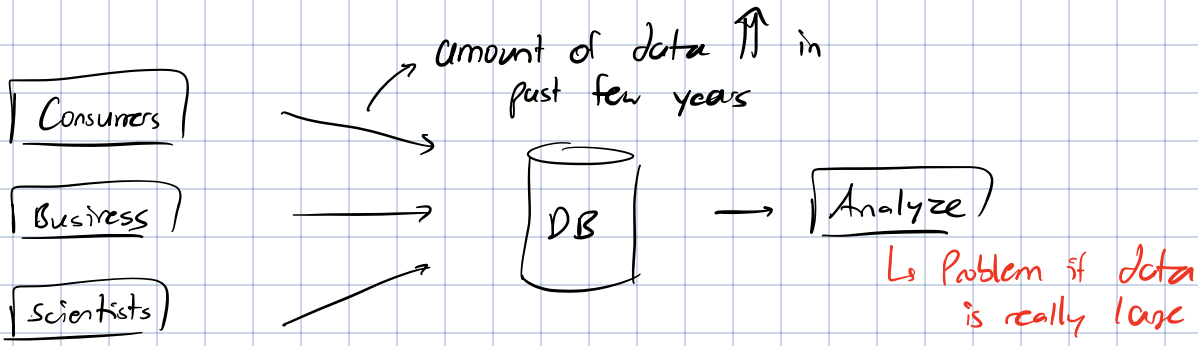
Introduction

Motivation



↳ Problem if data is really large

Analysis at scale leads to interesting sights

↳ Ex:// Target predicting pregnancy, Walmart used big data to design stores
black hole pictures

How to analyze lots of data at scale?

Vertical Scaling
- Really expensive HW can do work
- Cap on how much scale to achieve

focus of course

Horizontal Scaling
- Get more computers
- Cheap computes work together
  ↳ Parallelization
  ↳ Fault tolerance

We will use distributed computing frameworks as an abstraction

MapReduce

The Algorithm

Revolves around key-value pairs

↳ Ex:// In a textfile:

Functions

Map:

$(k_1, v_1) \rightarrow$ List $[(k_2, v_2)]$

not the same!

Reduce

$(K_2,$ list $[v_2]) \rightarrow$ List $[(k_3, v_3)]$

all values associated w/ K_2

Basically, extract keys & values from some KVP

Takes all values associated w/ one key => some operation

Ex://

Input:

0: "abcdaaa"
1: "bcdadc"
2: "dadb"

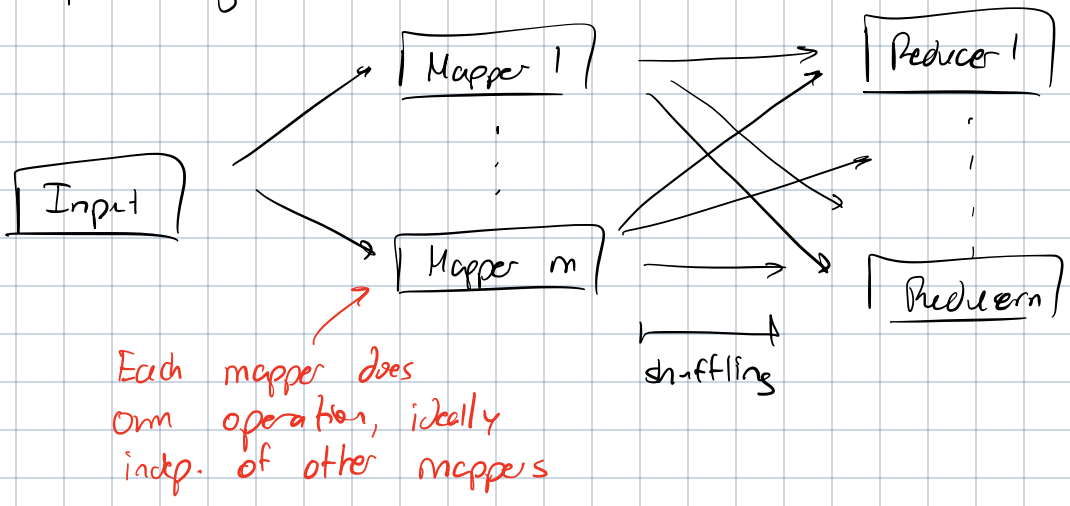
Output.

0: {a: 4, b: 1, c: 1, d: 1}
1: {a: 1, b: 1, c: 2, d: 2}

Ex:// If aggregating:

(a, [4, 1, 3]) —reduce→ (a, 8)
(b, [1, 1, 3]) —reduce→ (b, 2)
⋮

What is the design:



Shuffling: mappers partition particular keys s.t. all keys go to same reducer

↳ Partition: (k_1, v_1), $n \in \mathbb{N} \to [0, n)$
Assigns k_1 to some n (n = # of reducers)

Apache Hadoop

Coordinating computing nodes to run MapReduce is hard → Hadoop

What does it do:

① Assign workers to map & reduce tasks

② Divides data b/w mappers

③ Groups intermediate data → sends to right reducers

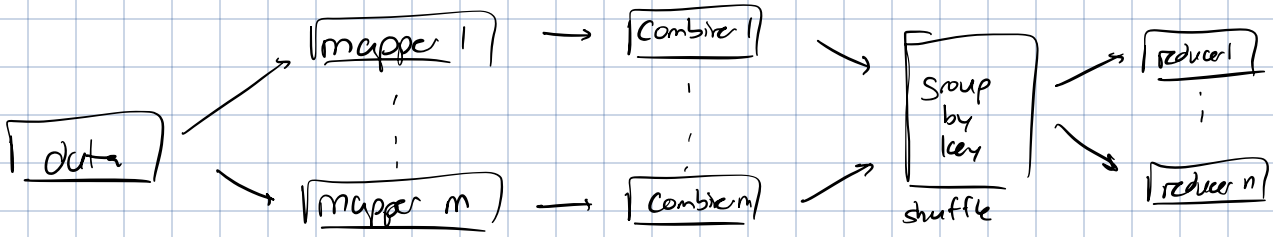
④ Handles errors

Slowest Hadoop op: sending data b/w mappers & reducers

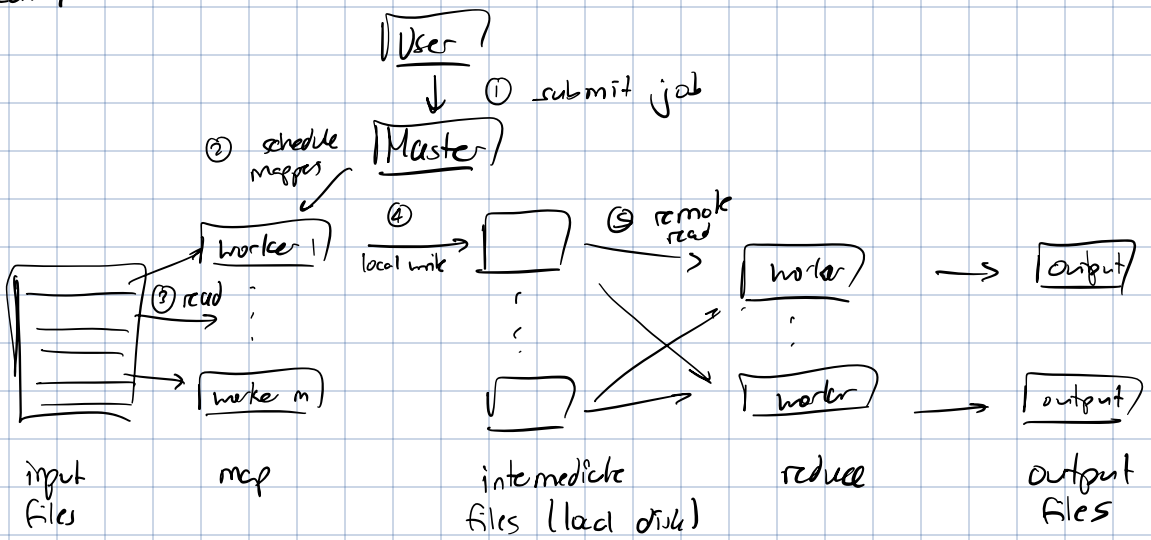
Combiner: $(k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_2, v_2)]$

↳ Take all values associated w/ k_2 & combine. Ideally, len(output) = 1

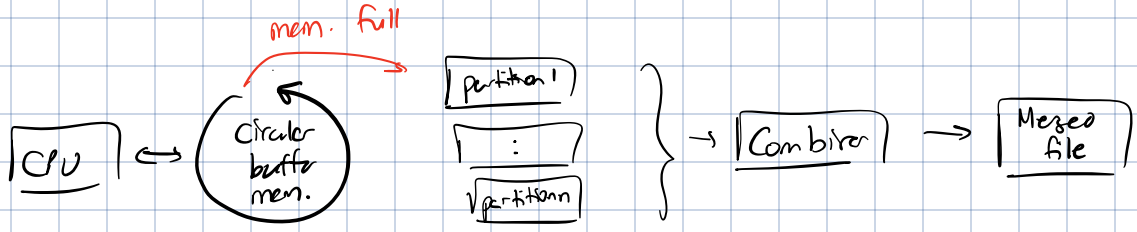
↳ May / may not function like reducer. Doesn't have to run ← decided by Hadoop



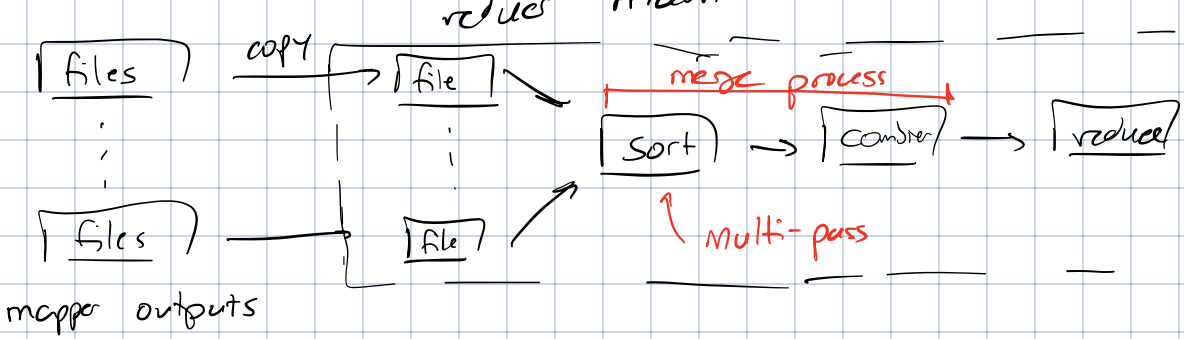
Physically:



Mapper:

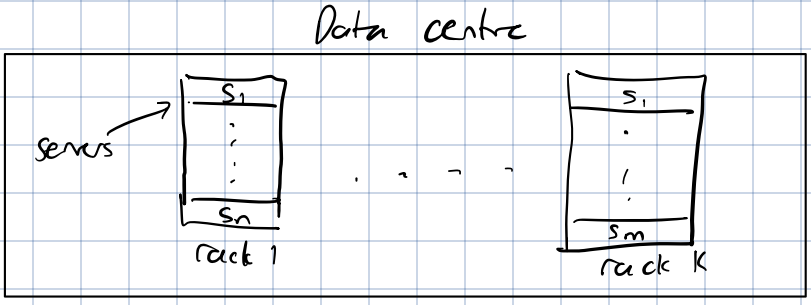


Reducer



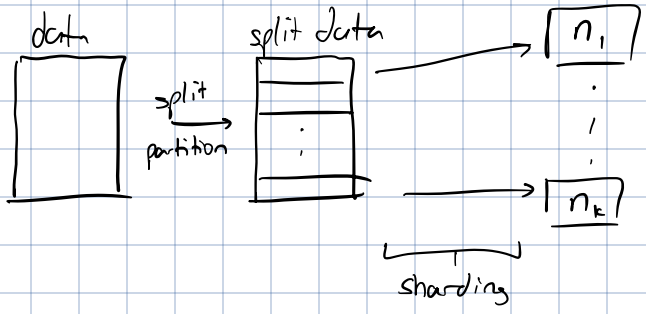
Distributed Storage

Physically:



If we need to perform data ops, locality matters → latency ↓, bandwidth ↑

To store a large amount of data across servers:



↳ To ensure fault tolerance, store each data partition on multiple nodes (replication)

Hadoop Distributed File System (HDFS)

Can handle 10K nodes, 100M files, 10 PB of data ⇒ very large distr. file sys.

Assumptions:

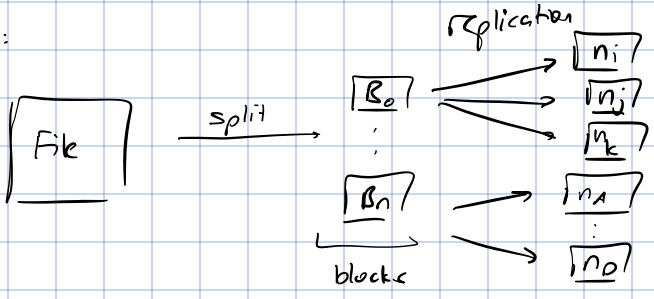
① Commodity hardware: file replication is baked in

② Batch processing

Basic principles:

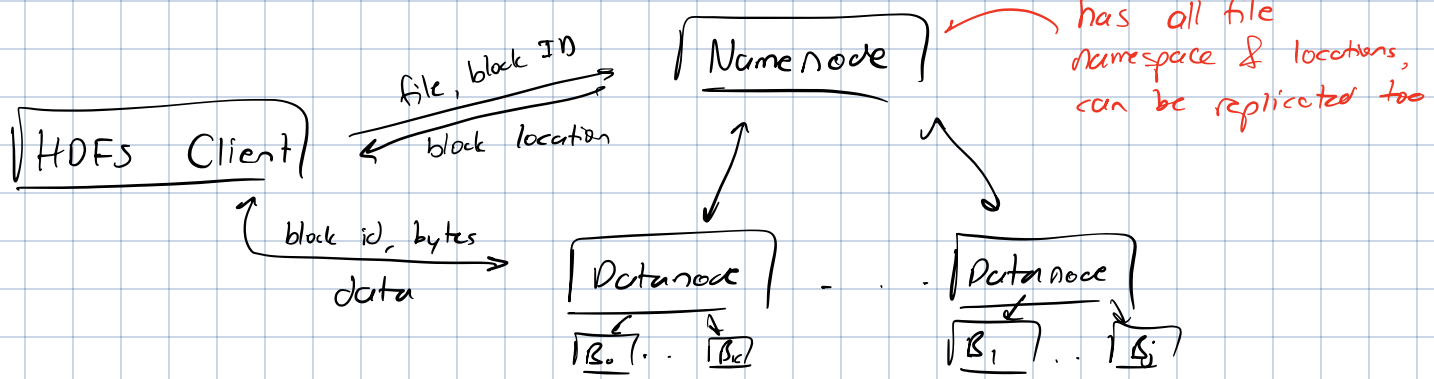
① Data coherency: enforces write-once-read-many data model → client can only append to files

② Blocking:



③ Intelligent client: client knows block location & can access it

Architecture:



Roles of namenode:

① Manage file system namespace
   i) File ↔ blocks
   ii) Blocks ↔ datanodes
   } all in memory

② Cluster configs. management

③ Block replication engine

④ File metadata & transaction log

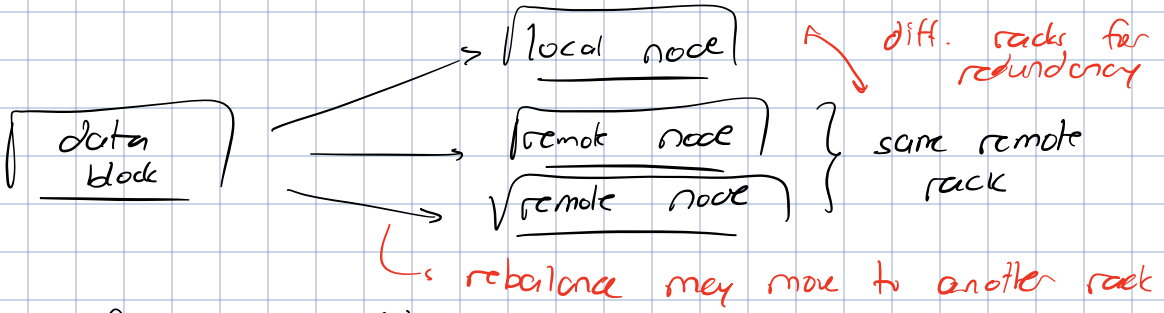
Roles of datanode:

① Store data & metadata of blocks

② Report block status to namenode periodically

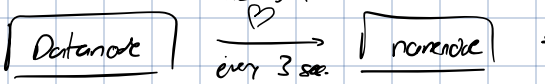
③ Forward data to other datanodes

Q1 Where exactly is data stored? Which datanodes? Default:



Client reads from closest block

Q2 How does namenode know if datanode failed?



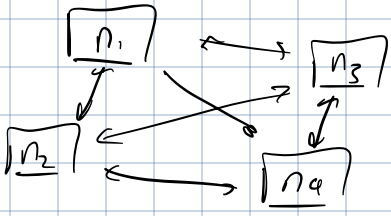
⇒ If no heartbeat → failure

On fail, Namenode chooses datanode for new replicas.

↳ Balances disk usage & comm. traffic

Q3 Which nodes actually do work on data?

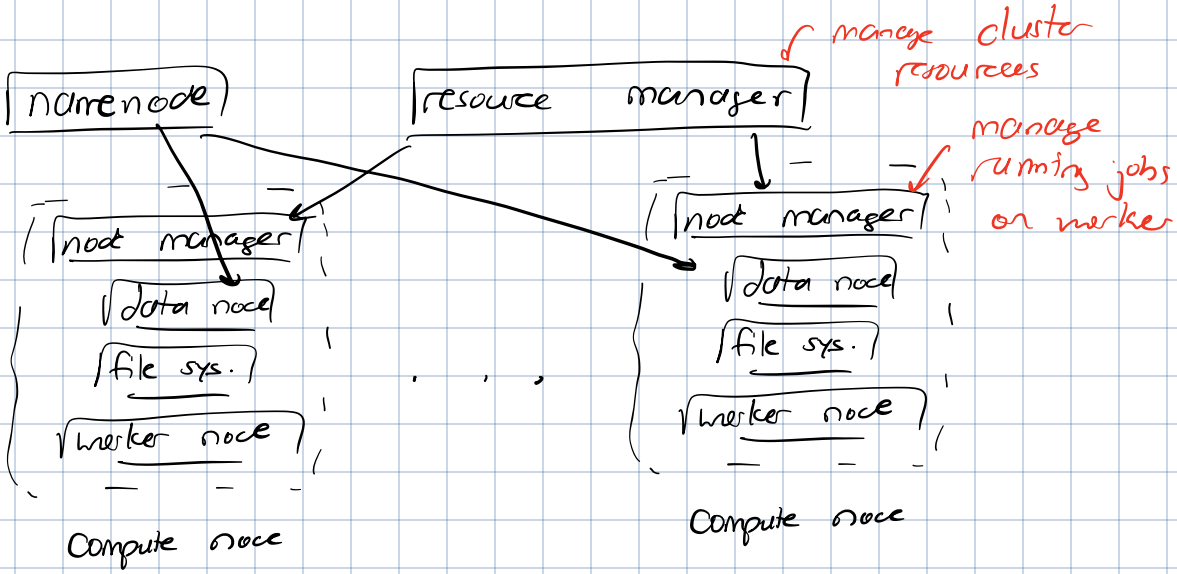
For compute-intensive tasks, can distribute data to compute nodes



} Compute time >> communication time

Problem for data-intensive comp: comm. time is too long!

Solution: do as much computation locally → Hadoop assigns workers to node, that already have data



Algorithm Design

① Combiner design

B/c combiners may not run, we need to ensure that it matches the input & output types of mapper & reducer

Why do we want combiners? Combiners are run locally to do file merges, reducing network traffic → latency ↓

② Class variables

Sometimes, it can hold enough data in-memory. Be careful

③ In-mapper combine / in-memory combiner

Defn: perform reduce in a map task → network traffic ↓

Pros: speed

Cons: memory management required

MapReduce To Spark

Higher-level Programming

Problem w/ Hadoop: too much boilerplate & repetition

Companies created abstractions: FB Hive, Yahoo's Pig

Spark

Fast & efficient cluster computational engine compatible w/ Hadoop

Efficient

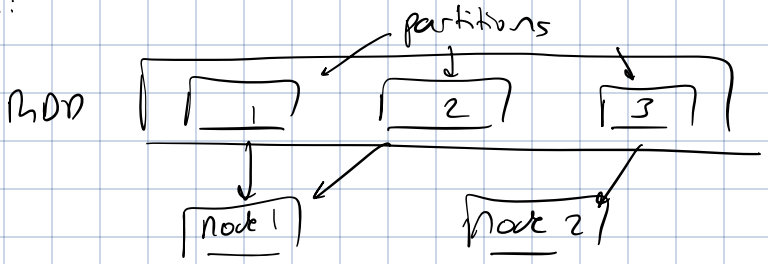
① Execution graphs & optimizations

② In-memory storage

Usable b/c lots of good APIs

Spark performs ops. on RDD: resilient distributed datasets

- Defn: collections of objects spread across cluster
- Resilient b/c auto-rebuilds on failure as it tracks lineage data for RDD
- Diagram:



Programming w/ RDDs

① Creating RDDs:

sc.parallelize(...) OR sc.textFile("...")

Spark context

① Basic functions:

A: map:

rdd.map(func) ⇒ run each elem. through func.

B: filter:

rdd.filter(pred.) ⇒ keep elements passing predicate

C: flatMap

rdd.flatMap(func) ⇒ map each element ∈ RDD to 0/more other elem.

D: collect

rdd.collect() ⇒ collects RDD into local obj.

E: take

rdd.take(k) ⇒ emit 1st k elements

F: Count

rdd.count()

G: Reduce

rdd.reduce(func) ⇒ merge elem. w/ associative func.

H: SaveAsTextFile

rdd.saveAsTextFile("hdfs://...")

② Key-value pair ops.

Spark's distributed reduce transforms requires RDDs of KVPs

A: reduceByKey: rdd.reduceByKey(..) ⇒ Each key has all assoc. values reduced.

B: groupByKey: rdd.groupByKey() ⇒ merge KVP into key: [values...]

C: sortByKey: ll.sortByKey()

All ops. take a parallelism level operator

④ DAG:

Spark makes DAG for all ops. & uses it to perform efficient ops.

Data locality & partitioning is factored in

Spark Architecture

YARN is used to handle scheduling of jobs & resource management in Hadoop 2.0

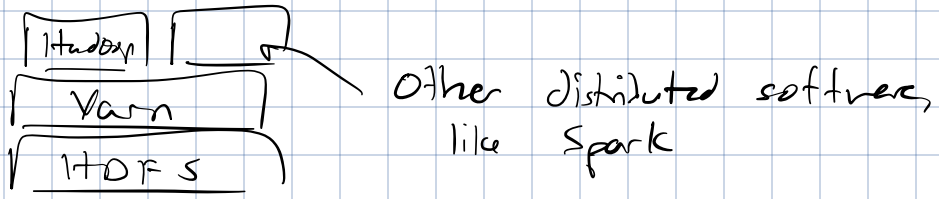
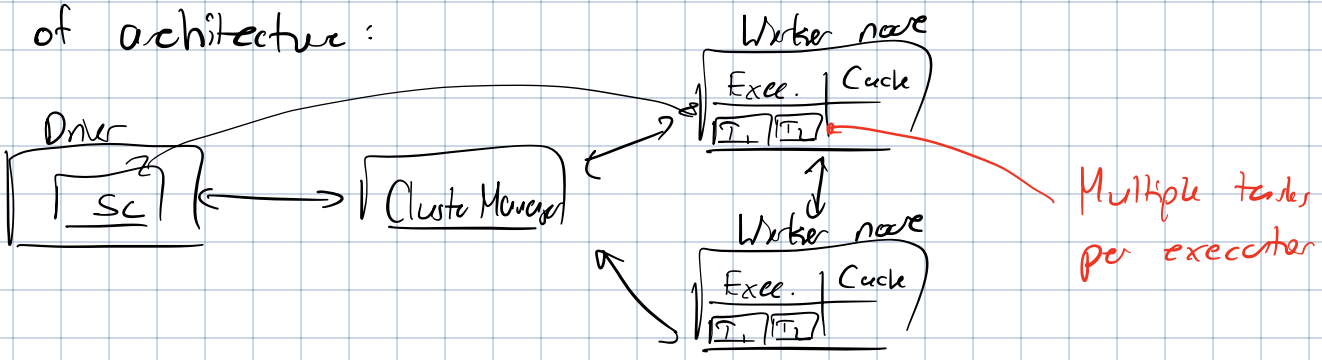


Diagram of architecture:



Driver sends Spark code on a per task level -> memory problems.

Solution: broadcast

- Spark sends copy of value to executor, not just per task
- Broadcast values are read-only

To communicate back to driver, use sc.accumulator(...)

- Driver can inspect, workers can only write

Spark uses hash partioner by default, but you can override.

Algorithm Design

① reduceByKey

- Reduces b/f & after shuffle!

② CombineByKey

- More fine-grained reduceByKey
- Signature:

RDD.[(K, V)].combineByKey(create, append, merge) => RDD[(K, C)]

create: V → C
append: C + V
merge: C + C

③ Aggregate By Key

- B/w Combine By Key & reduce By Key

RDD.[(K,V)].aggregateByKey(init, append, merge) ⇒ RDD[(K,C)]

init: initial value of type C; append: C+V; merge: C+C

④ Group By Key

Needed only if not reducing [K, List(V)]

Analyzing Text

Natural Language Processing

Aim: create a probabilistic model $P(w_1, w_2, \ldots, w_k)$

- All LLMs are doing is given $w_1 \ldots w_k$, find prob. distribution of w_{k+1}
- Conditionally: $P(w_1, \ldots, w_k) = P(w_1) \cdot P(w_2 | w_1) \cdot P(w_3 | w_1, w_2) \cdots$

Using this to calculate $P(w_{k+1})$ is hard! Why? length |sentence| is unbounded so possibly LOTS of calculation

We can scale it down so we only use past n-words to construct prob.

- This is called n-grams
- Ex:// If bigram:

$$P(\text{I like ham}) = P(\text{I}) \cdot P(\text{like} \mid \text{I}) \cdot P(\text{ham} \mid \text{like})$$

Problem: unlikely words can make prob → 0

- Solution:

① Robin Hood: every $P(n\text{-gram}) > 0$

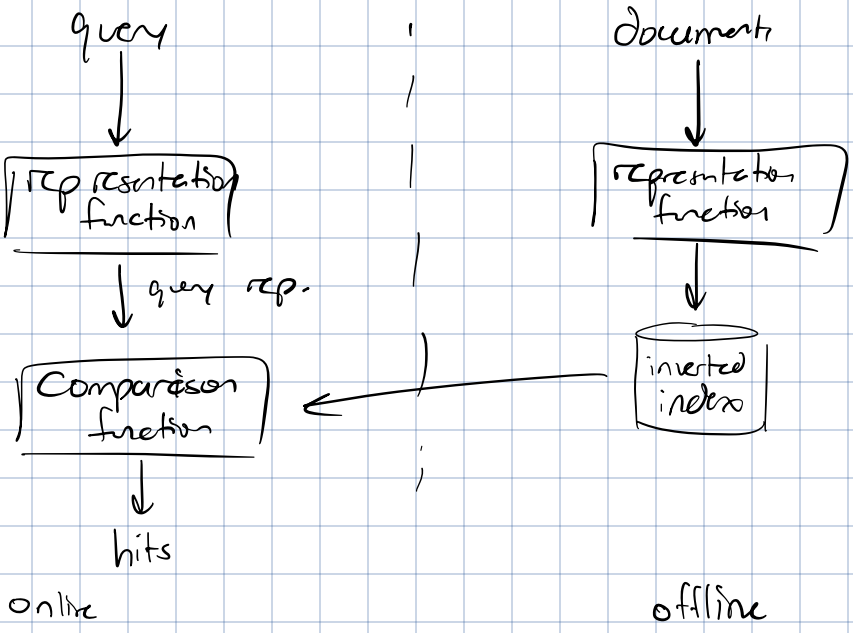
② Laplace smoothing: every count of a n-gram starts at 1

$$P(A, B) = \frac{\text{Count}(A, B)}{N} \rightarrow P_L(A, B) = \frac{\text{Count}(A, B) + 1}{N}$$

This requires adding V^2 to N (V big vocab size)

Searching & Indexing

Abstract information retrieval (IR) architecture



Representation function:

① Take text, tokenize & case fold (lower case & Unicode → canonical form)

② Create embeddings

Inverted index: context → documents. (forward index: documents → context)

term1 → document1, document2, ...
term2 → document2
⋮

} Postings list

Scaling:

- Heap's Law: $M = kT^b$ (vocab size = M, # of docs = T, constant k ≈ 30–100, constant b ≈ 0.4–0.6) ⇒ vocab size grows unbounded
- Zipf's Law: $f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N} 1/n^s}$ (k = rank, N = # of elem, s = exp.) ⇒ Few elem occur v. freq., Many elem occur v. infreq.

Map Reduce:

Mapper

```
def map(docid: Long, doctext: String)
    counts = Counter()
```

Reducer

```
def reduce(term: strin, postings):
    p = list()
```

```
for term in tokenize(doctext):
    counts.add(term)
for term, freq in counts:
    emit(term, (docid, freq))
```

Should only be big for few terms ⇒

```
for docid, freq in postings:
    p.append((docid, freq))
p.sort()   ← Hadoop is good at sort.
emit(term, p)
```

Delta encoding:

- Insight: if a term is common, delta of associated docs w/ each other is small
- Store only $\nabla(doc_i, doc_{i+1})$ for common terms
- V. useful if dealing w/ variable-length ints since it can be compressed
- Variable length ints: 1000SLLL — L bytes for x
  - 1000 special
  - S ↳ 0 if +, 1 if −

Elias-γ code: another compression format

- Assumptions: dealing w/ nat. # w/ no upperbound, smaller # more common (power law)
- Encoding x:
  1. $N = \lfloor \log_2 x \rfloor$
  2. Write N 0's
  3. Write x as $N+1$-bit #

Decoding x

1. Read 0s until 1, call it N
2. Interpret $N+1$ bits as binary #

Golomb Code: assume x is what we want to encode & decode, $x > 0$

1. $z = \frac{\#\text{ of docs w/ term}}{\#\text{ of total docs}}$
2. $M = \left\lceil \frac{\log(2-z)}{-\log(1-z)} \right\rceil$
3. $q = (x-1)/M \Rightarrow$ quotient
4. $r = x - qM - 1 \Rightarrow$ remainder

Unary encoding: n is n 0s, then 1
↳ 5 ⇒ 000001, 0 ⇒ 1

Truncated binary:

1. $k = \lfloor \log_2 n \rfloor$
2. $u = 2^{k+1} - n$
3. First u codewords: first u codes w/ length k
4. Last $n-u$ codewords: last $n-u$ codes w/ length $k+1$

- Each term gets own encoding scheme

Ex: // Encode $n = 15$ #s in truncated binary

1. $k = \lfloor \log_2 15 \rfloor = 3$
2. $u = 2^{3+1} - 15 = 1$

③ First U codewords ($\{0, \ldots, 14\}$) is code w/ length k

$\therefore U=1 \Rightarrow 0 \rightarrow \underbrace{000}_{k=3}$

② Last $n-U$ codewords: $k+1$ length

$1 \Rightarrow 0010$

$2 \Rightarrow 0011$

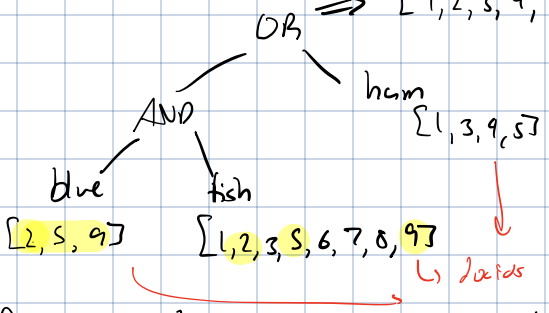
We can also collect term frequency for each document, but need to use a diff. approach

Indexing is a perfect problem for MapReduce

Searching & Retrieval

Boolean retrieval: take Boolean query → query syntax tree → for each leaf, look up postings list to find docids → apply query tree → return hits (unsorted)

- Ex:// Q: (blue AND fish) OR ham:



Term-at-a-time: for each term, generate docs, then apply Boolean ops.

- We may need inverted sets for dealing w/ NOT operators

Document-at-a-time: run query tree against each document

- If index partioned by document, this is nice → each partition has all term for a document, so no cross-partition queries needed

What if we want to sort output by relevance? Ranked retrieval

↳ No Boolean queries b/c we want to allow documents that might not have all terms

↳ Insight for relevance:

① If doc. has a term repeated many time (high term freq.), should be relevant

② Should be weighted according to commonality of term (doc. frequency)

TF-IDF:

total # of docs

$w_{ij} = tf_{ij} \log N/df_i$

- w_{ij}: weight of term i in doc j
- tf_{ij}: # of occurrences of term i in doc j
- df_i: # of docs containing term i

- Relevance of doc j: $\sum_{i=1}^{N_i} w_{ij}$

Document-at-a-time ranked retrieval:

- Algo:

```
for each doc:
    score each query term ⇒ sum
    accumulate best k hits via min-heap
```

Time: $O(n \log k)$
Space: $O(k)$

- Con: can't terminate early, need to look @ all docs.

Term-at-a-time ranked retrieval

- Algo:

```
collect hits & rankings for rarest term in accumulator
for each other term in query
    if term ∉ doc:
        remove doc from acc.
    else:
        add term ranking to overall ranking
```

- Early end possible, but uses memory

Tradeoff b/w 2. Commonly, docs partioned by ID & sorted by quality ⇒ can do "early termination" document-at-a-time.

ANALYZING GRAPHS

Graph Review

Representing graphs:

① Adjacency matrix

$n \times n$ matrix (n = # of nodes). $M_{ij} = \begin{cases} 1 & \text{if node } i - \text{node } j \\ 0 & \text{o.w.} \end{cases}$

	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0

Pros: matrix is great for math

Cons: wasted space w/ lots of 0s → sparsity leads to wasted space

② Adjacency list

Row i is list of out-neighbors of vertex i

1 : 2, 4

2 : 1, 3, 4

3 : 1

4 : 1, 3

Pro: smaller than adj. matrix & easy to find out-neighbors of v

Cons: Difficult to find in-neighbors of vertex

③ Edge list

List all the edges

Pros: simple. Cons: not useful in finding neighbors

Web Data

Web is basically a huge graph, but quite sparse (most pages have 0/1 link)

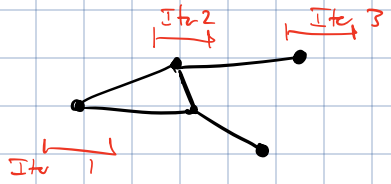
We use adj. lists for graph representation

Problem: find shortest path b/w vertex u & v:

- We could use Djikstra's, but not parallel. Instead use parallel breadth-first search (pBFS).
- Algo:

$$dist(s) = 0$$

$$dist(v) = \begin{cases} 1 & \text{if } s \text{—} v \\ 1 + \min_u (dist(u)) & \text{o.w. } \forall u \text{ s.t. } u \leftrightarrow v \end{cases}$$



- Hadoop

  D.S.: { node n : (d: distance to n, adj list of n) }

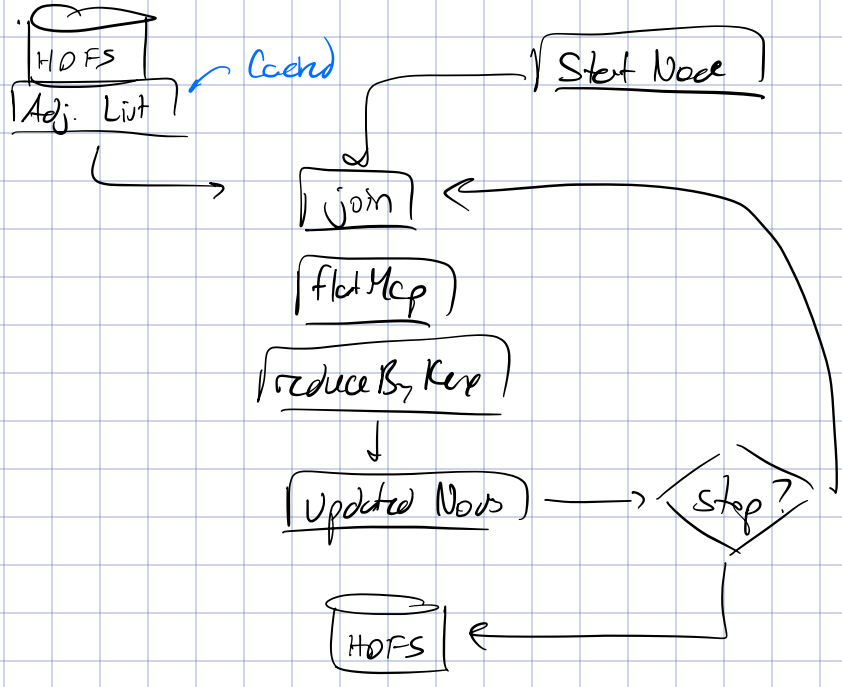
  Mapper:
  ```
  emit (id, node)
  for m in adjList:
      emit (m, node.d + 1)
  ```

  Reducer:
  ```
  d = infinity
  node = None
  for o in vals:
      if isNode(o): node = o
      else: d = min(d, o)
  node.d = min(node.d, d)
  emit (id, node)
  ```

- Can modify to actually give path:
  1. Message includes path that gives length d → bigger message
  2. Node has a "prev" field → requires reconstruction

- Problem w/ Hadoop: if we want to do this for all nodes, we need to read from disk a lot. Also need to keep sending graph struct to reducers every iteration

- Spark



Page Rank

Ranked retrieval fails b/c website w/ more terms != more relevant

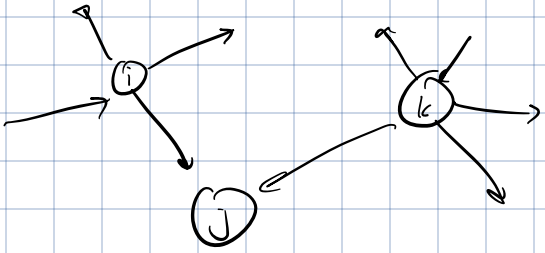
Instead, proxy will be # of in-links, w/ more important in-link weighted more

If page j w/ rank r_j has n out-links, then each link counts

as r_j/n votes.

Then: rank of page k = $\sum$ ranks of in-links

Ex://



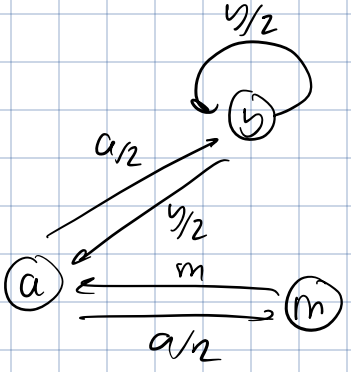
$$\Rightarrow r_j = \frac{r_i}{3} + \frac{r_k}{4}$$

Stochastic adjacency matrix:

Let M be the matrix, page i has d_i out-links.

$$M_{ji} = \begin{cases} 1/d_i & \text{if } i \to j \\ 0 & \text{o.w.} \end{cases} \Rightarrow \text{Cols. sum to 1}$$

Ex://



$\Rightarrow$

	y	a	m
y	1/2	1/2	0
a	1/2	0	1
m	0	1/2	0

Solving for ranks:

① Set $r_j = 1/N$ (N = # of nodes)

② $r'_j = \sum_{i \to j} r_i/d_i$

③ $r_j := r'_j$

} until convergence

Random walk interpretation: assume surfer is following links on web.

$\vec{p}(t)$: vector, i^{th} comp. is P(surfer on page i at time t)

$\therefore \vec{p}(t+1) = \underbrace{M}_{\text{stochastic matrix}} \cdot p(t)$

Important result: stationary distn. is unique & will be reached for graphs that satisfy certain conditions

Problems to solve:

① All out-links are w/in a group (spider-traps) → importance is too high

↳ Google soln: teleports

- Surfer follows random out-link w/ prob. of β, can also jump to random page at prob. $1-\beta$

W/out this, ranks are too high for groups

② Some pages have no outlinks (dead-ends) → importance "leaks out"

↳ Soln: always teleport from deadends

Ex://

	y	a	m
y	1/2	1/2	0
a	1/2	0	0
m	0	1/2	0

Dead-end

transform →

	y	a	m
y	1/2	1/2	1/3
a	1/2	0	1/3
m	0	1/2	1/3

W/out teleports, colums are not stochastic

Google's Page Rank formula:

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

⇒ Assums M has no dead-ends

↳ # of out-links for d_i

Page Rank w/ MapReduce

① Simple model: no random jumps & dead-ends

Map: node sends importance to out-links

Reduce: node sets importance to sum of received values

```
def map(id, n):
    emit (id, n)
    p = n.rank / len(n.adj)
    for m in n.adj:
        emit (m, p)
```

```
def reduce(id, msgs):
    n = None
    sum = 0
    for o in msgs:
        if o is node: n = o
        else: sum += o
    n.rank = s
    emit (id, n)
```

② Random jumps

```
def reduce(id, msgs):
    n = None
    sum = 0
    for o in msgs:
        if o is node: n = o
        else: sum += o
    n.rank = s * β + (1-β)/N
    emit (id, n)
```

→ Prob. of jumping to any random node

③ Dead-ends

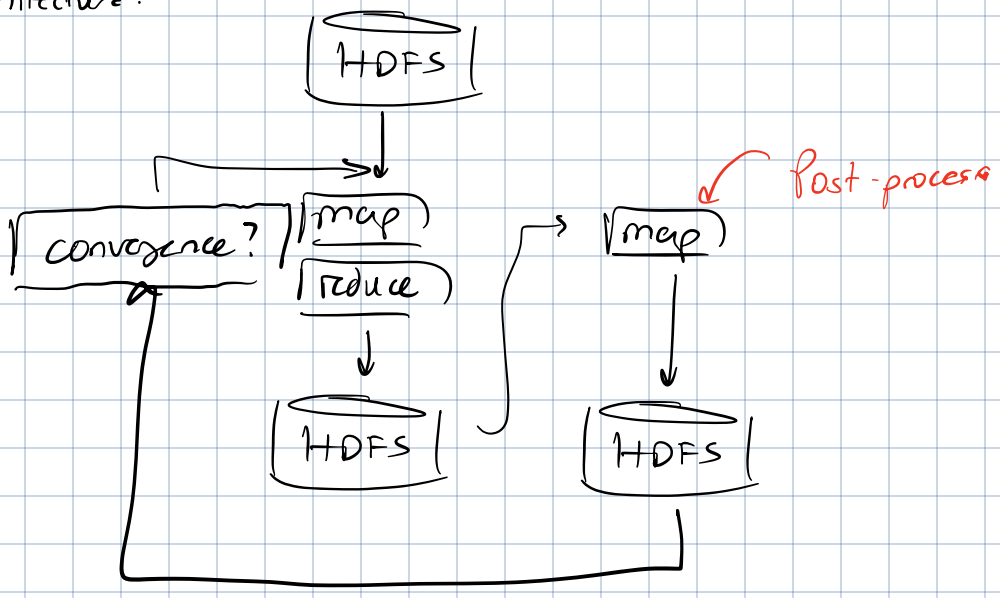
Option 1: add links to all nodes ⇒ too many msgs.

Option 2: post-process

R = sum of ranks of all nodes.
$1-R$ = " " deadends.

∴ Add $\frac{1-R}{N}$ to all nodes.

Architecture:



Alternative:

Mapper:

1. β(node.rank) to all out-links
2. $(1-\beta)$(node.rank) to "everyone"
3. node.rank to "everyone"

Reducer:

S += "everyone" · 1/n

Can we reduce the amount of space? Yup! Use log masses.

Q1: What is log mass?

Let $a = \log m$, $b = \log n \Rightarrow m = e^a$, $n = e^b$

$\therefore m \times n = e^{a+b} \Rightarrow m \times n \sim a + b$

Expanding:

$$m + n = \begin{cases} b + \log(1 + e^{a-b}), & a < b \\ a + \log(1 + e^{b-a}), & a \geq b \end{cases}$$

Q2: Why is this helpful?

Ranks $\in (0, 1)$. $\log(\text{rank}): (0, 1) \rightarrow (-\infty, 0)$

Uses more of the floating pt. range & won't have underflow b/c few vals will be close to

PageRank w/ Spark

Problems w/ MapReduce:

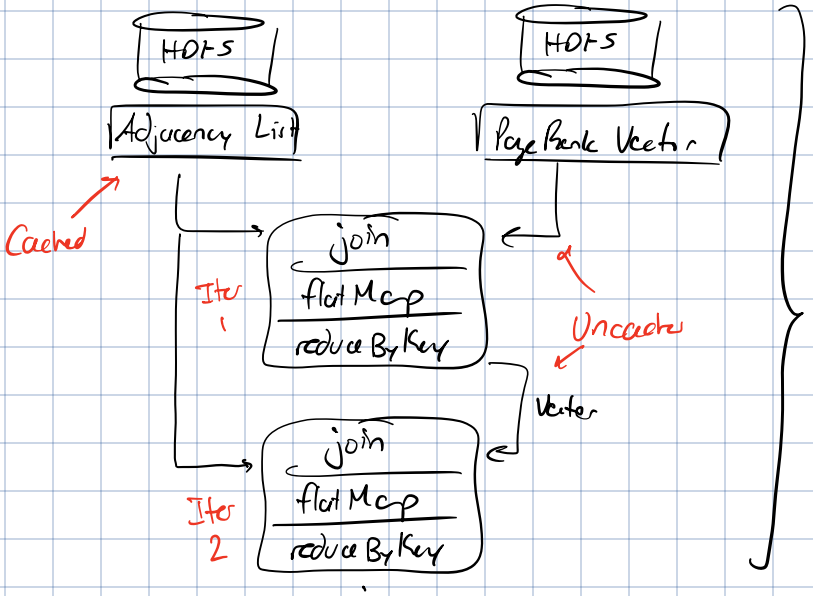
① Sending adjacency lists w/ each iteration → unnecessary messages

② Too much shuffling & file system access

③ Verbose

④ Each iteration of PageRank is a new job → lots of time taken on startup

Spark architecture:



Page Rank Improvements

W/ Page Rank, we no longer have term spam that would elevate importance in trad. TF/DF algos.

Problem #1 w/ Page Rank: Spam Farming

- Post a website on a website w/ high rank → your website rank ↑
- Alternately, spider trap: a bunch of websites just point to 1 website
  - ↳ Random jump in Page Rank avoids accumulating too much rank in website, but still high
- Soln:
  1. Add a "nofollow" attr. to links posted on popular websites to prevent rank inflation
  2. Personalized page rank: can set trustworthy pages to start Page Rank

Q: How to determine if a website is trustworthy?

Trust Rank:
1. Set small set of ultra-trustworthy websites to source nodes in personalized page rank, set to 1
2. Run personalized Page Rank → every website has trust score $\in [0, 1]$
3. Set trust threshold & cut off websites under threshold

NOTE: spam pages usually don't make cut b/c v. few trustworthy websites link to them

NOTE: personalized page ranks → teleport to known good websites (source nodes) → only good websites get rank

Need to ensure that starting set is trustworthy & big enough

Algorithm to enhance trust set:

1. Run Page Rank
2. Select top k pages as seed
3. Run TrustRank
4. Select from websites w/ trust rank ≥ threshold ⇒ remove spam

Q: How can we then determine definitively if a page is spam?

r_p := Page Rank of page p

r_p^+ := " , random jumps only lead to trusted source

r_p^- := $r_p - r_p^+$ ⇒ Contrib. of low trust pages to rank

S_p := $\frac{r_p^-}{r_p}$ ⇒ Spam mass

$S_p \uparrow$ → more likely p is spam

DATA MINING / MACHINE LEARNING

Supervised ML

Defn: give model training set where you already label data w/ soln. Model learns & then applies to new data

Data is represented as vectors of features. } Also called embeddings

↳ Features can be sparse: lots of 0s

You want to choose good features! Don't choose everything

Process:

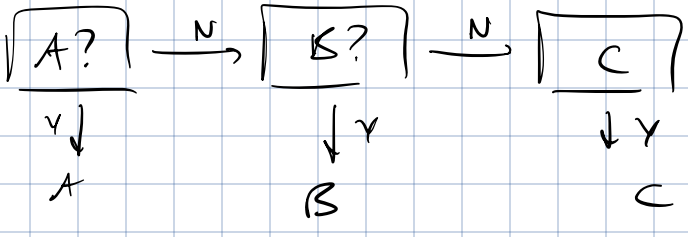
① Acquire data: usually want as much as you can for better perf.

② Determine labels of data: can crowdsource, bootstrap or exploit user behaviour

③ Choose model.

We will use binary classifier as building block.

More complex models use cascade of binary clf. to do multi-label classifier



④ Train model

Want to find function f s.t. it minimizes

$$\frac{1}{n}\sum_{i=1}^{n} \ell(f(x_i, \theta), y_i)$$

(x_i: data; θ: model params.)

↳ ℓ is a loss fxn.

We often do this computationally via gradient descent

↳ For each component x of θ, find $\frac{\partial \ell}{\partial x}$ & go downhill → compute θ_{i+1}

$$\therefore \theta' = \theta - \gamma \nabla L(\theta)$$

(γ: step; $\nabla L(\theta)$: gradient)

How do we avoid getting stuck in a local minima?

1. Momentum
2. Stochastic gradient descent: randomly select subset of training data

In binary clf, we use logistic function:

$$f(X; w, b): \mathbb{R}^d \to [0, 1] = \sigma(w \cdot x)$$

$$\downarrow$$

$$\sigma(z) = \frac{e^z}{e^z + 1}$$

Then:

$$\ell(y, t) = \frac{1}{2}(y - t)^2$$

This bounds the loss & it is differentiable

Q: Can we run gradient descent on MapReduce?

Yes, but we need to run multiple jobs (1 per iteration). This incurs:

1. Startup time
2. Need to retain state b/w iter.
3. Bottlenecked jobs (1 reducer)

Q: Could Spark do better?

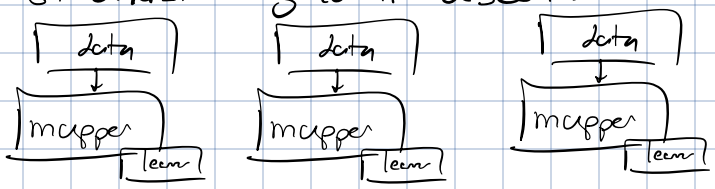
Yup! It caches data, no shuffles & driver only does 1 reduce.

Ensemble learning: use multiple models indep. & combine predictions

↳ Works b/c errors or uncorr., P(all models wrong) ↓, variance reduced

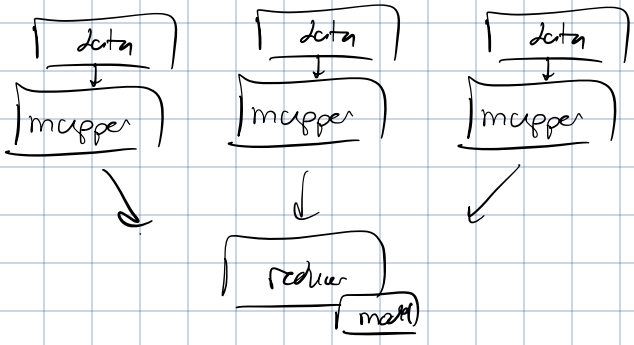
Can use this for stochastic gradient descent!

no iteration.



→ Each mapper stores model params.

OR: mapper is parser & reducer is learner:

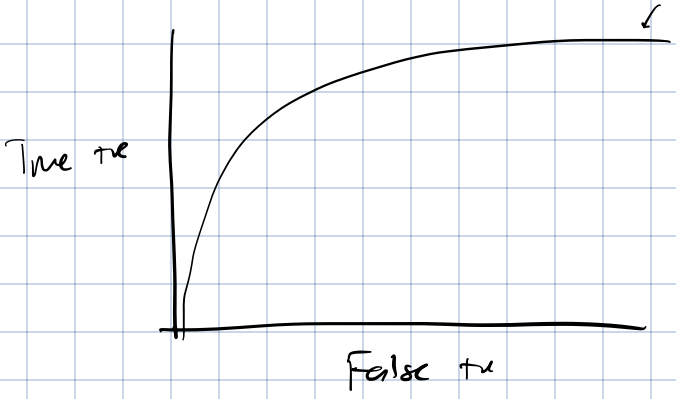


Model Evaluation

Recall that loss measures fit of model; we want to see how well model does on unseen data

Many metrics:

① ROC curve:



very discrimination threshold
(e.g.: if P(mal) = 0.1 → be
⋮
0.9 → be)

Perfect ROC curve → area is 100%.

Can use k-fold cross-validation to continuously test & train model

Embeddings

Problem w feature embeddings is that they can be really big & sparse

Solution: reduce dimensions

↳ One soln. is to hash → modulo by some prime.

Nearest neighbors problem

Obj.: Given high-dim. data points $x_1 \dots x_n$ and distance function $d(x_1, x_2)$, find all pairs (x_i, x_j) s.t. $d(x_i, x_j) < s$

① Naive approach:

Compute $d(..)$ for all pairs → $O(n^2)$

② Locality sensitive hashing approach

Locality sensitive hashing: items that are "close" will have hashes that are "close"

Also:

1. Make LSH table
2. Use buckets of width w. Buckets can overlap
3. Most values x_i, x_j s.t. $d(x_i, x_j) < s$ will have at least 1 bucket in common

How to make locality sensitive hash algorithm (specifically for sets)

↳ Jaccard similarity:

Let C_1 be set of n-grams of doc 1, C_2 for doc_2

$$sim(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}$$

$$d(C_1, C_2) = 1 - sim(C_1, C_2)$$

For multisets (elem. can appear multiple times), union & intersect defn. changes:

If $C = A \cup B$, then $C[k] = A[k] + B[k]$ (# of times k in A)

If $C = A \cap B$, then $C[k] = \min(A[k], B[k])$

For text, we will use n-grams as embeddings. size of n depends on doc size

Recall that sets & vectors can be represented as each other:

Sets → vectors:

1. Label all elem. of $\mathcal{U}$ set from $1 \to n$
2. If set contains i, then vector at $i = 1$, o.w. 0

Min-hashing: let C be a set of n-grams and π is an enumeration func. (n-gram → int.)

$$h(\pi; C) = \min_{i \in C} \pi(i)$$

Property: $P(h(C_1) = h(C_2)) = sim(C_1, C_2)$

↳ Proof: $y = h(C_1 \cup C_2) \Rightarrow y = h(C_1)$ or $h(C_2)$.

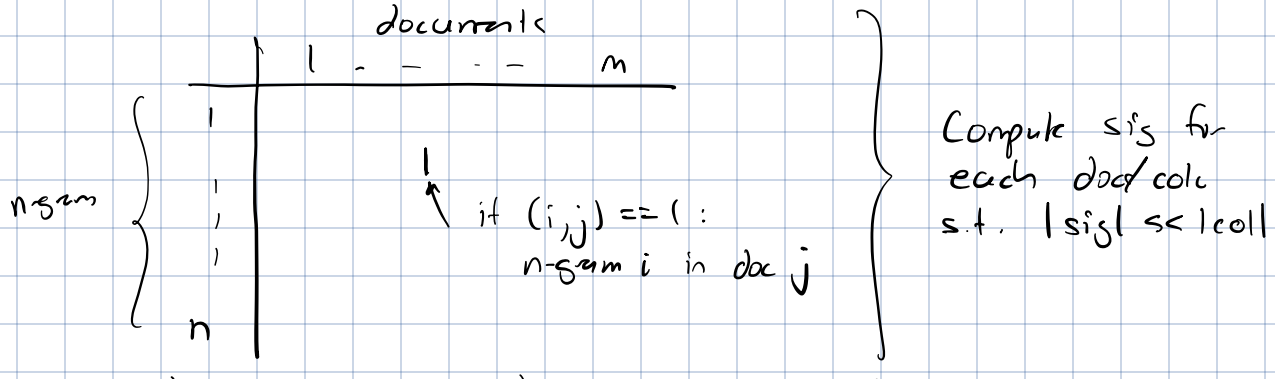
This is because C_1 or C_2 might be smallest value of π, we don't know since it depends on π

Only same if in intersection of C_1 & C_2

$$\therefore P(h(C_1) = h(C_2)) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}$$

This property indicates that $h(\pi; C)$ is an LSH

We want to do the following:



Define: $sig(C)[i] = h(\pi_i; C)$ ⇒ min-hash for i^{th} perm out of K.

↳ $|sig| = K \times 4$ bytes << $|col|$

Ex:

π_1	π_2	π_3
2	4	3
3	2	4
7	1	7
6	3	2
⋮	⋮	⋮

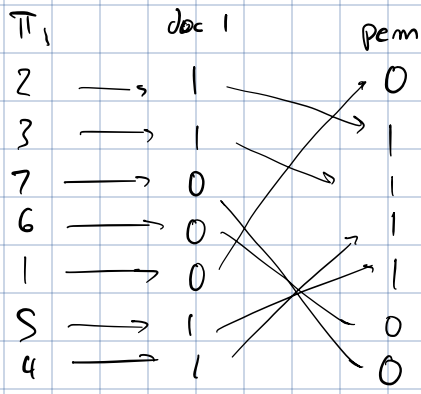
input matrix

	doc 1	doc 2	doc 3	doc 4
n.s. 1	1	0	1	0
n.s. 2	1	0	0	1
n.s. 3	0	1	0	1
n.s. 4	0	1	0	1
⋮	0	1	0	1
	1	0	1	0

Compute signature for documents.

Steps:

1. start w/ doc 1. Recode according to π_1



Problem: calculating π functions is too much!

2. Find row index w/ first 1:
   This case, index 2.
3. Repeat for π_2 & π_3, ...
4. Combine to form $sig(C_1)$

Define $sim(C_1, C_2)$ = % of entries that are equal.

↳ For signatures, just compare $\frac{\#\text{ of indices in same order}}{\text{sig. size}}$

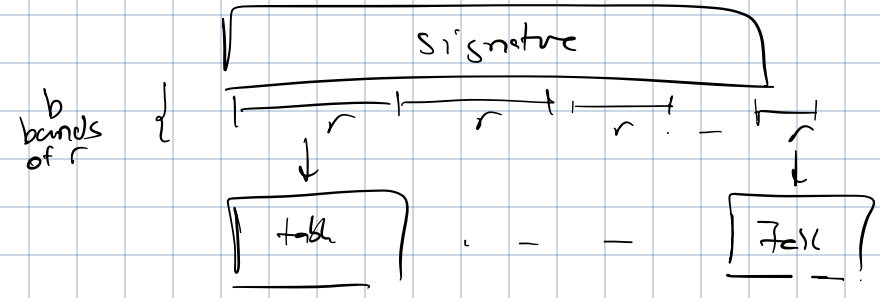
We can create a k-universal hash function:

$$h(x) = c_1 + c_2 x + \ldots + c_k x^{k-1} \mod p$$

↳ Ensures that $h(x_1) \ldots h(x_k)$ do not correlate.

Another problem: signatures are vectors → still need to compute if signatures of 2 docs are similar, which requires n^2 comparisons (if n = length of sign.)

Soln:



If 2 signatures collide in any ID table, signatures are candidate pair

Ex// Assume 100k docs, each w/ signature of 100 ints. Choose $b = 20$ bands and $r = 5$ integers / band. Want to find docs pairs that are $s = 0.8$ similar

Assume $sim(C_1, C_2) = 0.8$

↳ $P(C_1 \& C_2$ identical in 1 band$) = (0.8)^5 = 0.328$

$P(C_1 \& C_2$ not similar in all 20 bands$) = (1 - 0.328)^{20} = 0.00035$

This tells us that 1/3000 of our similar col pairs are false neg. So we can find 99.965% of sim. documents.

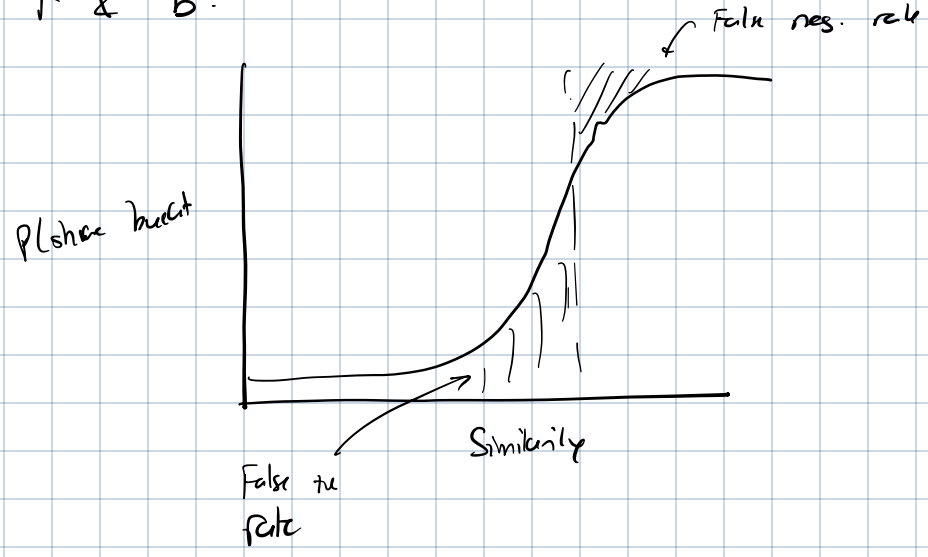
What if $sim(C_1, C_2) = 0.3$

↳ $P(C_1 \& C_2$ identical in 1 band$) = (0.3)^5 = 0.00243$

↳ $P(C_1 \& C_2$ are identical in 1/20 bands$) = 1 - (1 - 0.00243)^{20} = 0.047$

↳ ~ 5% of docs of 0.3 sim are considered candidate pairs

To pick r & b:



Summary:

① Doc → n-grams → integers

② Generate universal hash functions

③ Compute signature vector for each doc

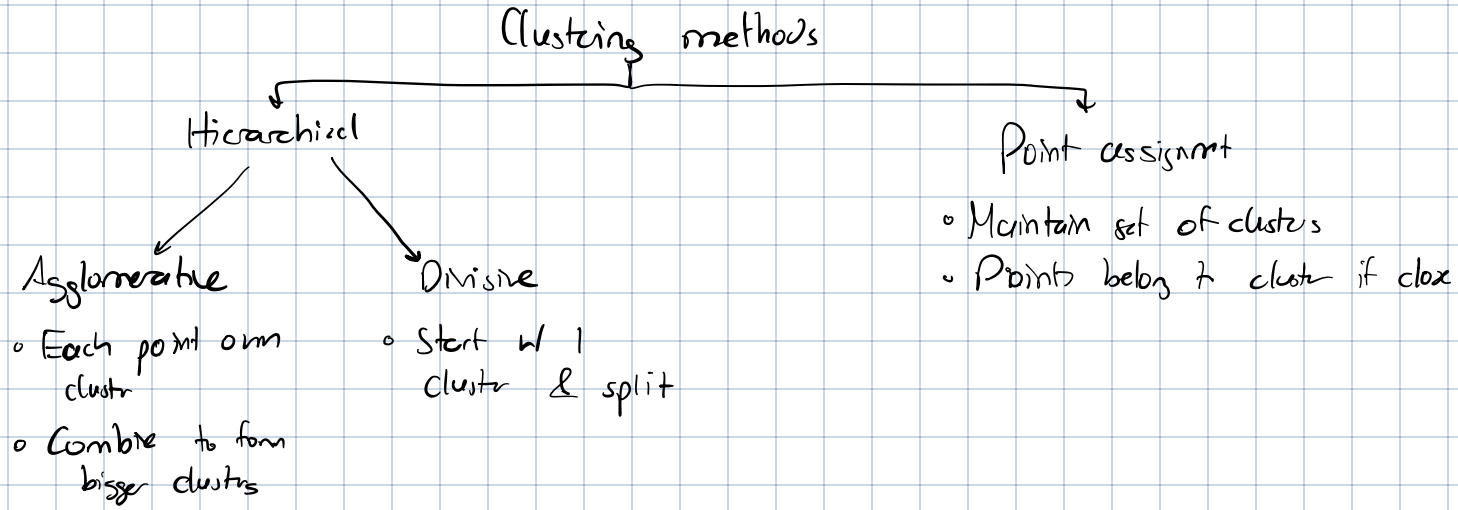
④ Tune R, B for false +ve & false -ve rate

⑤ Hash each of B bands for each doc to find pairs

Clustering

Problem: given set of points, group into clusters

↳ Difficult b/c points often defined in high-dimensional spaces



① Hierarchical clustering

1) How to represent a cluster of points?

In Euclidean case, cluster has a centroid (avg. of data points)

If not Euclidean, choose clustroid: point closest to other points

2) How to measure nearness of clusters.

Distance b/w centroids for Euclidean case.

For non-Euclidean:

1) Intercluster distance: min (distance b/w any 2 points from each cluster)

2) Cohesion

A: Diameter of merged cluster

B: Avg. distance b/w points in cluster

C: Density of cluster

② K-means clustering

Assume Euclidean space & pick k to be # of clusters

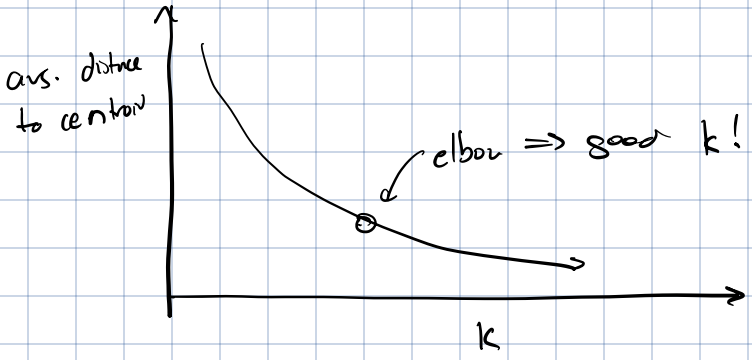
Initialize k clusters by picking k points.

Also:

1. For each point, place in cluster whose centroid it is closest to
2. Update centroid locations for all clusters.
3. Reassign points

Repeat until convergence

How to select k?



Map Reduce implementation

```
def setup:
    clusters = loadClusters()

def map(id, vector):
    emit(clusters.findNearest(vector), vector)

def reduce(clusterId, values: Iterable[Vector]):
    for (vector ← values):
        sum += vector
        cnt += 1
    emit(clusterId, sum/cnt)
```

average distance per cluster

Spark implementation

Faster b/c
1. All data in mem.
2. Points that don't change cluster they don't go to another worker

1. Pick random centroids
2. Create RDD w (cluster #, data point) → partition by key & cache
3. Reduce by key & map → new centroids
4. Create new RDD by reassigning centroids.
   Track # of points that changed via accumulator

If accumulator > 0.

Relational Data

DB workloads

OLTP

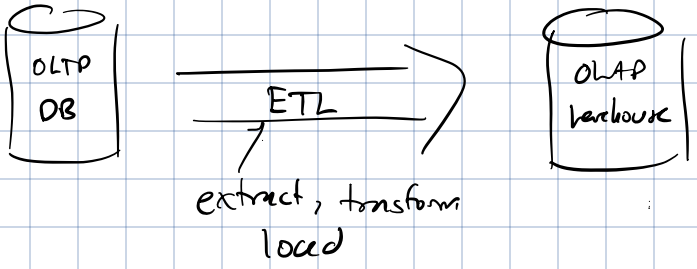
- Most apps use OLTP
- Use facing: fast, concurrent
- Usually small set of common queries
- Random reads & small writes.

OLAP

- This is for BI & data mining
- Batch workloads, low concurrency
- Complex, ad-hoc analytics.
- Full table scans, big data

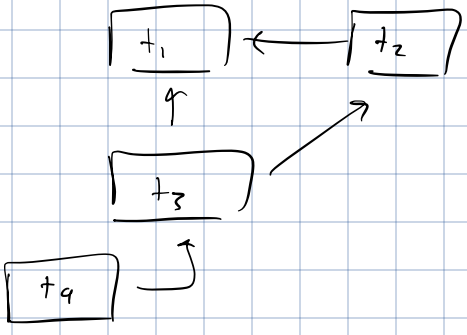
1 DB doing both causes a lot of problems

Soln: Data warehouse



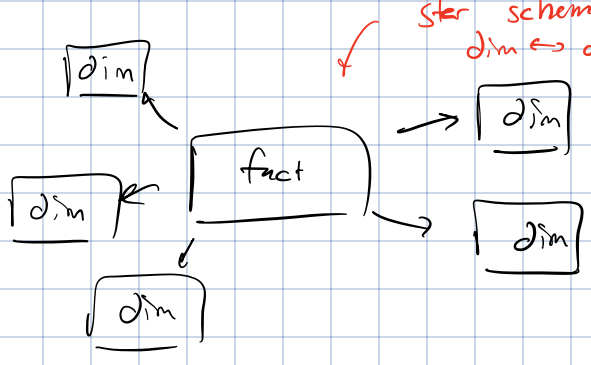
Schemas will be diff:

OLTP:



Captures relations b/w objects.

OLAP:



Fact tables: have unique info per row
Dim tables: foreign keys to fact table that is not unique to a single row.

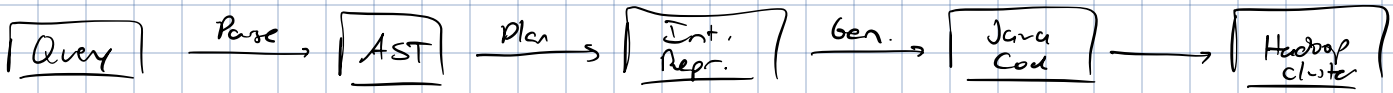
Ex: Sale of widget → fact table (color w/ price, item Id, ...)
Store sold, manufacturer, ... ⇒ dim table

At large scale, OLAP warehouse can just be Hadoop. Analysts use Hive for SQL → Hadoop

Data lake: schema-less dump of data w/ mix of raw & curated data

Data lake —ETL→ warehouse

Q: How does a query get executed on big data warehouses?



RDBMS will do query → AST → execute (no Java int. rep.)

Relational Algebra

① Selection (σ)

Equivalent to SQL: SELECT ... WHERE ...

MapReduce implementation: map-side filter

↳ B/c it is a map operation, it can be pipelined!

↳ Bottleneck will be I/O of loading from text file

② Projection (π)

Equivalent to SQL: SELECT col1, ..., coln ...

MapReduce impl: map-side filter

↳ Need to remember column mappings after selection (eg. col4 → col1)

③ Rename (ρ)

SQL: select col1 α, ...

Not important in MapReduce b/c colnames don't matter

④ Union ($\cup$)

SQL: (select ...) union (select ...)

MapReduce: use MultipleInputFile class → 1 Mapper class / input file

⑤ Difference (-)

SQL: (select ...) minus (select ...)

MapReduce:

1. MultipleInputFile class

2. Each mapper emits: ((data, mapperId), ...)
3. If $S_1 - S_2$, sort S_2 tuples b/f equal S_1 tuples
4. Reducer: Remember last S_2 tuple & emit only S_1 tuples not equal S_2 tuple

(a_1, S_1) → emitted by reducer
(a_2, S_1) → emitted by reducer
(a_3, S_2) → remembered by reducer
(a_3, S_1) } ⇒ not emitted
(a_3, S_1) }

⑥ Cartesian product (x)

SQL: select table1..., table2... from table1, table2

MapReduce: cross joins are extremely expensive → lots of data

- If it is unavoidable, Hadoop will cross file partitions & custom reader will emit tuples to mapper

⑦ Aggregation:

SQL: COUNT, MIN, MAX, SUM ...

MapReduce: do it on reduce si

Relational joins

3 types of joins: inner (or 1-to-1), one-to-many, many-to-many

Joining on MapReduce:

① Hash Join (also known as broadcast, sidebacked, replicated)

Assumption: t_1 is small enough to fit on single rack of memory

Process:

1. Every mapper loads t_1 as hash table
2. Each mapper joins it's partition of t_2 against t_1
3. Map emit

} Fast!

② Map side join

If files sorted by key & split in same partitions, can join row by row rather than crossing entire files

Exactly like cross join, but you should have the sort & partition cond.

③ Reduce-side join (shuffle / repartition join)

Mappers send join key + data → reducers get tuples in join key sort order so reducer joins

Join types:

A: 1-1 joins ⇒ 1 key per t_i, so easy to join

B: 1-many join ⇒ hold 1 tuple in reducer memory from t_1 (ensure it comes 1st through key ordering) & then join w/ t_2 tuples

↳ Key: (join key, origin) (origin order ⇒ $t_1 < t_2$)

C: Many-many joins: have to hold all tuples in memory from t_1 → memory problems.

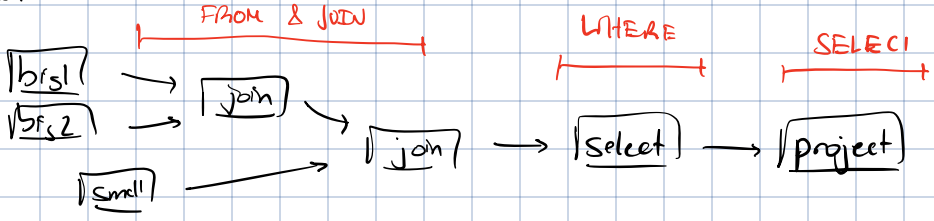
SQL to Hadoop

Know how to do this!

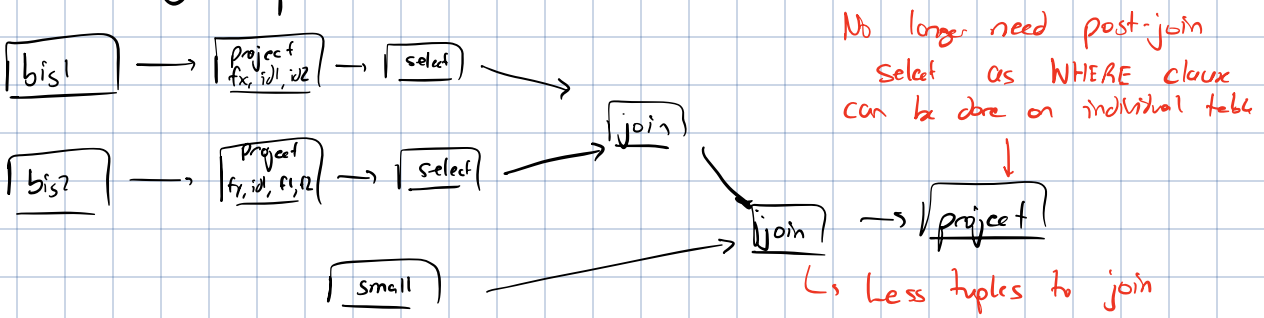
Ex// Convert SQL to Hadoop:

```
select  big1.fx, big2.fy, small.fz
from    big1
join    big2  on  big1.id1 = big2.id1
join    small on  big1.id2 = small.id1
where   big1.fx = 2015  and
        big2.f1 < 40    and
        big2.f2 > 2
```

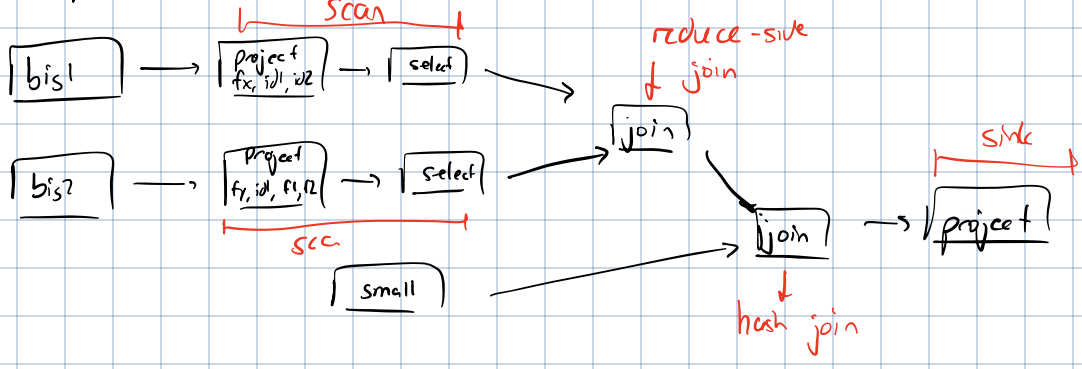
① Logical plan

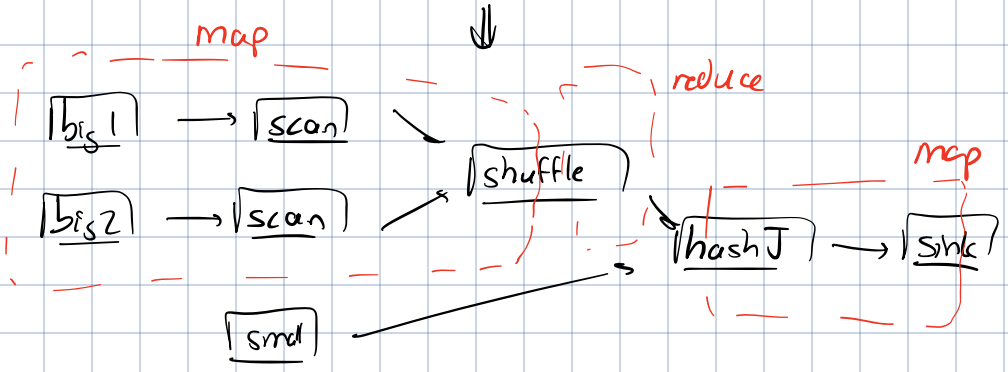


② Optimize logical plan



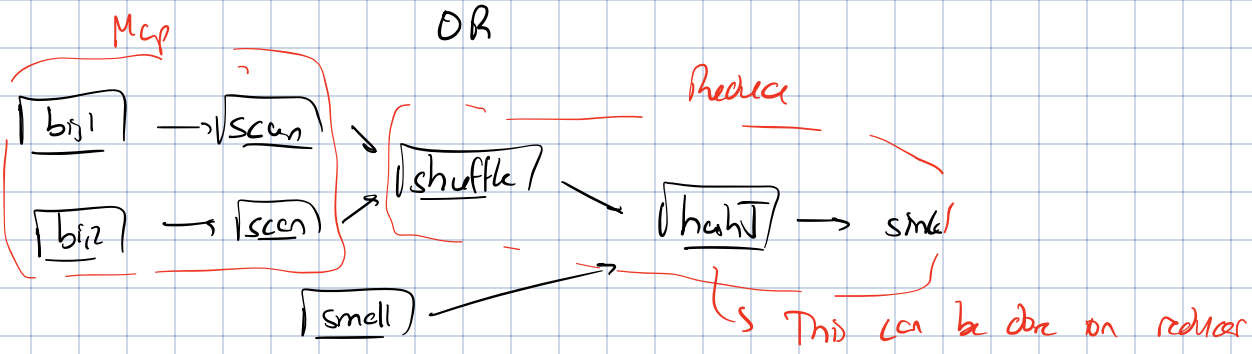
③ Physical plan





2 MapReduce jobs

OR



1 MapReduce job

We know table schemas from metadata

SQL → Spark

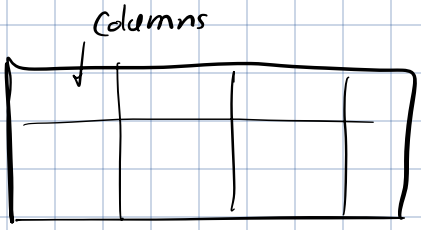
Most relational operators are already RDD transformations

Join:

- Spark will automatically choose map-side or reduce-side depending if it's a narrow dependency (no shuffling) or wide dependency (shuffling needed)
- If hash join needed, use a broadcast var

Spark also has Spark SQL which will transform queries into Spark operations

- Uses dataframes as base data struct instead of RDD.

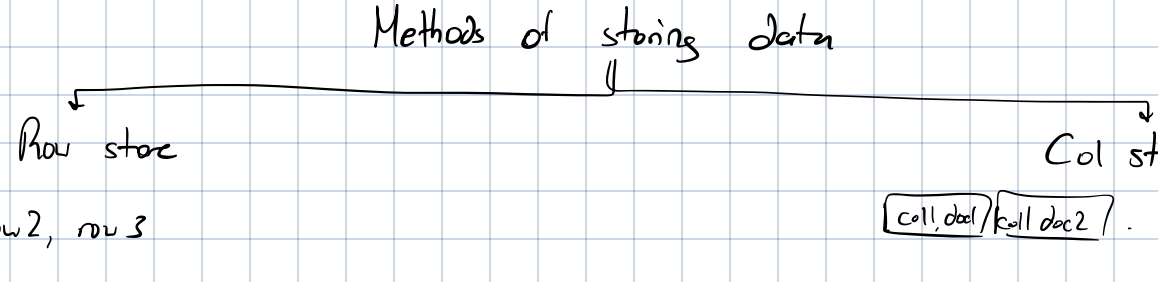


⇒ Equality comparisons: ===

- RDD → DF: `rdd.toDF("colName", "col2Name", ...)` ← Can also pass in column schema
- To query: `df.createOrReplaceTempView("tableName")`
  `spark.sql(sqlQueryUsingTableName)`

Row vs. Column Storage

Reading in text is slow. Convert to binary format & include a schema to convert binary-coded columns to logical columns.



row1, row2, row3
↓
col1 col2 ... coln

- Easier to modify record
- Requires reading a lot of data for processing

Col values usually repeated!

- Only read necessary data
- Writes are more complicated
- Can compress data
- Vector executor & compiler queries work well

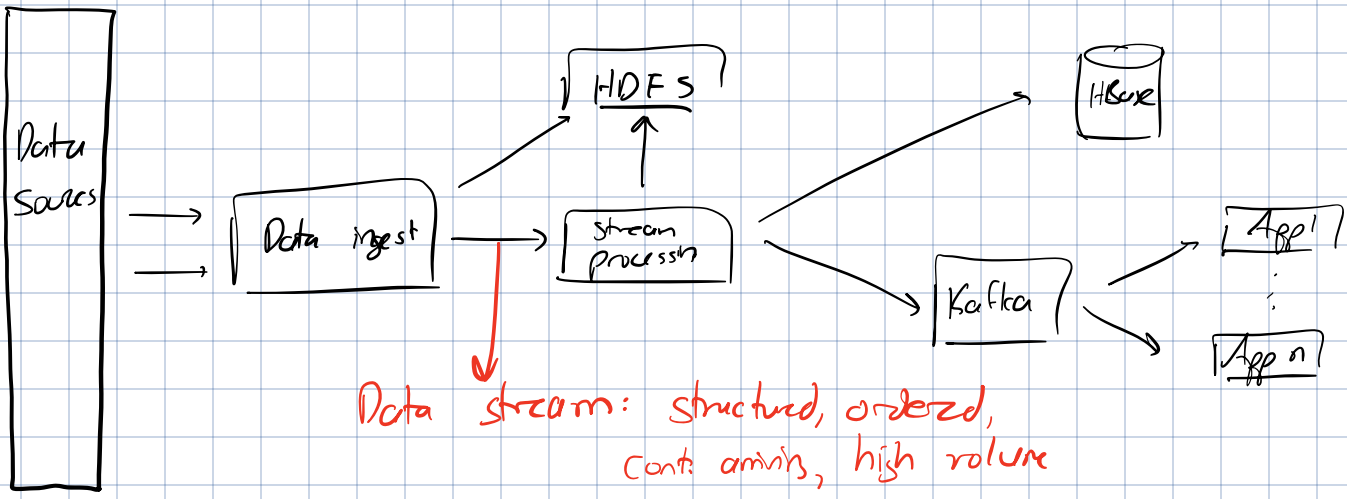
How can we use binary input in Hadoop? Parquet!

- Input: ParquetInputFormat
- Output: ParquetOutputFormat

STREAMING

Stream processing: process data as it arrives, like real-time

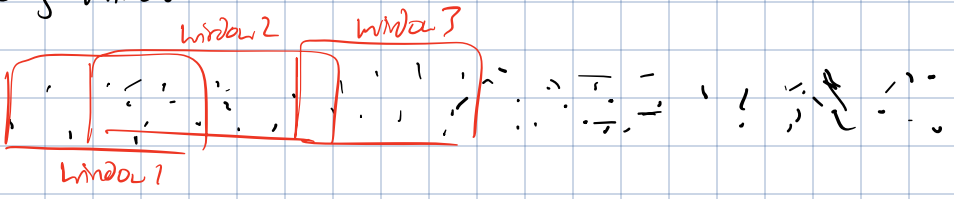
How does it fit in the modern data arch:



Q: How do you do reduce-ops on continous data?

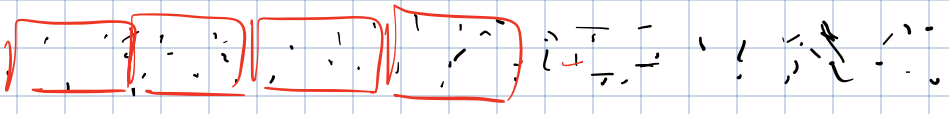
A: Define windows (10-minute, 1-hour, ...)

① Sliding windows



A window of x minutes update y seconds would update y seconds to most current timestamp & have x minutes of data before y

② Tumbling windows



No data in common b/w windows

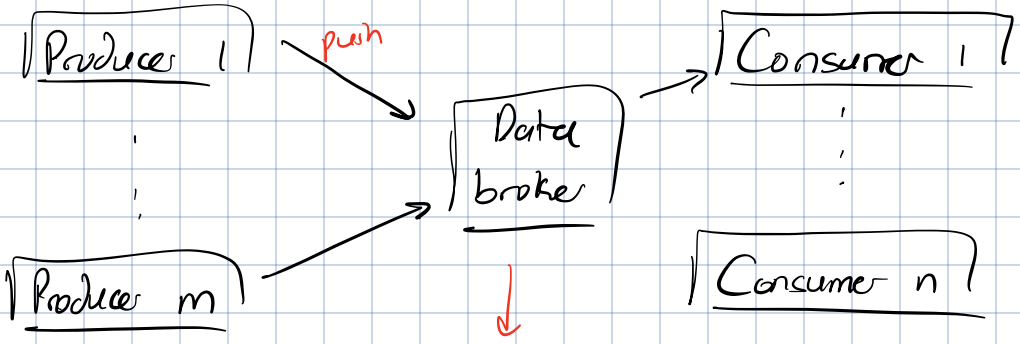
Windows can be based on:

1) Ordering attribute: time
2) Counts: window has n data points b/f new window
3) Punctuation: ingest apps define window end, but it window size becomes unpredictable

Stream processing challenges:

- Latency requirements
- Memory storage
- Burstiness
- Load balancing
- Data consistency
  - ↳ Exactly once
  - ↳ At least once (dupes possible)

Stream processing often works on pub-sub:



Broker can either push to consumers or consumer pulls from broker (polling is wasteful)

Spark Streaming

Tumbling window of extremely small duration → turn into RDD

Needs ssc (Spark streaming context) to perform opps on RDDs (seq. known as DStream)

Can process 6 GB/sec, higher throughput than Storm

Data Brokers

Kafka is very widely used for brokering. MQTT is for IoT uses

Kafka is fast (2M writes/sec). Why?

① Writes almost always go to OS caches

② Reads are fast b/c fast transfer from cache to network socket

Kafka topics are partitioned & replicated for fault tolerance

↳ Messages are hashed & balanced to diff. partitions

We want idempotence, so UUIDs attached to each message. Broke rejects messages if UUID seen b/f.

By keying a message, you guarantee that all msgs w/ same key get to same partition, which is ordered.

Multiple consumers can read a single partition & each consumer remembers how much it read

↳ Consumers can be grouped to read from single topic, but each consumer reads 1 partition

High Volume Streams

① Sample data randomly: select S values randomly

1) Reservoir sampling: store 1st S values, then store k^{th} w/ prob. S/k, discard already stored

↳ Theoretical result: $k \geq s \cdot \forall i \in [1, k]$, $P(V_i \text{ in reservoir}) = \frac{S}{k}$

Multiset Ops

Q: How do we know how many items in set

A: HyperLogLog counter

Observation: hash(item) → vector of 32 bits

↳ 1/2 of items have code w/ 0., 1/4 w/ 00...

We can record longest string of prefix 0s & estimate 2^x

Q: How do we know if an item is in a set

A: Bloom filter

Create a vector of m bits & k unique hashing func.

put function: hash x & flip bits of k hashes in m-vector

contains function: " & check if all of K hashes is 1

No false negatives, but sometimes false trues (can be tuned)

Q: How do I count frequency of x in set

A: Count-min sketch

Like Bloom filter, but each hash func. has own m-vector bit vec.

Increment each vector hash elem

To get # of times x seen, take minimum of vector elems at hash

$$\min \begin{Bmatrix} A[h_1(x)] \\ \vdots \\ A[h_4(x)] \end{Bmatrix} = \dots$$

MUTABLE STATE

Problems w/ RDBMS:

① How to scale out ← relations b/w tables

② Doesn't support semi structured

Alternative: NoSQL

↳ Features:

① Horizontal scaling

② Replicated & distributed data

③ Not ACID (eg. eventual consistency)

④ Schemas flexible

Follows BASE:

① Basically available: should be avail. most times

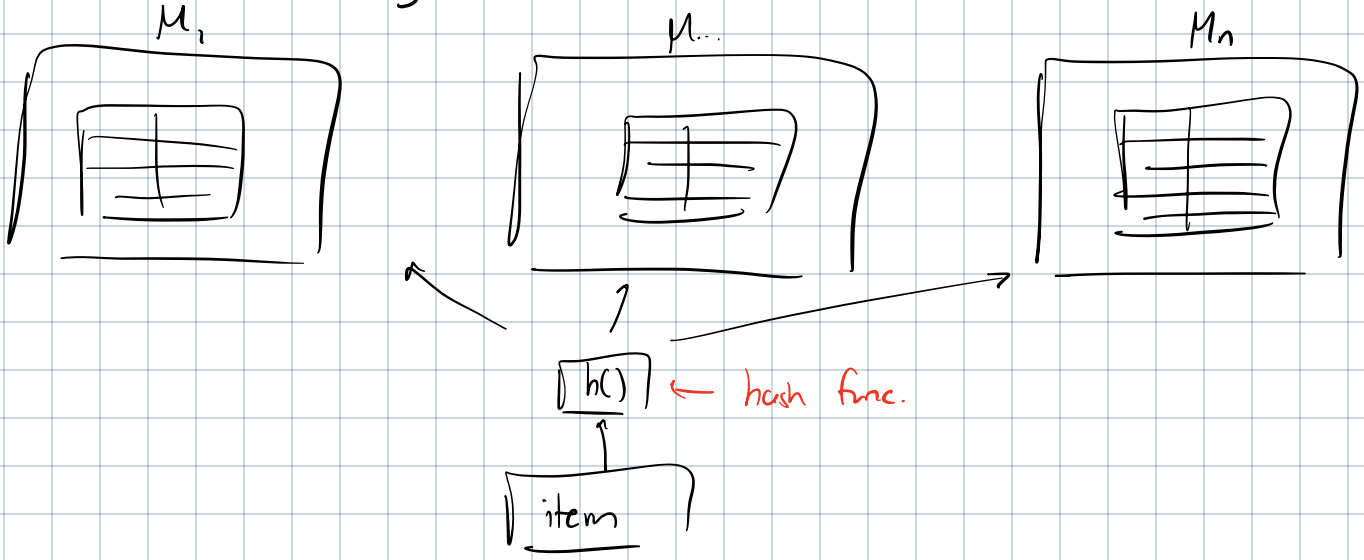
② Soft state: state may change even w/out input

③ Eventually consistent: updates may cause inconsistency, but will work out

Types of DBs: KV stores, column stores, document-based, graph

Key Value Stores

Goal: distributed hashing



Consistent hashing:

① Assign each node a GUID from hash func. h

② For key k, compute h(k) ← same hash func.

③ sucessor(k): first GUID $\geq h(k)$

Why is this approach good? Each node has at most $\frac{(1+\varepsilon)\cdot k}{n}$ keys.

↳ If node goes down, only remap $O(k/n)$ keys

How to make successor(k) fast? Chord distributed protocol

① Naive sol'n: link nodes in doubly linked circle & check all nodes until cond. matched

$O(n)$ worst case

② Finger table: each node holds predecessor, successor & finger table:

successor	node
h+2	1
h+4	1
h+8	1
⋮	⋮

} Finger table

Also:

1) Find h(k) → go to node

2) In node, look at finger table & go as far as you can that exceeds $h(k)$

3) Repeat until successor found.

$O(\log n)$

To handle new nodes, we have to do following:

1) Create new finger table for new node

2) Update other node finger tables: background process

Assumption: peer-to-peer nodes, most systems are not like this.

Memcached: distributed KV store, no replication of data

↳ Acts as a cache (no persistence)

↳ Every node in cluster acts independently.

↳ hash func: $h(k) = k \% NUM\_SERVERS$

Redis: can handle more than just KV

↳ Distributed: nodes can communicate → replication → redundancy

↳ Persistent

↳ Hashing: $bucket(k) = h(x) \% 16384$ → cluster coordinate which bucket belongs to which node.

Buckets can be added/reassigned to diff. nodes based on load/node health

Document Stores

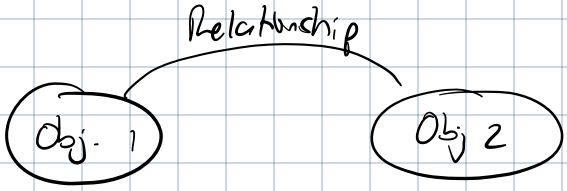
Idea: key ⟷ {. . . .} (JSON-like doc.)

Many systems use multi-value concurrency control for eventual consistency

key : $\boxed{t_1 : doc1}$ → $\boxed{t_2 : doc2}$ → . . .

- Since it is a linked list, reads & writes can be concurrent!
  - Read: look at latest committed version
  - Write: append to linked list

Graph DBs



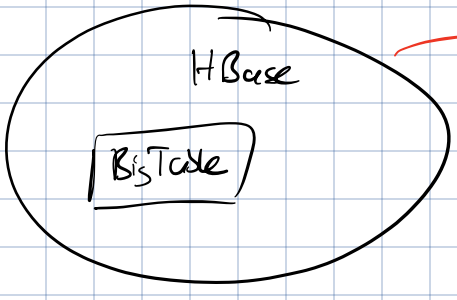
Used if you have a lot of relations (eg. social media)

Common tech: Neo4j

Column Stores

Google BigTable: (row key, column key, timestamp) ↔ <bytes stream>

↳ Bloom filters to check if r_{key}, c_{key} exist



→ Like MapReduce to Hadoop

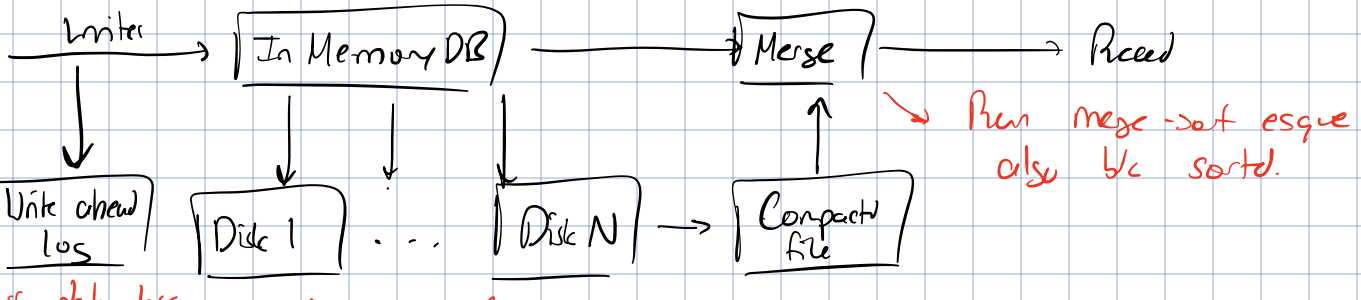
Backed by HDFS

Rows are maintained in sorted lexicographical order → efficient row scans

Columns are grouped into families (C_{key}: $C_{family} + C_{qualifier}$)

ID within family

Underlying data structure: log structured merge tree (LSMT)

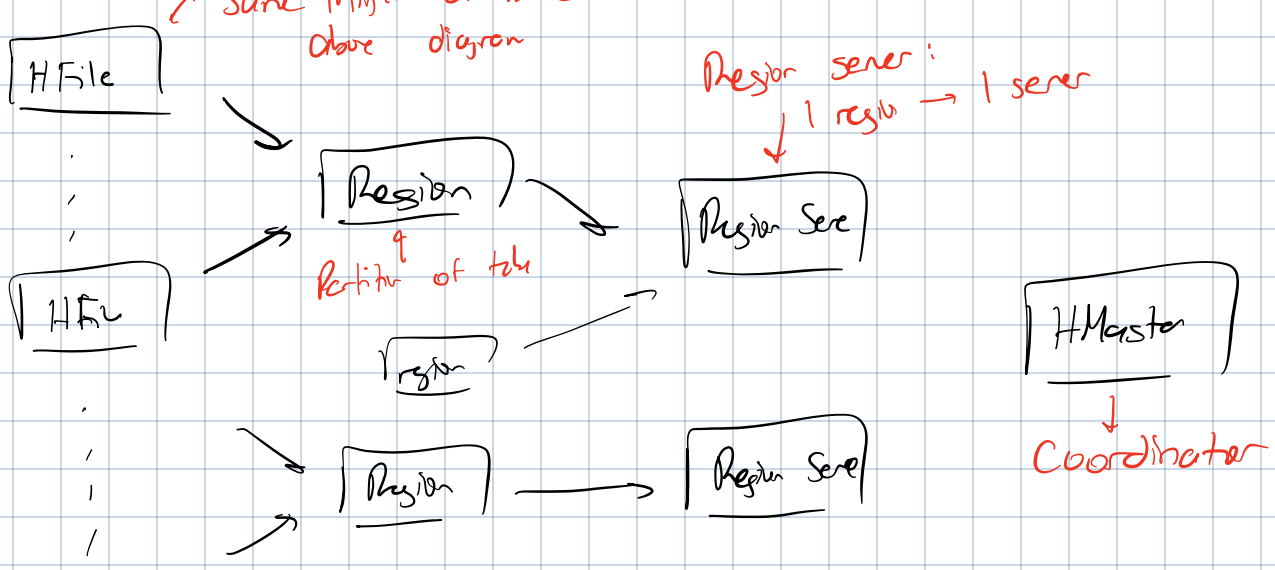


Run merge-sort esque algo b/c sorted.

1) Replay if data loss
2) Can send to other replicas

Why not append to 1 disk file? Appending will break sorted order

Architecture:



Consistency

Consistency: all nodes return same value

Availability: function w/out all nodes being available

Partition tolerance: " network being split

CAP theorem: pick 2 from above, cannot have all 3

↳ We want dist. sys. to work during partition tolerance. Pick b/w cons. or avail.

↳ SQL prefers consistency, NoSQL prefers availability

Types of consistency:

① Strong: after update, all subsequent access returns updated value

- 2-phase commit protocol

1) Coordinator makes sure everyone can commit
2) " tells subordinates to commit data
3) " ensures data got committed

} can be call backed

Blocking & slow

- Distributed consensus protocol (eg. Paxos)

  Nodes can be proposers (advocate for data change), acceptors (decide if change should go) and/or learners (remember values)

  Paxos algo:

  1) Proposer selects proposal # n > all prev. proposals
  2) Proposes n to quorum (≥ majority) of nodes
  3) Node, either:
     a) Promise to reject all future proposals $< n$
     b) Reject proposal
  4) If quorum sends promises, then proposer tells everyone to accept

② Weak: after update, some will return new value, other return old

- Eventual: after some time, all nodes return updated value

If no partitions, having tradeoff b/w consistency & latency (high consistency requires time)

HBase always keeps consistency no matter what

- Transactions only allowed to update 1 row
- Since 1 region → 1 region server, it will be consistent