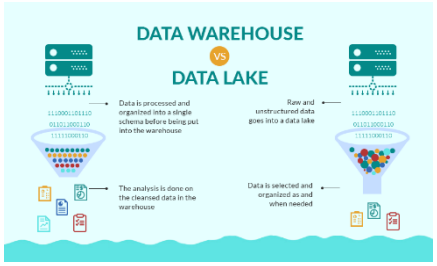
Typical Distributed Architecture: SparkStreaming



→ Input is a small window of the stream, coming in as an RDD
→ Buffer a small window then pass to spark. If we do not buffer, then the parallelism is quite bad when we take 1 or 2 tuples and create an rdd out of it. If we wait too long, then it will affect latency (too long). Adequate amount can support parallelism (fast computation) and acceptable latency

Data Lake
→ System or repository of data stored in its natural/raw format, usually object blobs or files (blob stores like S3/google cloud storage)
→ Data representation – raw copies of source system data, sensor data etc.
→ Can also be transformed data use for tasks such as reporting, visualization, analytics and machine learning

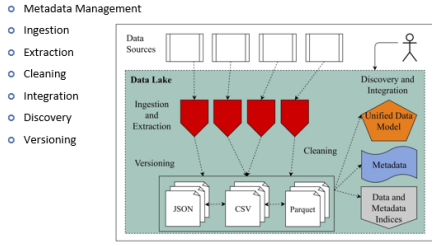
Data Warehouse vs Data Lake



→ Lake stores everything, warehouse stores business critical data
→ Unprocessed vs highly processed
→ Unstructured/semi-structured/structured vs tabular and structured
→ Lake allows for anyone to manage the data that they need. Warehouse is optimized for data retrieval
→ Lake is highly agile, configurable, warehouse fixed and less agile.
→ Lake can ingest/process fast, is cheap whereas warehouse takes time to introduce new content and requires more expensive storage.

Common tasks in Data Lake

Common Tasks in Data Lakes

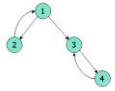


Challenges in Data Lake
→ Data mistakes in Parsing: "Event" vs "event", might be interpreted as different tokens/words
→ Tedious to book keep on what is correct (e.g. late/out of ordered data are hard to catch)
→ Might handle a lot of small files, and performance can be a concern

Calculating Topic Specific PageRank

Problem 3

Consider the following link topology.

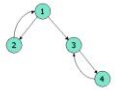


Compute the Topic-Specific PageRank for the following link topology. Assume that pages selected for the teleport set are nodes 1 and 2 and that in the teleport set, the weight assigned for node 1 is twice that of node 2. Assume further that the teleport probability, (1 - beta), is 0.3.

Solution 3

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ \frac{1}{2} & 0 & 1 & 0 \end{bmatrix} \quad N_e = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$N = \begin{bmatrix} \frac{2}{3} & \frac{2}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$



Solution 3

$$A = \beta M + (1-\beta)N = \begin{bmatrix} 0.2 & 0.9 & 0.2 & 0.2 \\ 0.45 & 0.1 & 0.1 & 0.1 \\ 0.35 & 0 & 0 & 0.7 \\ 0 & 0 & 0.7 & 0 \end{bmatrix}$$

$$\begin{bmatrix} tr_1 \\ tr_2 \\ tr_3 \\ tr_4 \end{bmatrix} = A \cdot \begin{bmatrix} tr_1 \\ tr_2 \\ tr_3 \\ tr_4 \end{bmatrix}, tr_1 + tr_2 + tr_3 + tr_4 = 1$$

$$tr_1 = 0.3576, tr_2 = 0.2252, tr_3 = 0.2454, tr_4 = 0.1718$$

Standard PageRank

Problem 1

Consider three Web pages with the following links:

Suppose we compute PageRank with a β of 0.7, and we introduce the additional constraint that the sum of the PageRanks of the three pages must be 3, to handle the problem that otherwise any multiple of a solution will also be a solution. Compute the PageRanks a, b, and c of the three pages A, B, and C, respectively.

Solution 1

$$M = \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 1 & 1 \end{bmatrix} \end{array} \quad N = \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix} \end{array}$$

$$A = \beta M + (1-\beta)N = \begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.45 & 0.1 & 0.1 \\ 0.45 & 0.8 & 0.8 \end{bmatrix}$$

Solution 1

$$\begin{bmatrix} r_a \\ r_b \\ r_c \end{bmatrix} = A \cdot \begin{bmatrix} r_a \\ r_b \\ r_c \end{bmatrix}$$

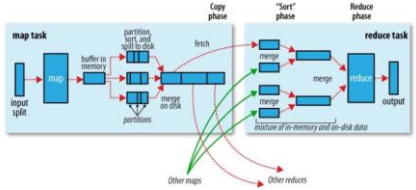
$$r_a = 0.1r_a + 0.1r_b + 0.1r_c$$
$$r_b = 0.45r_a + 0.1r_b + 0.1r_c$$
$$r_c = 0.45r_a + 0.8r_b + 0.8r_c$$

$$r_a + r_b + r_c = 3$$

$$r_a = 0.3, r_b = 0.405, r_c = 2.295$$

MapReduce Shuffle and Sort

Shuffle: To transfer map output from mapper to a reducer in MapReduce.
Sort: Cover merging and sorting of map outputs.
→ Shuffle will take the values with same key and bring them to the reducers.
→ Once they are brought over, the sorting of the keys are done at the reducers. **Note: Values are not sorted and can be in any order.**
→Partitioner will decide where a key value pair from a map task goes to which reducer.



Bloom Filter false probability

Problem 2

Consider a bloom filter with an m-bit vector, and k hash functions: $h_1, h_2, \ldots, h_k$. What is the false positive probability for a membership operation?

- The probability that a specific bit is still 0 after a hash:
  - $p_1 = 1 - \frac{1}{m}$
- The probability that a specific bit is still 0 after a member has been hashed:
  - $p_2 = \left(1 - \frac{1}{m}\right)^k$
- The probability that a specific bit is still 0 after all members of S (the size is n) have been hashed:
  - $p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$
- For a non member, it may be found to be a member of S (all of its k bits are nonzero) with false positive probability
  - $(1-p)^k$

→ Probability a specific bit is 0 after a member is hashed is just p1 to the power of k because each value of the hash is independent.

1 job contains map phase and reduce phase. Map phase can contain multiple map tasks. A map task is run by a single machine. A single machine may run multiple map tasks. Suppose a hashmap is used as part of an in-memory combiner. This combiner is unique to this current Mapper object. This Mapper object can be run on multiple chunks, depending on which chunks the Hadoop runtime chooses the machine to run the tasks on. Now the interesting part: the hashmap for the Mapper object (as part of in-memory combiner) is shared across all the chunks that the mapper executed on. Each chunk consumes 1 map task.

Performance Considerations

- Parallelism, i/o and memory usage
Parallelism: Look at the number of partitions used, whether it is enough? For example, for joins, we can increase the number of partitions to perform the join. (I guess this is assuming that our buckets usually have clashes, then increasing the number of partitions reduces the number of clashes and increase parallelism)

i/o: Network and disk i/o. Operations like joins and mapreduce's shuffle are network bottlenecks. Mapreduce intermediate results have high disk i/o

Memory: If rdd cannot fit in memory then it is an issue, another example is in join, if dataset is skewed then one machine's partition might be much larger (which is also a memory issue)

Column DB vs Row DB

- Column DB calculates queries which involve only 1 or 2 columns very quickly, such as aggregation queries.
- Column DB has better compression because each column has a consistent data type, but in row DB, there can be as many data types as schema attributes to compress from for every tuple.
- Column DB are generally not great with queries that need multiple fields from each tuple.
- Column DB in general takes more time to write new data. Insertion can be done in 1 operation in row DBs, but each column needs to be written one by one in column DBs.

Pros of a Columnar Database

- Queries that involve only Row-Oriented Database columns
- Aggregation queries against vast amounts of data
- Column-wise compression

Cons of a Columnar Database

- Incremental data loading
- Online Transaction Processing (OLTP) usage
- Queries against only a few rows

- Pregel, Hadoop, Spark are all meant for batch processing, but spark can be used to perform streaming if we process small windows

Increasing the number of replicated chunks in HDFS

(C) In HDFS, each chunk is replicated for three times by default. Explain one strong point and one weak point to the Hadoop/HDFS system, if we replicate each chunk for four times rather than three times.

[2 marks]

Answer:
(good) system will be more robust. Scheduling is more flexible. MapReduce Performance could be better.
(bad) storage overhead and slow write performance.

Increasing number of reducers on a wordcount program

(D) Suppose we execute a MapReduce program that counts the number of occurrences of each word in a large repository of 10,000 different books. Suppose there are 1,000 distinct words this repository. The program is running on a cluster of 10 machines (each with a single CPU core). Currently we use 10,000 Map tasks (each for one book) and Y Reduce tasks. Explain how the settings of Y's value will affect the performance of the MapReduce program, and justify your answers. Consider Y's value from 1 to 1,000.

[2 marks]

Answer:
From 1 to 10: better performance due to parallelism.
10- a certain point X0: better performance due to multi-tasking on the same machine.
X0 to 1,000: performance become stable or even degrade due to saturation.

→ X0 to 1000: Overhead from maintaining/creating so many reducers

Design consideration for MapReduce across data centers

	US West	Ireland	Singapore
Bandwidth	25	19	6.6
Latency	0.16	0.17	0.35
Distance	Short	Medium	Long

Suppose that we have a data set with partitions that are distributed to the data centres shown in Figure 1. Each data centre has a partition of the data set. Suppose now we want to redesign Hadoop to run on the geographically distributed data centres. Suggest two potential design performance considerations for adjusting the original Hadoop system to extend its current design on a single data centre to this environment.

[2 marks]

Answer:
Examples include:
the network latency tends to increase and the network bandwidth tends to decrease, as the distance between two data centers increases.
Extend the combiner to geo data center.
HDFS Replication adaptation for geo-reliability (two within the data center, and one on another data center).

Using Hashmap for MapReduce

```
1: class Mapper
2:     method Initialize
3:         S ← new AssociativeArray
4:         C ← new AssociativeArray
5:     method Map(string t, integer r)
6:         S{t} ← S{t} + r
7:         C{t} ← C{t} + 1
8:     method Close
9:         for all term t ∈ S do
10:            Emit(term t, pair (S{t}, C{t}))
```

Figure 4. Computing the average: Version 4

Using combiner

- In the mapper we emit as the value a pair consisting of the integer and one—this corresponds to a partial count over one instance.
- The combiner separately aggregates the partial sums and the partial counts (as before), and emits pairs with updated sums and counts.
- The reducer is similar to the combiner, except that the mean is computed at the end.
- In essence, this algorithm transforms a non-associative operation (mean of numbers) into an associative operation (element-wise sum of a pair of numbers, with an additional division at the very end).

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, pair (r, 1))
1: class Combiner
2:     method Combine(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         Emit(string t, pair (sum, cnt))
1: class Reducer
2:     method Reduce(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         ravg ← sum/cnt
9:         Emit(string t, pair (ravg, cnt))
```

LSH Band Size 5 vs 1

If band size = 5, then there are less candidate pairs which will be matched, because more values need to match before they are considered candidate pairs. In verifying that they are candidate pairs, it is also faster since there are much less candidate pairs to verify.
- LSH cons – possibly high memory usage?