CS4225/CS5425 Big Data Systems for Data Science

MapReduce & Data Mining

Bryan Hooi
School of Computing
National University of Singapore
bhooi@comp.nus.edu.sg

School of Computing

Announcements

- Tutorial 1 will be held today after lecture.
- Assignment 1 will be released this weekend.

Week	Date	Topics	Tutorial	Due Dates
1	18 Aug	Overview and Introduction		
2	25 Aug	MapReduce - Introduction		
3	1 Sep	Polling Day Public Holiday		
4	8 Sep	MapReduce and Data Mining	Tutorial: MapReduce	Assignment 1 released
5	15 Sep	NoSQL Overview 1		
6	22 Sep	NoSQL Overview 2	Tutorial: NoSQL	
Recess				
7	6 Oct	Apache Spark 1		
8	13 Oct	Apache Spark 2	Tutorial: Spark	Assignment 1 due (15 Oct 11.59pm), Assignment 2 released
9	20 Oct	Stream Processing 1		
10	27 Oct	Stream Processing 2	Tutorial: Stream Processing	
11	3 Nov	Large Graph Processing 1		
12	10 Nov	NUS Well-Being Day		
13	17 Nov	Large Graph Processing 2	Tutorial: Graph Processing	Assignment 2 due (19 Nov 11.59pm)
	29 Nov	**Final Exam**		

Check mate: How ICA uses data analysis to conduct targeted checks for contraband

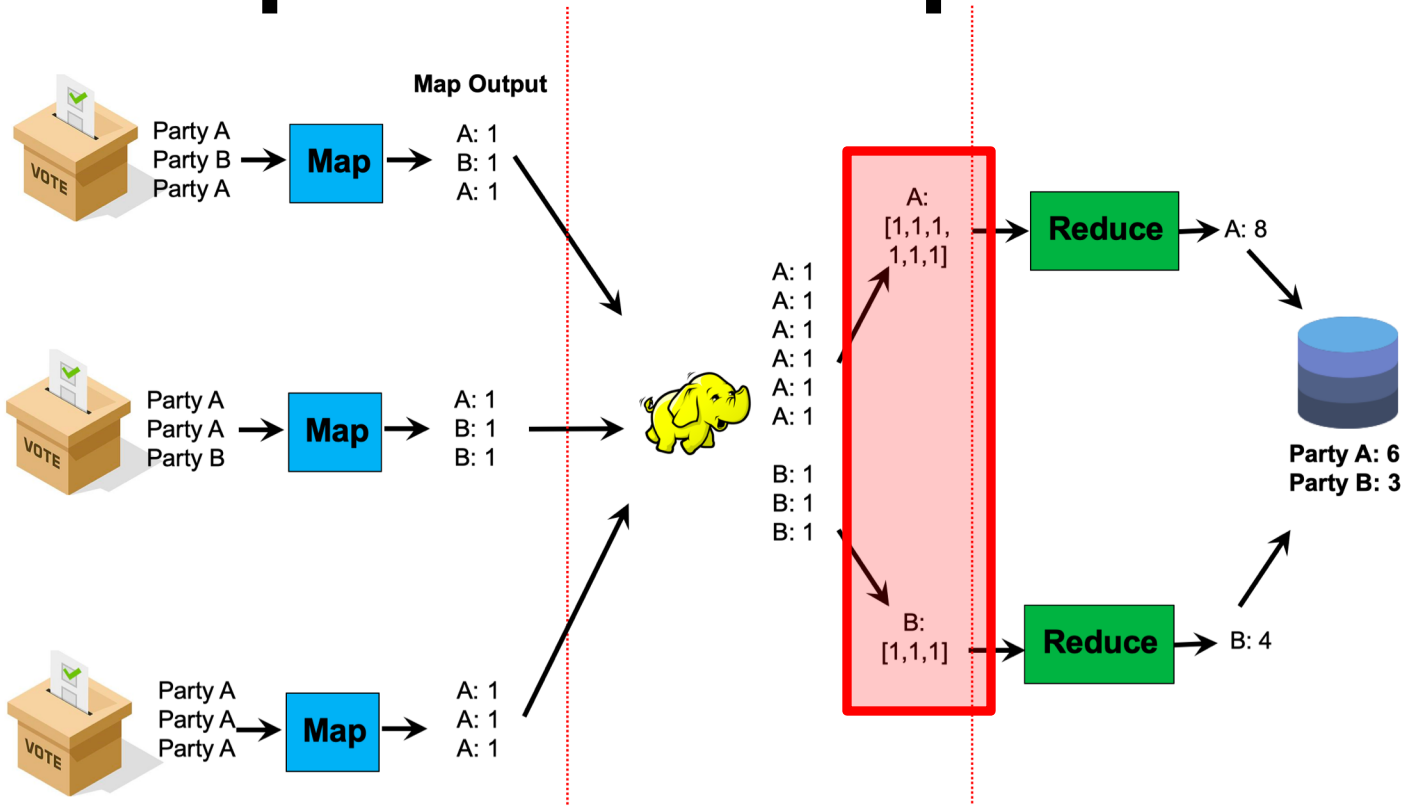
David Sun
Crime Correspondent

PUBLISHED AUG 26, 2023, 5:00 AM SGT

SINGAPORE – Each day, 400,000 travellers pass through the land checkpoints in Woodlands and Tuas.

Most are genuine travellers, but some attempt to sneak contraband and other items illegally into Singapore.

Recap: Partition Step



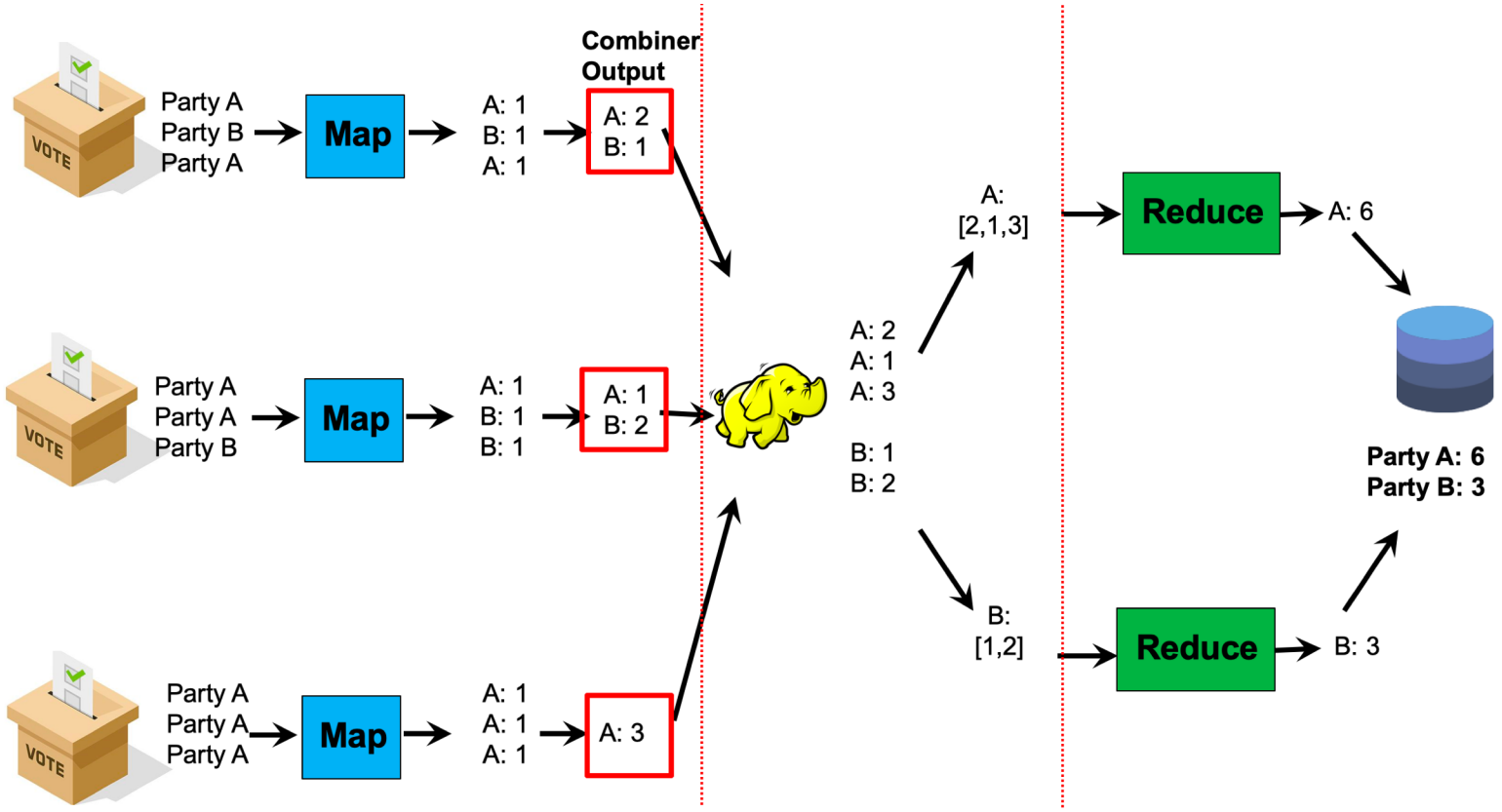
- Note that key A went to reducer 1, and key B went to reducer 2
- By default, the assignment of keys to reducers is determined by a **hash function**

  %

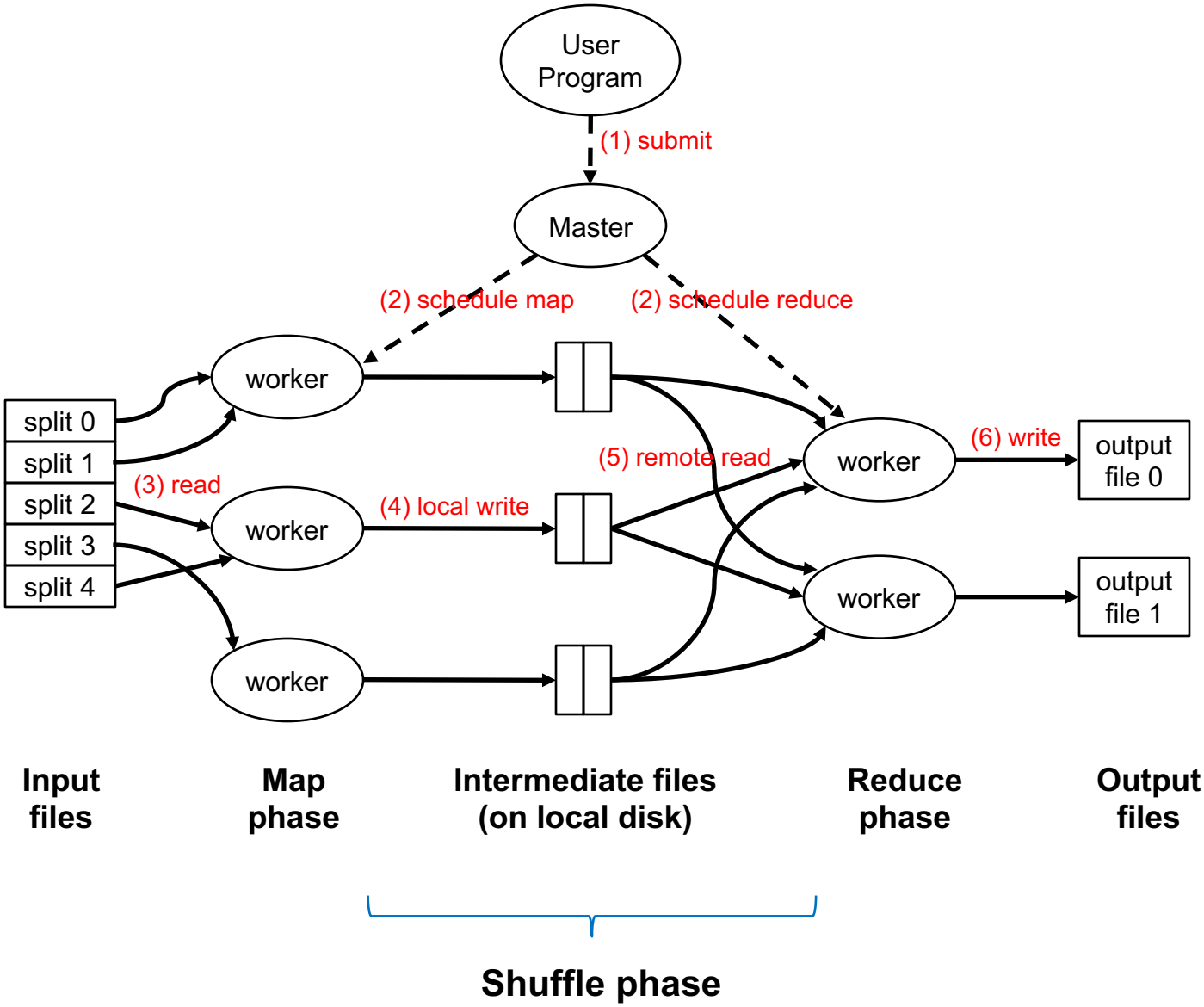
  - e.g., key *k* goes to reducer hash(*k*) mod *num_reducers*
- User can optionally implement a custom partition, e.g. to better spread out the load among reducers (if some keys have much more values than others)

Recap: Combiner

- The user must ensure that the combiner does not affect the correctness of the final output, whether the combiner runs 0, 1, or multiple times

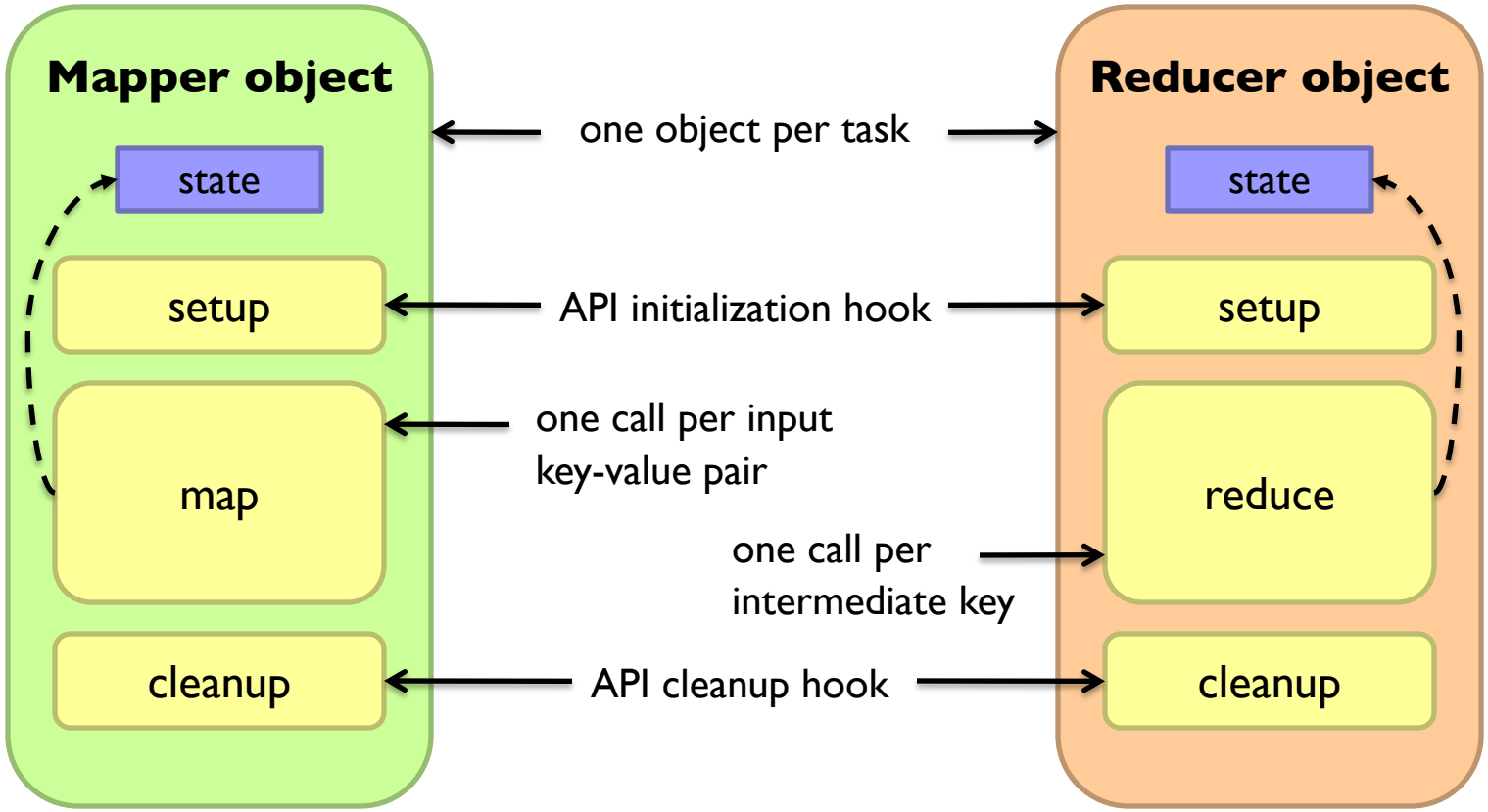


Recap: MapReduce Implementation

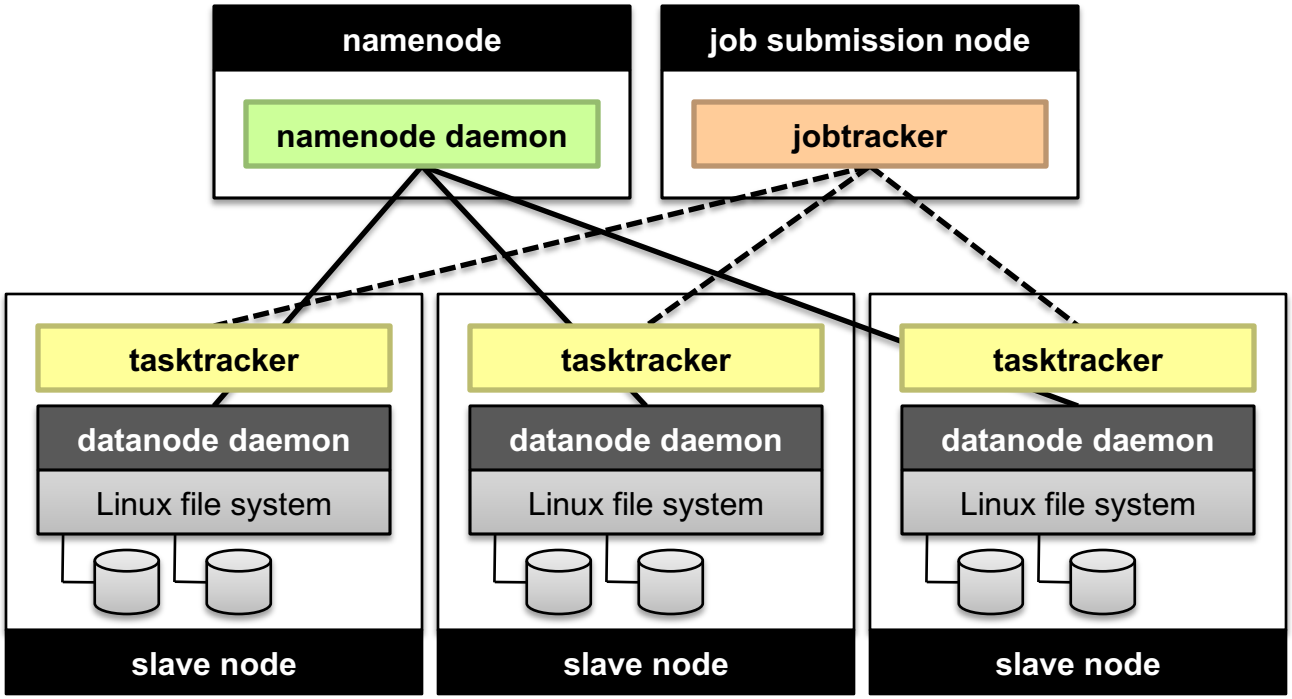


Adapted from (Dean and Ghemawat, OSDI 2004)

Preserving State in Mappers / Reducers

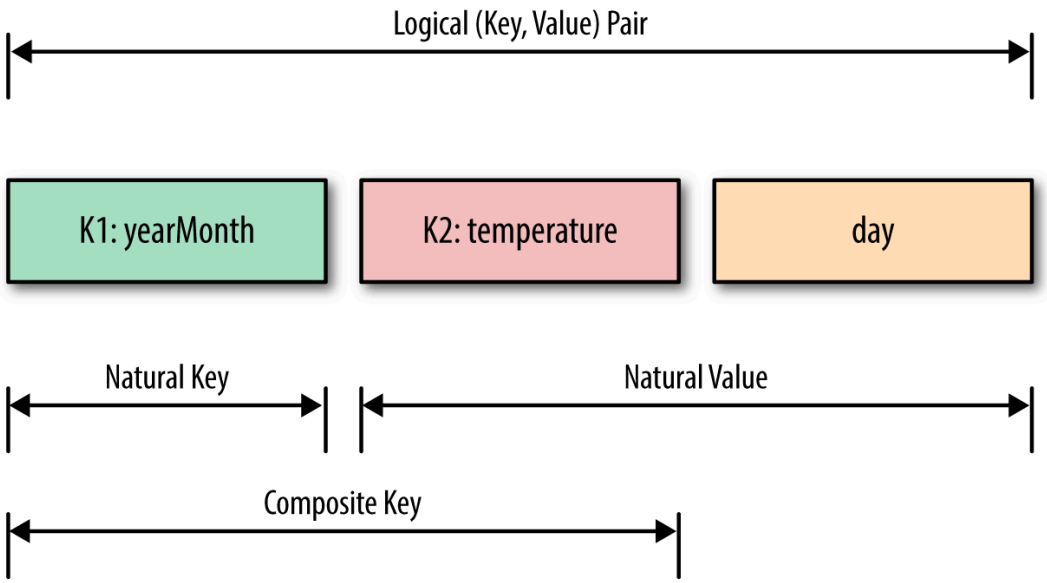


Recap: Combining HDFS and Hadoop



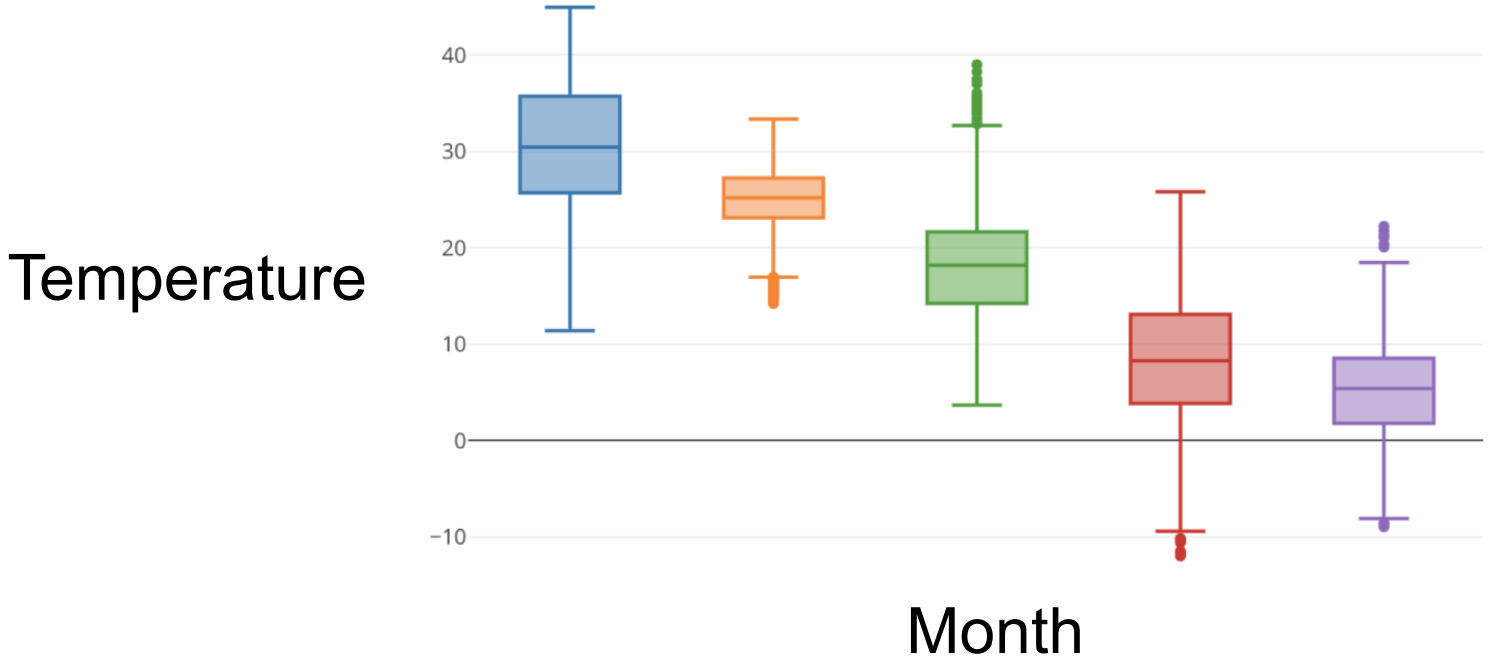
Secondary Sort

- **Problem**: each reducer's values arrive unsorted. But what if we want them to be sorted (e.g. sorted by temperature)?
- **Solution**: define a new 'composite key' as (K1, K2), where K1 is the original key ("Natural Key") and K2 is the variable we want to use to sort
  - **Partitioner**: now needs to be customized, to partition by K1 only, not (K1, K2)



Secondary Sort: Example

Assume we want to compute some statistics (median, 25% quantile, etc.) of the data **grouped by month**.

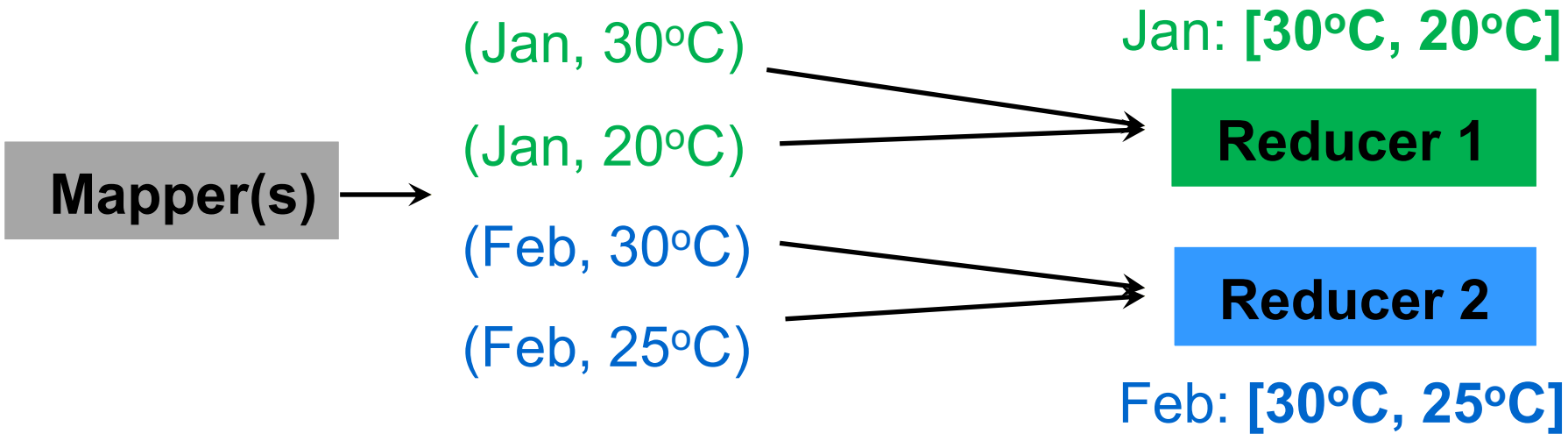


Secondary Sort: Example

Assume we want to compute some statistics (median, 25% quantile, etc.) of the data **grouped by month**.

Our map function emits (month, temperature) tuples, so that tuples from the same month go to the same reducer.

However, the values (temperature) arrive at each reducer **unsorted.**

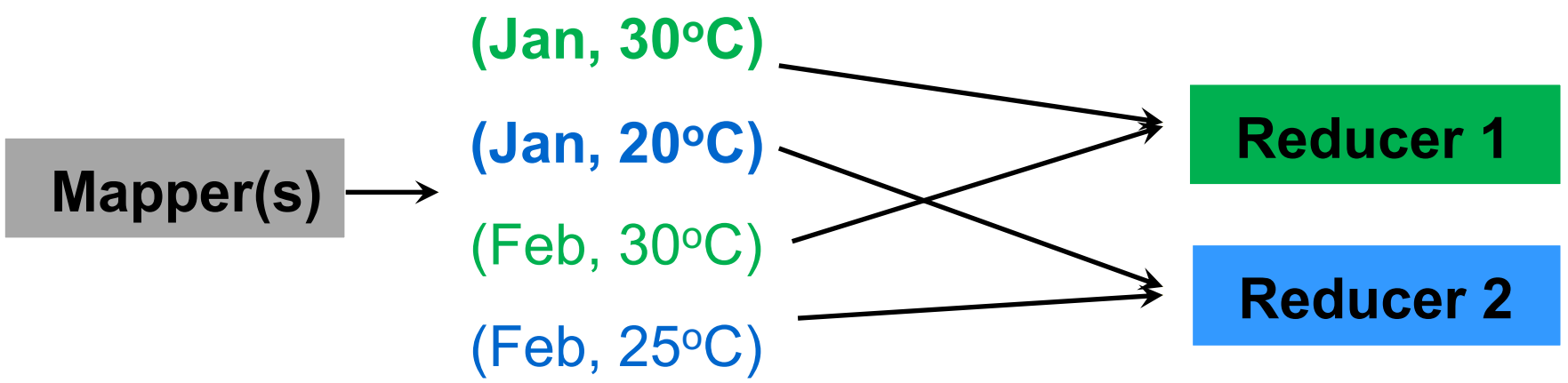


Secondary Sort: Example

Now suppose we use (month, temperature) as **composite key**, but **without changing the partitioner**.

The 4 tuples below have 4 different "values" of the composite key.

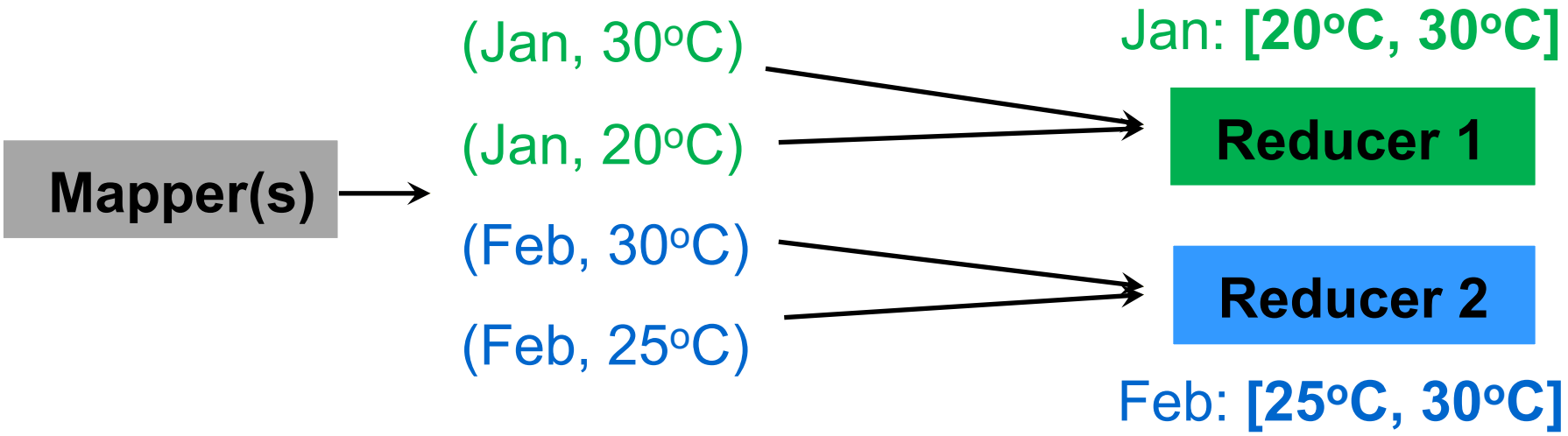
Recall that they will be partitioned by hashing the composite key values. So data with the same primary key may be sent to different reducers, which we don't want (since we can't compute the desired statistics for each month)



Secondary Sort: Example

Finally, secondary sort uses (month, temperature) as composite key, and uses a **custom partitioner, to partition by month only**.

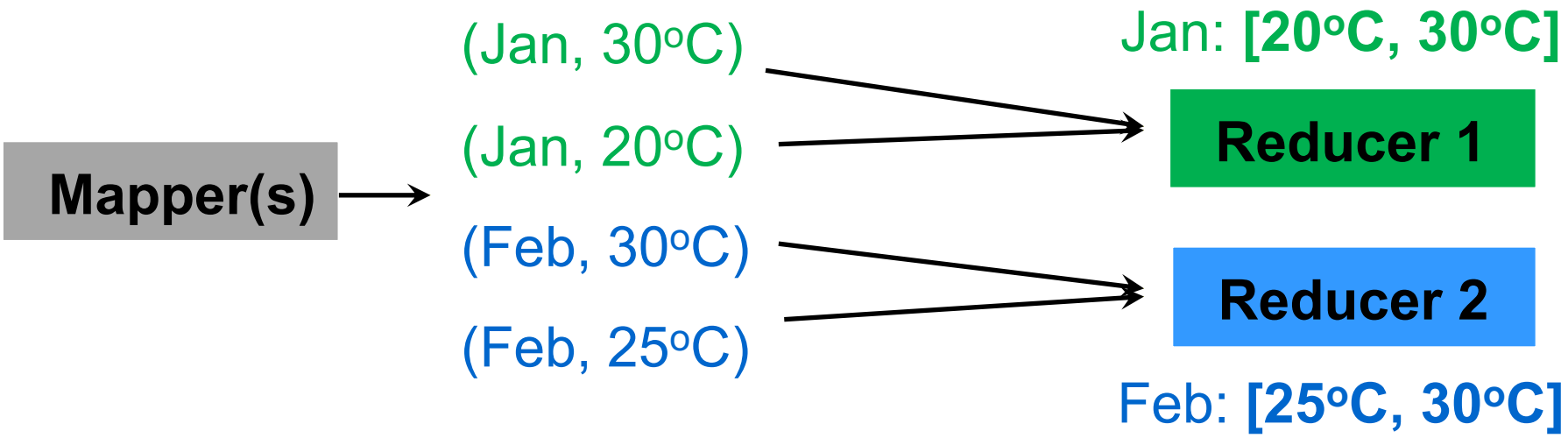
Now we see that 1) data for the same month goes to the same reducer, and 2) at each reducer, data arrives sorted by temperature, since the MapReduce framework always sorts the data by key before giving it to each reducer.



Secondary Sort: Example

Q: Could we just sort the data ourselves in the reducer?

A: Yes, but it may be expensive (in memory & time) if there are a lot of tuples for one key. Secondary sort is effectively "borrowing" the inbuilt distributed sorting process of Hadoop to sort the tuples for us.



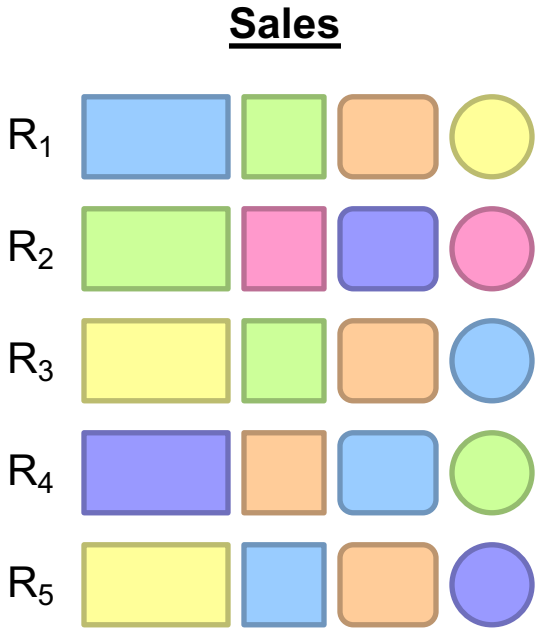
Overview

1. **Relational Databases and MapReduce**
2. Similarity Search
3. Clustering

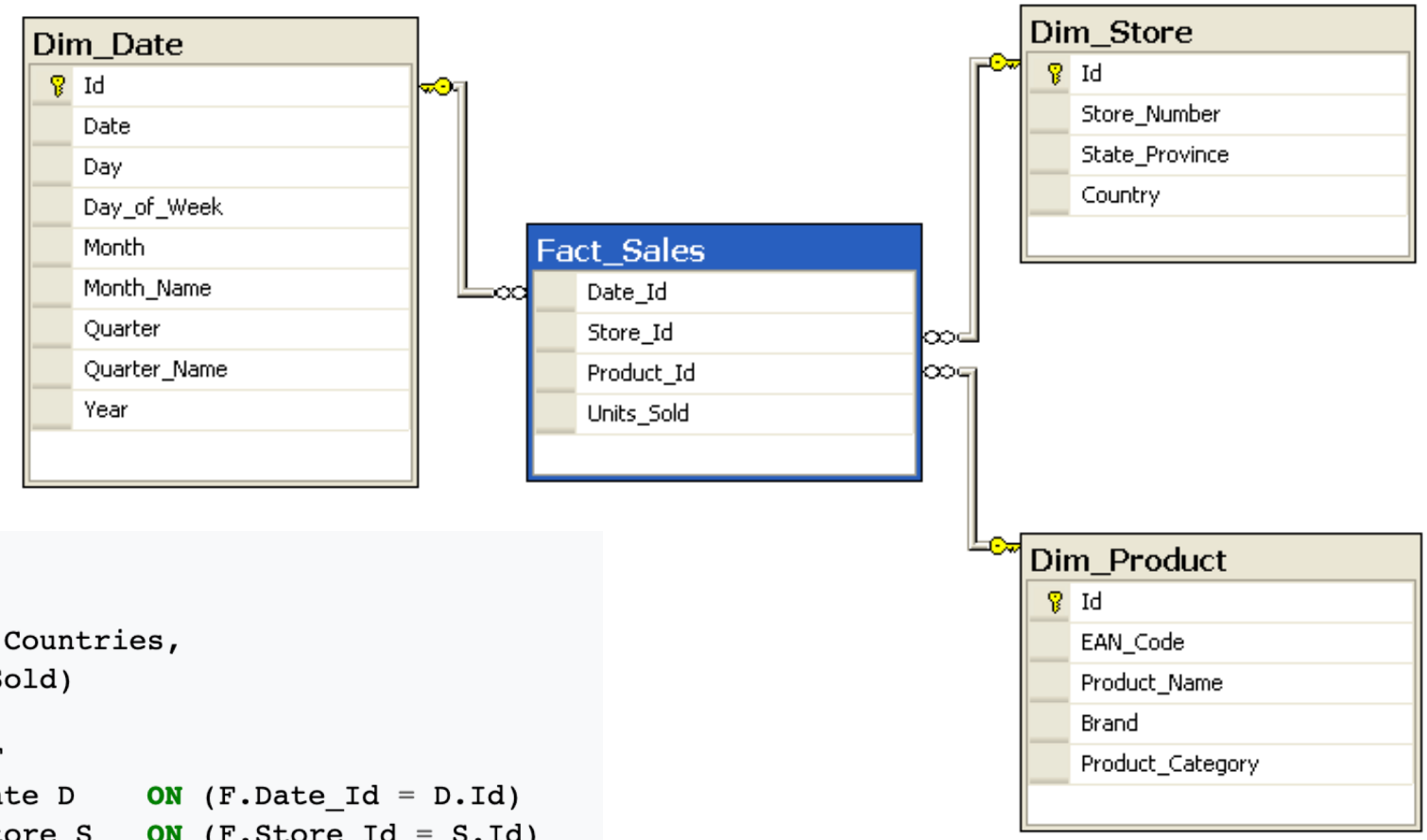
Source: www.flickr.com/photos/rheinitz/6158837748/

Relational Databases

- A relational database is comprised of tables.
- Each table represents a relation = collection of tuples (rows).
- Each tuple consists of multiple fields.



Star Schema and SQL Queries

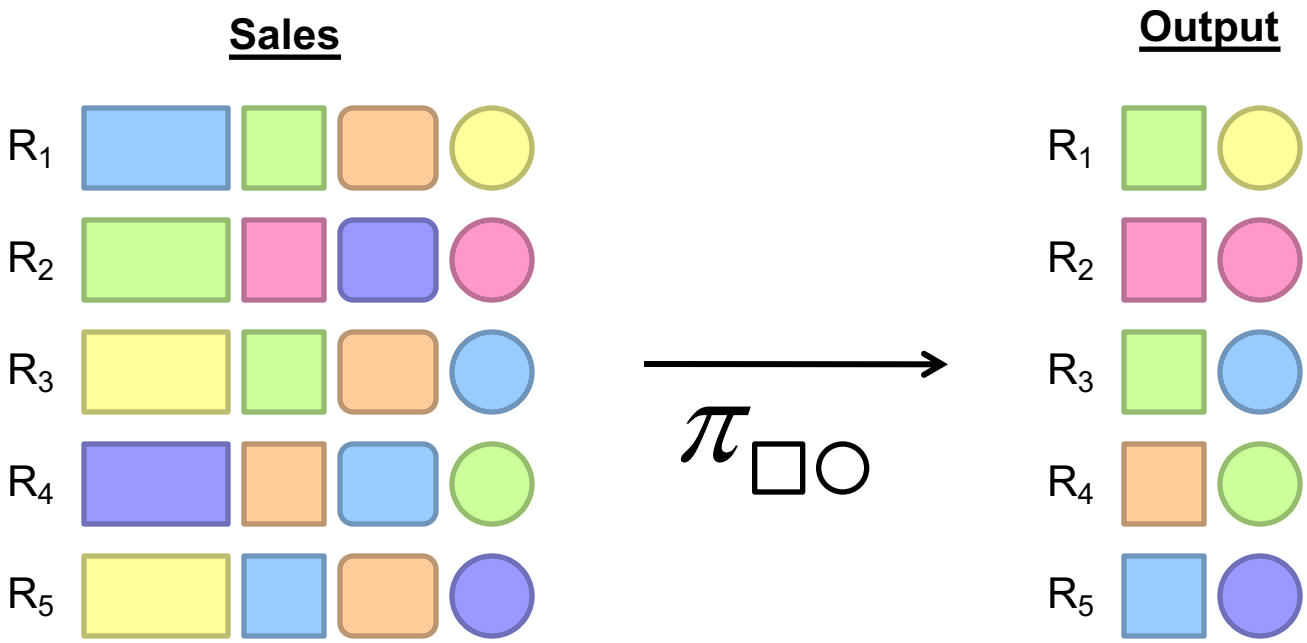



```
SELECT
    P.Brand,
    S.Country AS Countries,
    SUM(F.Units_Sold)

FROM Fact_Sales F
INNER JOIN Dim_Date D    ON (F.Date_Id = D.Id)
INNER JOIN Dim_Store S   ON (F.Store_Id = S.Id)
INNER JOIN Dim_Product P ON (F.Product_Id = P.Id)

WHERE D.Year = 1997 AND  P.Product_Category = 'tv'

GROUP BY
    P.Brand,
    S.Country
```


Source: Wikipedia (Star Schema)

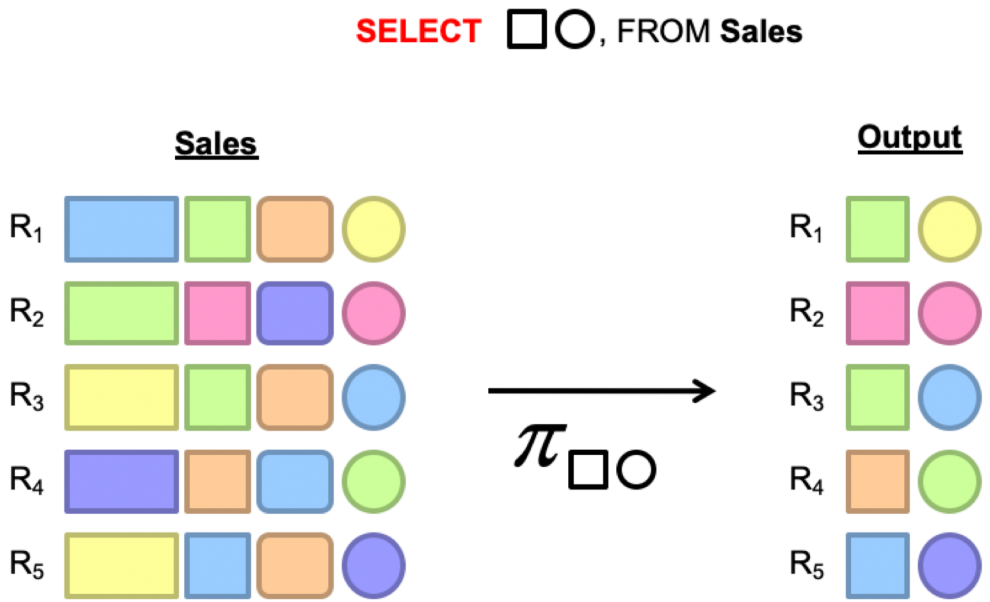
Projection

SELECT □○, FROM **Sales**



Projection in MapReduce

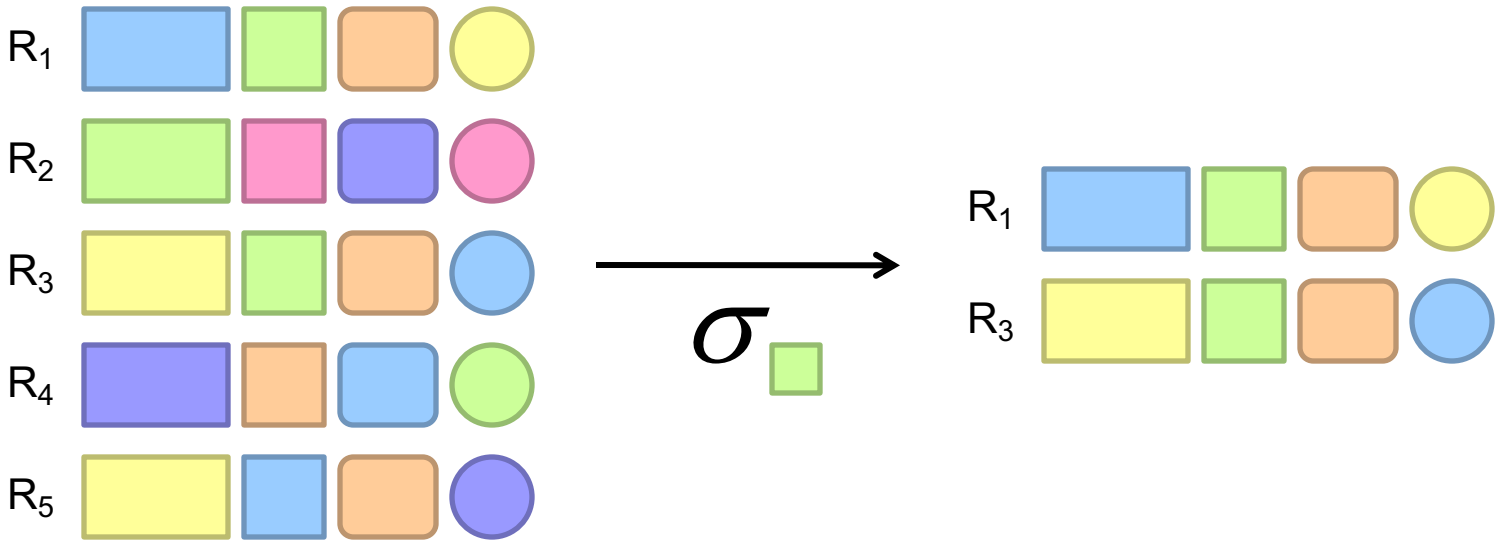
- **Map**: take in a tuple (with tuple ID as key), and emit new tuples with appropriate attributes
- No reducer needed (⇒ no need shuffle step)



Selection

predicate

SELECT * FROM **Sales** WHERE (price > 10)

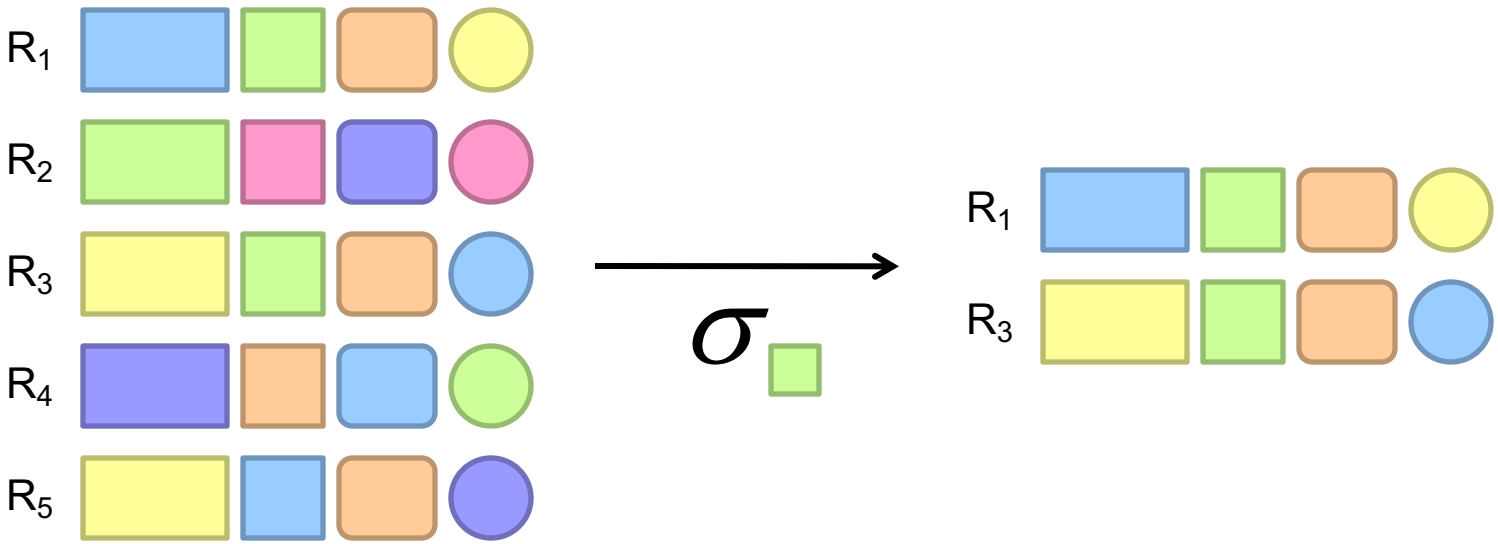


Selection in MapReduce

- **Map**: take in a tuple (with tuple ID as key), and emit only tuples that meet the predicate
- No reducer needed

predicate

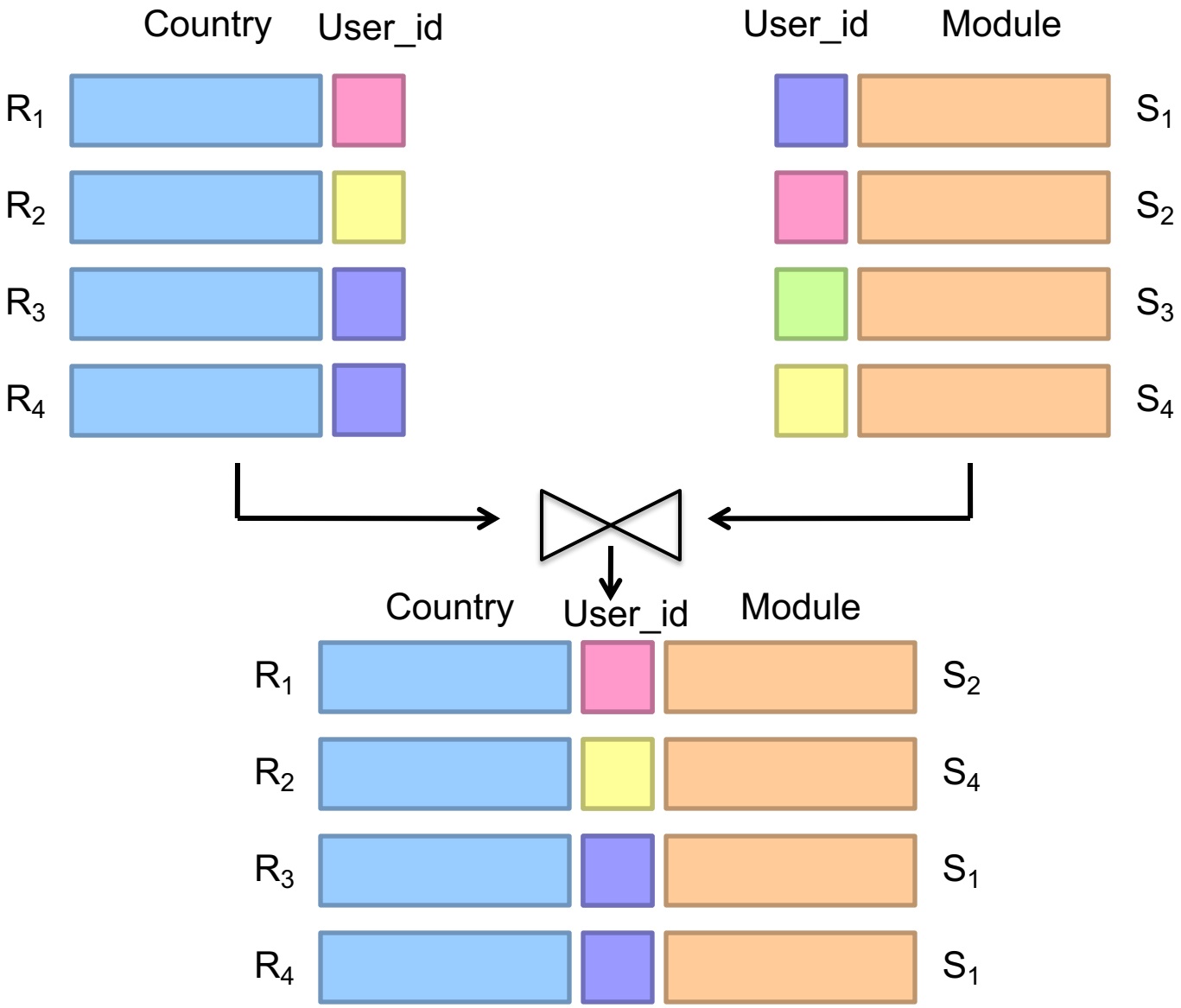
SELECT * FROM **Sales** WHERE (price > 10)



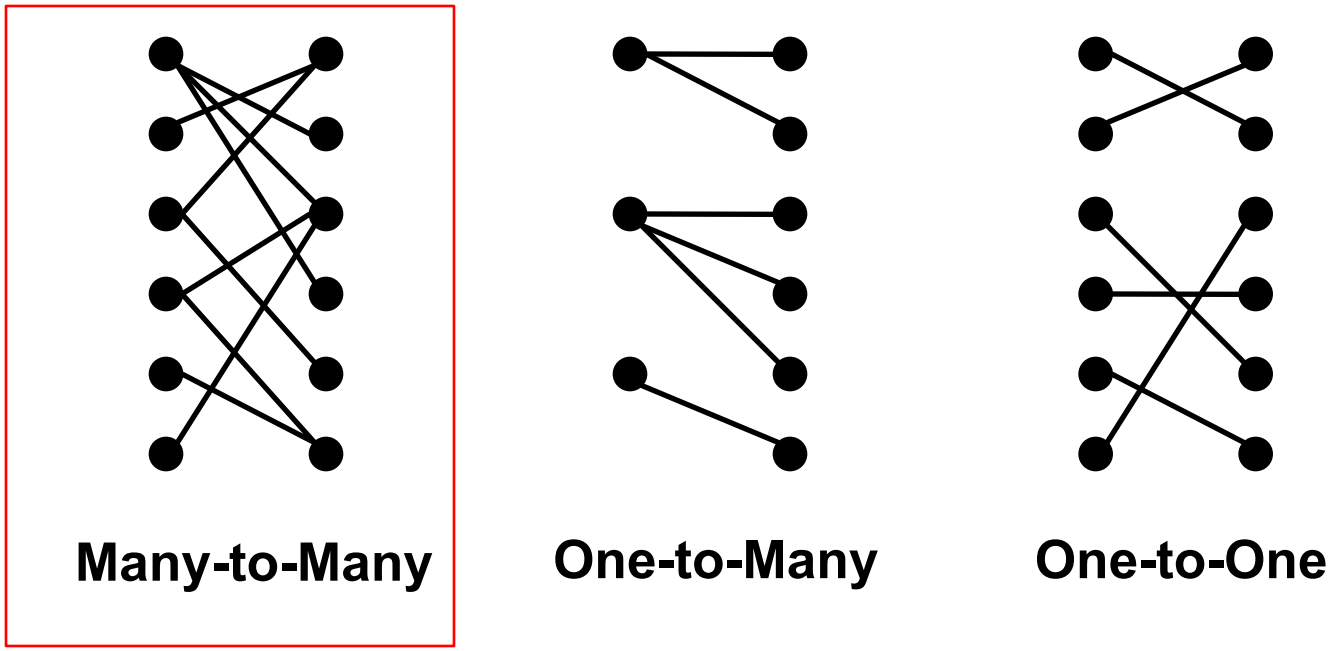
Group by… Aggregation

- Example: What is the average sale price per product?
- In SQL:
  - SELECT product_id, AVG(price) FROM sales GROUP BY product_id
- In MapReduce:
  - Map over tuples, emit <product_id, price>
  - Framework automatically groups these tuples by key
  - Compute average in reducer
  - Optimize with combiners

Relational Joins ('Inner Join')



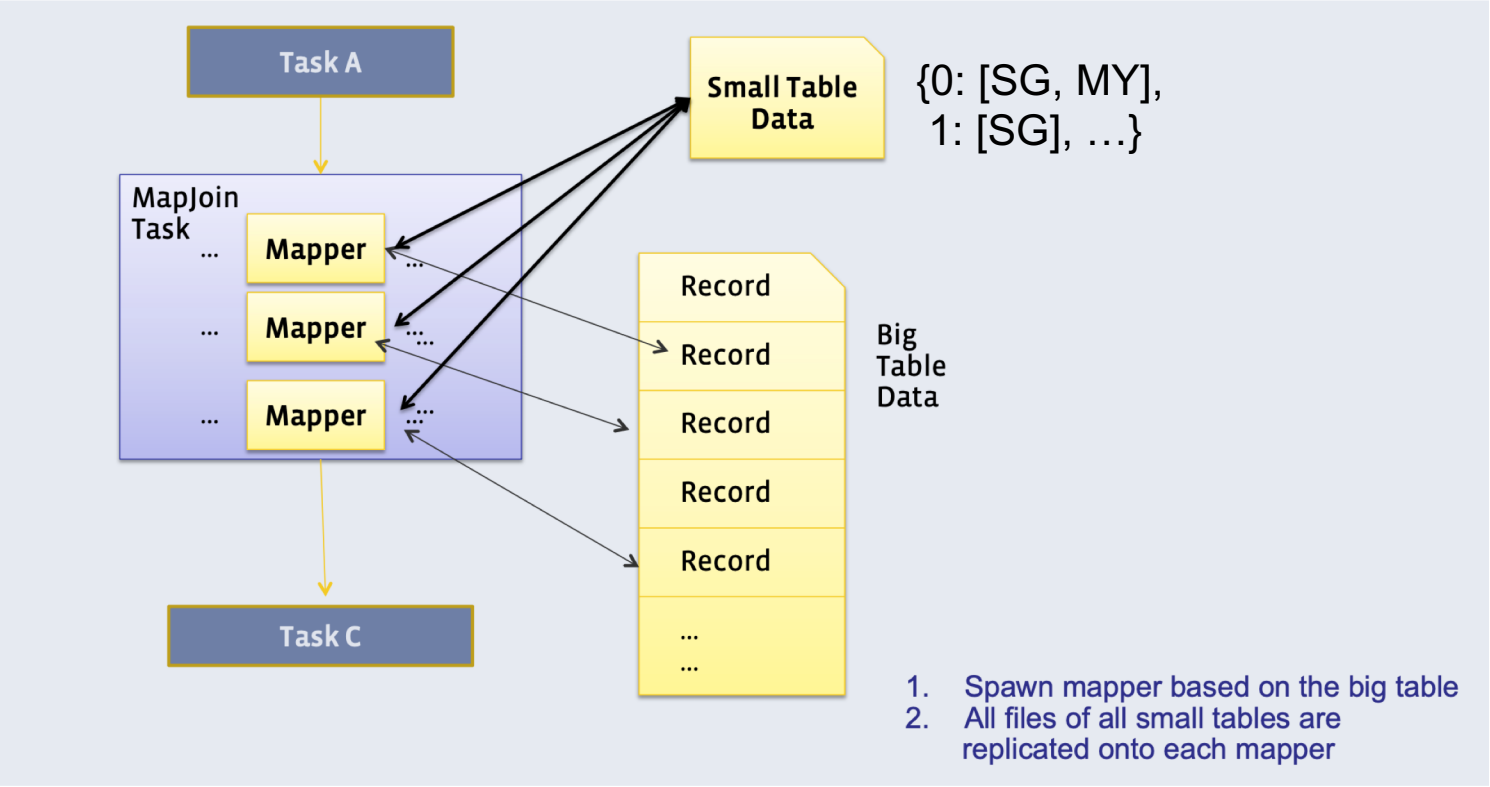
Types of Relationships



- "Many to many" relationship: e.g. a particular user ID can appear multiple times, in both tables

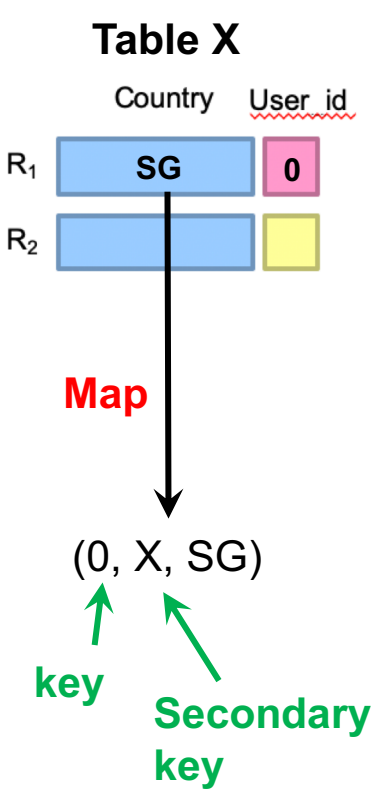
Method 1: Broadcast (or 'Map') Join

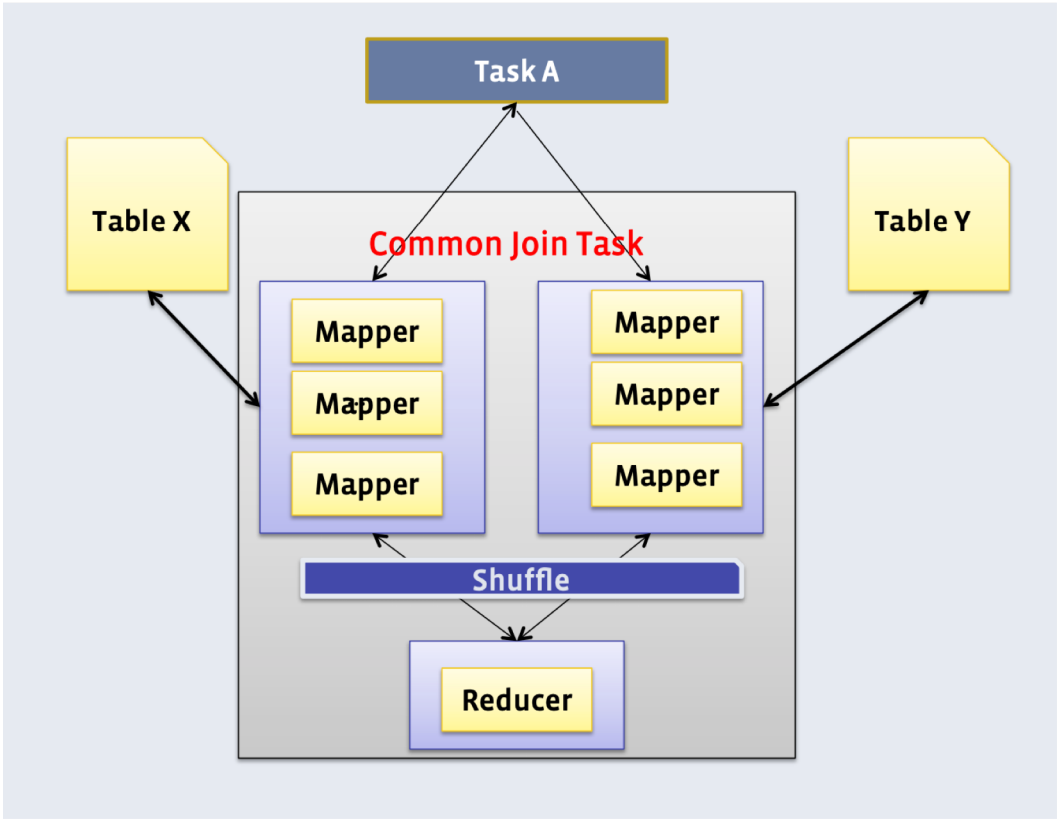
- Requires one of the tables to fit in memory
  - All mappers store a copy of the small table (for efficiency: we convert it to a hash table, with keys as the keys we want to join by)
  - They iterate over the big table, and join the records with the small table

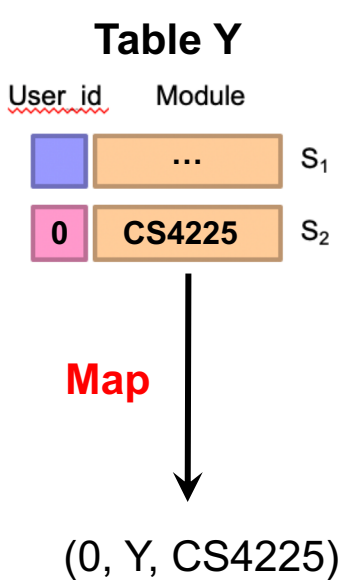


Method 2: Reduce-side (or 'Common') Join

- Doesn't require a dataset to fit in memory, but slower than broadcast join
  - Different mappers operate on each table, and emit records, with key as the variable to join by

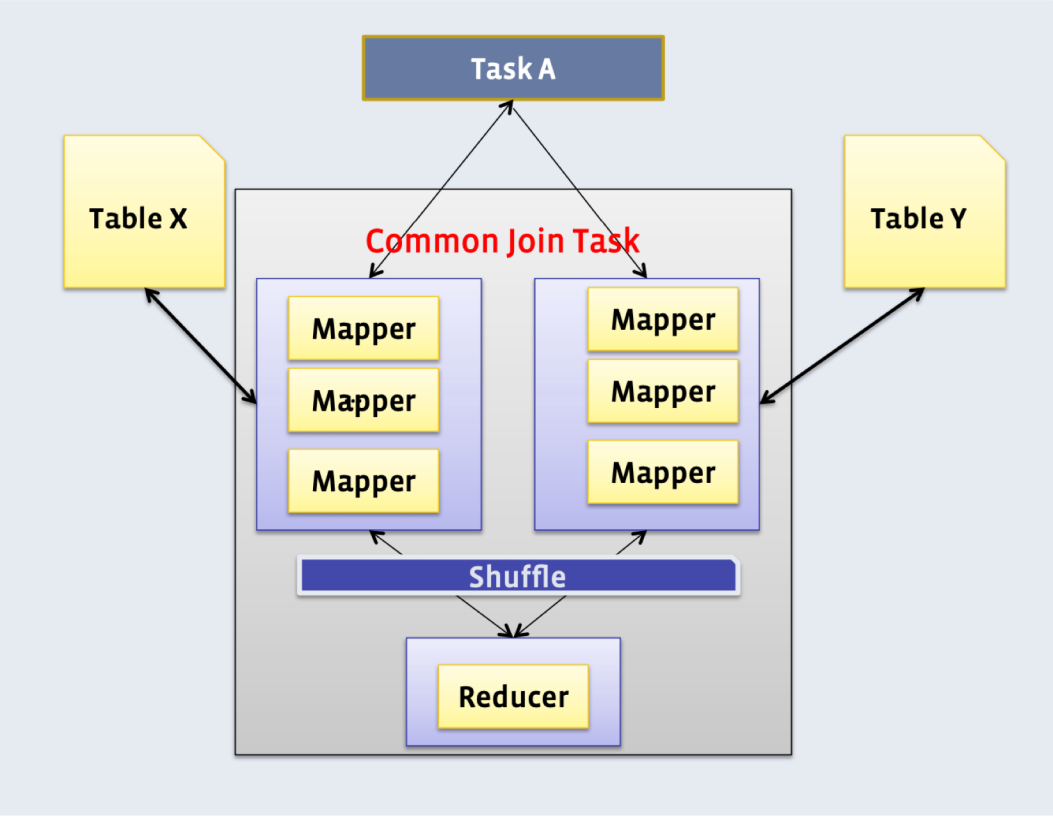




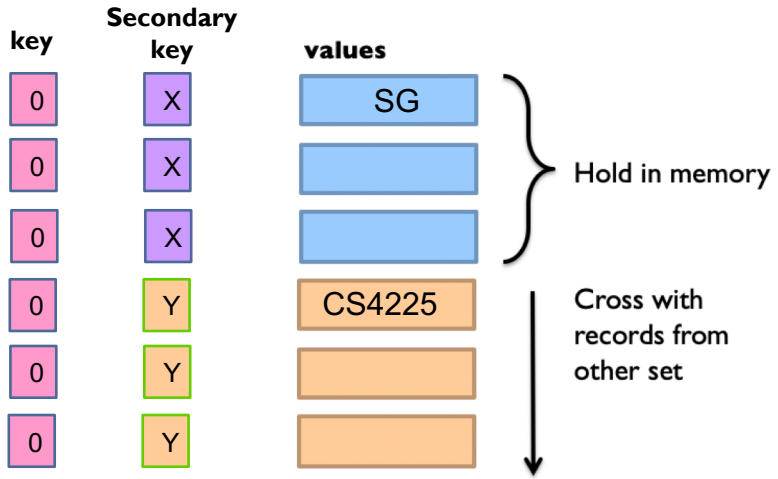


Method 2: Reduce-side ('Common') Join

- In reducer: we can use **secondary sort** to ensure that all keys from table X arrive before table Y
  - Then, hold the keys from table X in memory and cross them with records from table Y



In reduce function for key 0:



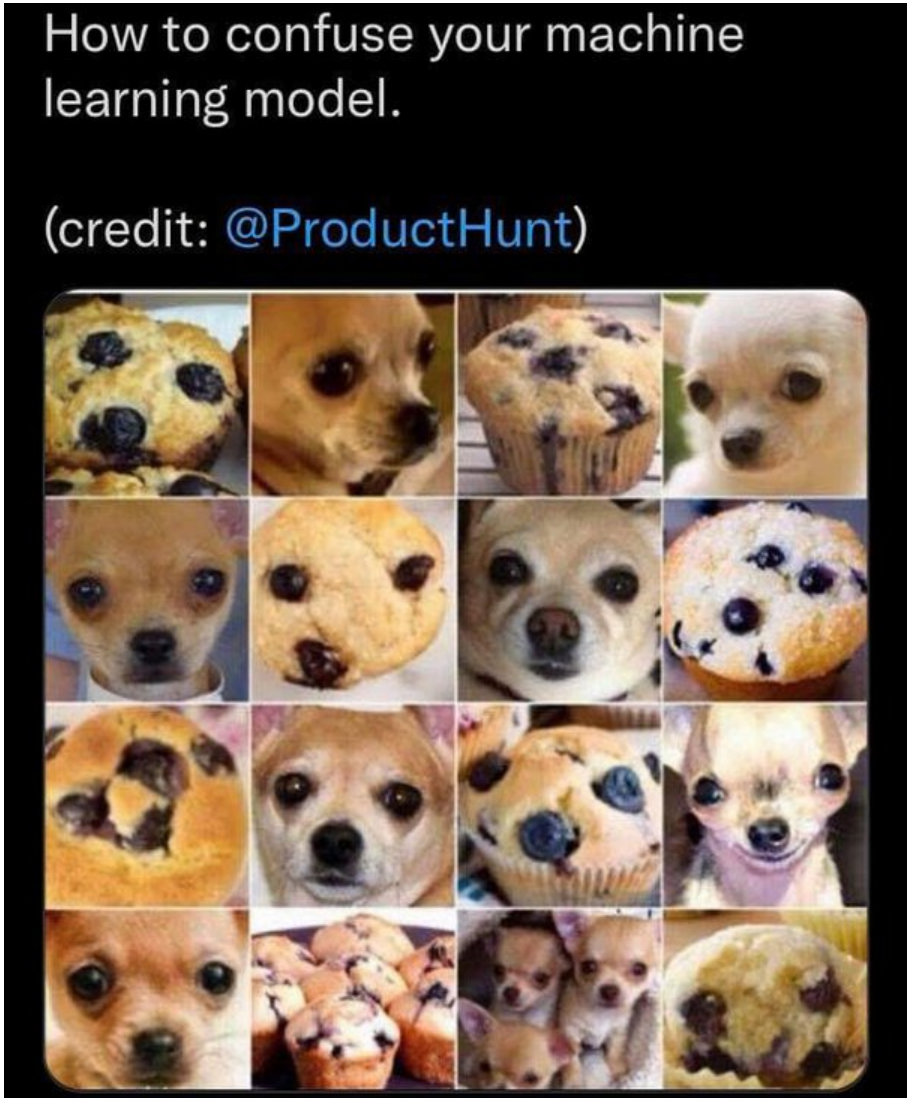
Overview

1. Relational Databases and MapReduce
2. **Similarity Search**
3. Clustering

Source: www.flickr.com/photos/rheinitz/6158837748/

A Common Subroutine

- Many problems can be expressed as finding "similar" objects:
- Examples:
  - **Pages with similar words**
    - For duplicate detection, classification by topic
  - **Customers who purchased similar products**
    - Products with similar customer sets
  - **Users who visited similar websites**
    - Users with similar interests.



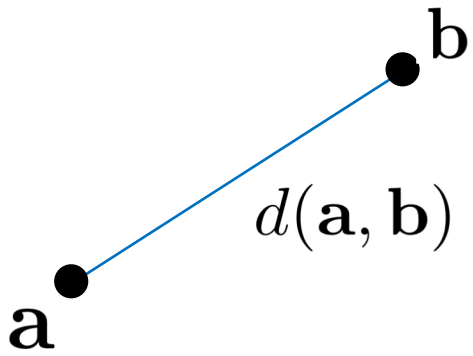
Distance / Similarity Measures

- We define "near neighbors" as points that are a "small distance" apart
- To measure the distance between objects x and y, we need a function d(x, y) which we call a "**distance measure**"
- **Similarity measures** are the opposite: lower distance = higher similarity, and vice versa

Common distance / similarity metrics

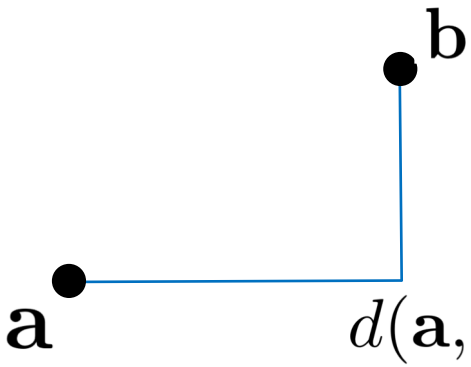
- **Euclidean distance**

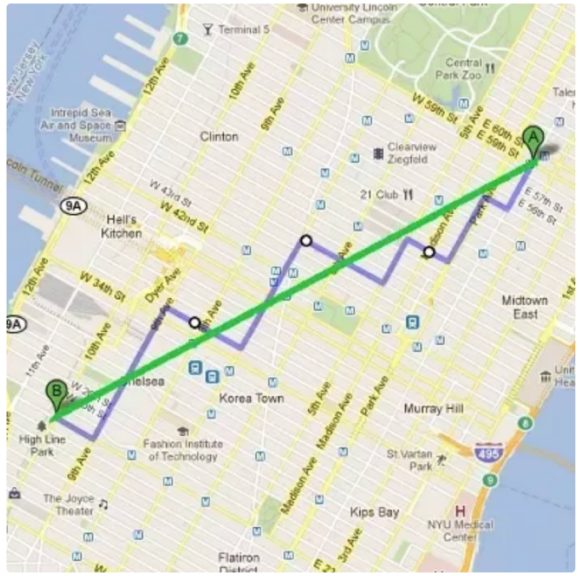
$$d(\mathbf{a}, \mathbf{b}) = \|\mathbf{a} - \mathbf{b}\|_2 = \sqrt{\sum_{i=1}^{D} (a_i - b_i)^2}$$



Common distance / similarity metrics

- **Manhattan distance**



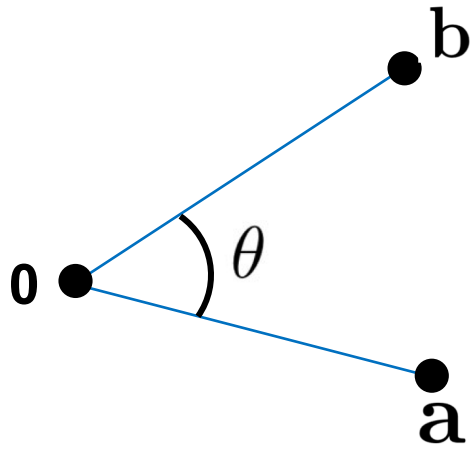


$$d(\mathbf{a}, \mathbf{b}) = \|\mathbf{a} - \mathbf{b}\|_1 = \sum_{i=1}^{D} |a_i - b_i|$$

Common distance / similarity metrics

- **Cosine similarity**

$$s(\mathbf{a}, \mathbf{b}) = \cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \cdot \|\mathbf{b}\|}$$

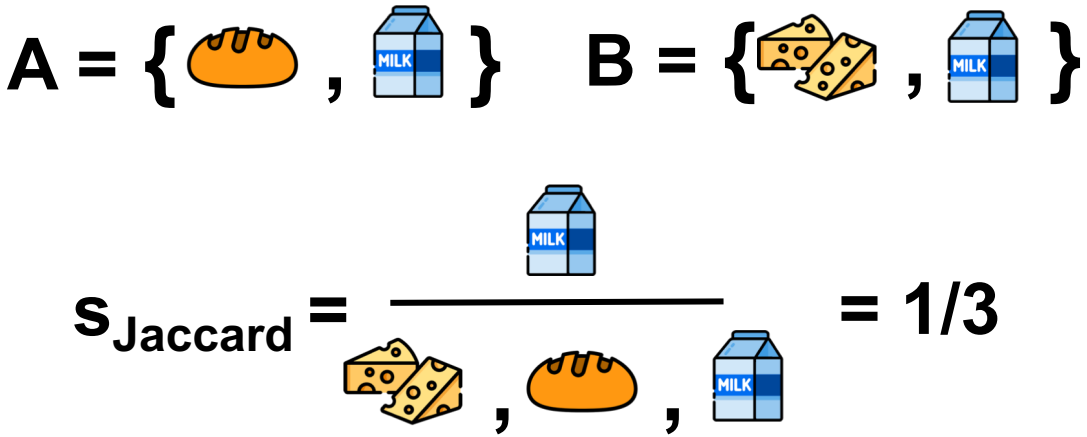


Only considers direction: cosine similarity doesn't change if we scale a or b (i.e. multiplying them by a constant)

Common distance / similarity metrics

- **Jaccard Similarity**
  (between **sets** A and B)

$$s_{\text{Jaccard}}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

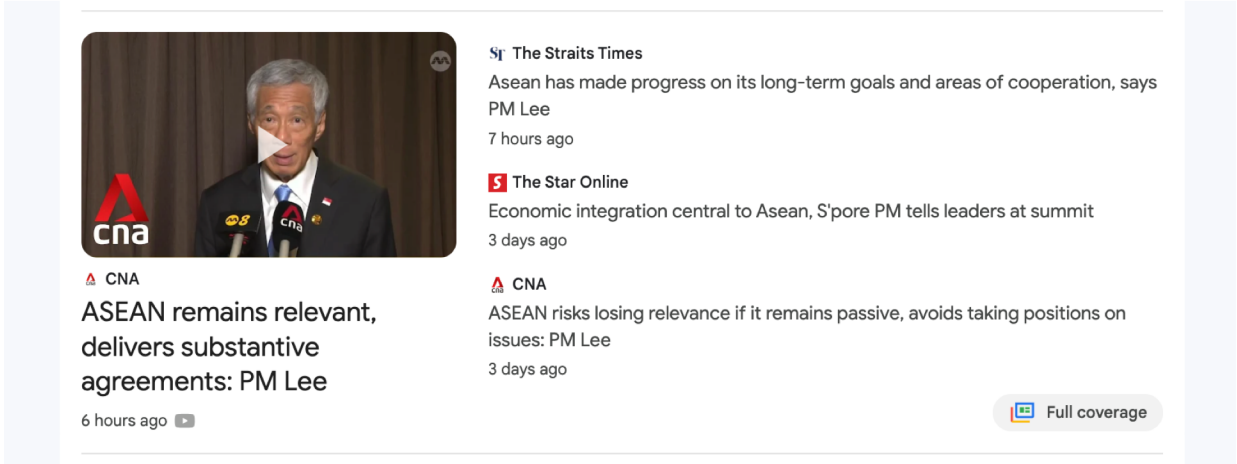


- **Jaccard Distance**

$$d_{\text{Jaccard}}(A, B) = 1 - s_{\text{Jaccard}}(A, B)$$

Task: Finding Similar Documents

- **Goals:**
  - **All pairs similarity:** Given a large number N of documents, find all "near duplicate" pairs, e.g. with Jaccard distance below a threshold
  - **Similarity search: Or**: given a query document D, find all documents which are "near duplicates" with D
- **Applications:**
  - Mirror websites, or approximate mirrors
    - Don't want to show both in search results
  - Similar news articles
    - Cluster articles by "same story"

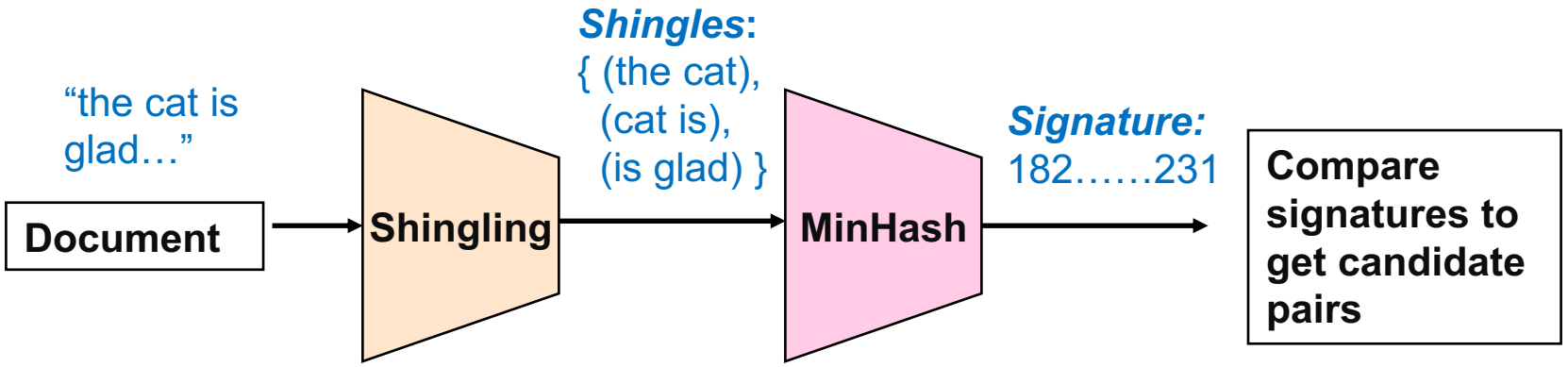


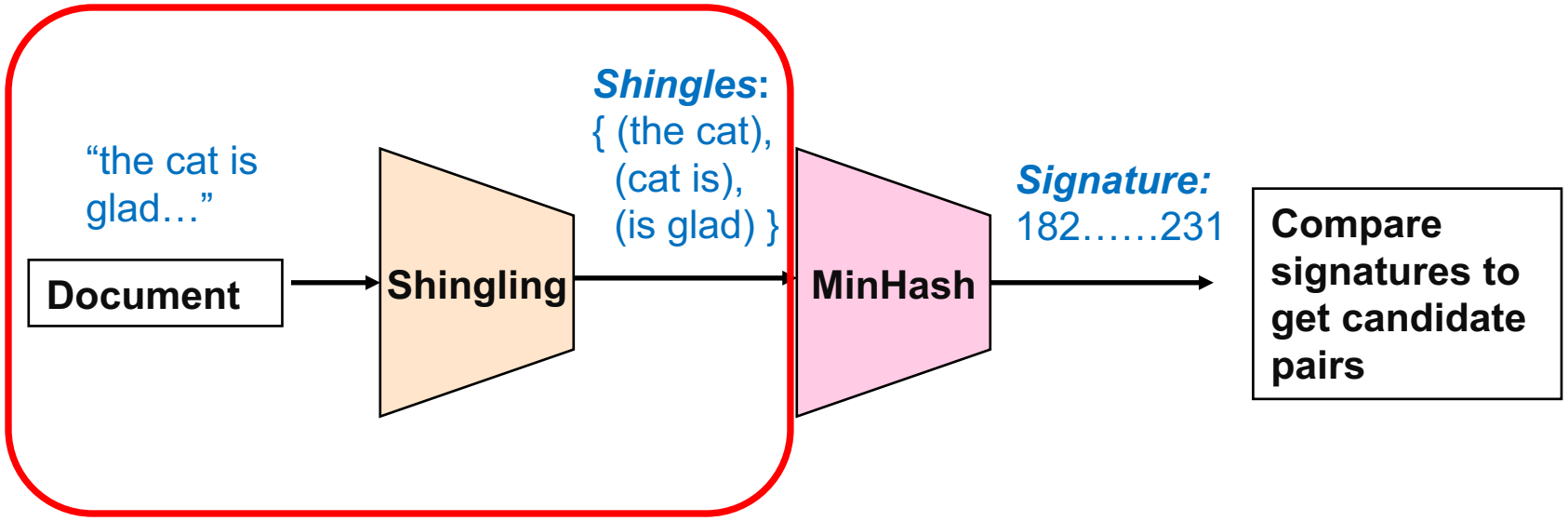
2 Essential Steps for Similar Docs

1. ***Shingling:*** Convert documents to sets of short phrases ("shingles")

2. ***Min-Hashing:*** Convert these sets to short "signatures" of each document, while preserving similarity
   - A "signature" is just a block of data representing the contents of a document in a compressed way
   - Documents with the same signature are candidate pairs for finding near-duplicates

The Big Picture





Shingling

Step 1: ***Shingling:*** Convert documents to sets of phrases

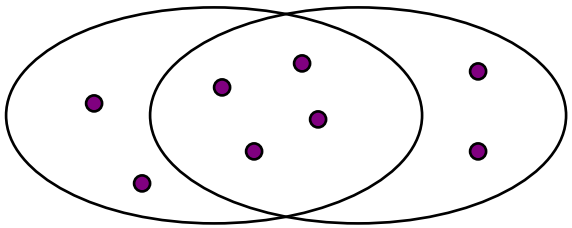
Define: Shingles

- A *k*-shingle (or *k*-gram) for a document is a sequence of *k* tokens that appears in the doc
- **Example: k=2**; document $\mathbf{D_1}$ = "the cat is glad"
  Set of 2-shingles: $\mathbf{S(D_1)}$ = {"the cat", "cat is", "is glad"}

Similarity Metric for Shingles

- Each document D_i can be thought of as a set of its k-shingles C_i
  - E.g. D = "the cat is" ⇒ C = {'the cat', 'cat is'}
- Often represented as a matrix, where columns represent documents, and shingles represent rows
- We measure similarity between documents as **Jaccard similarity**:

$$sim(D_1, D_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$$

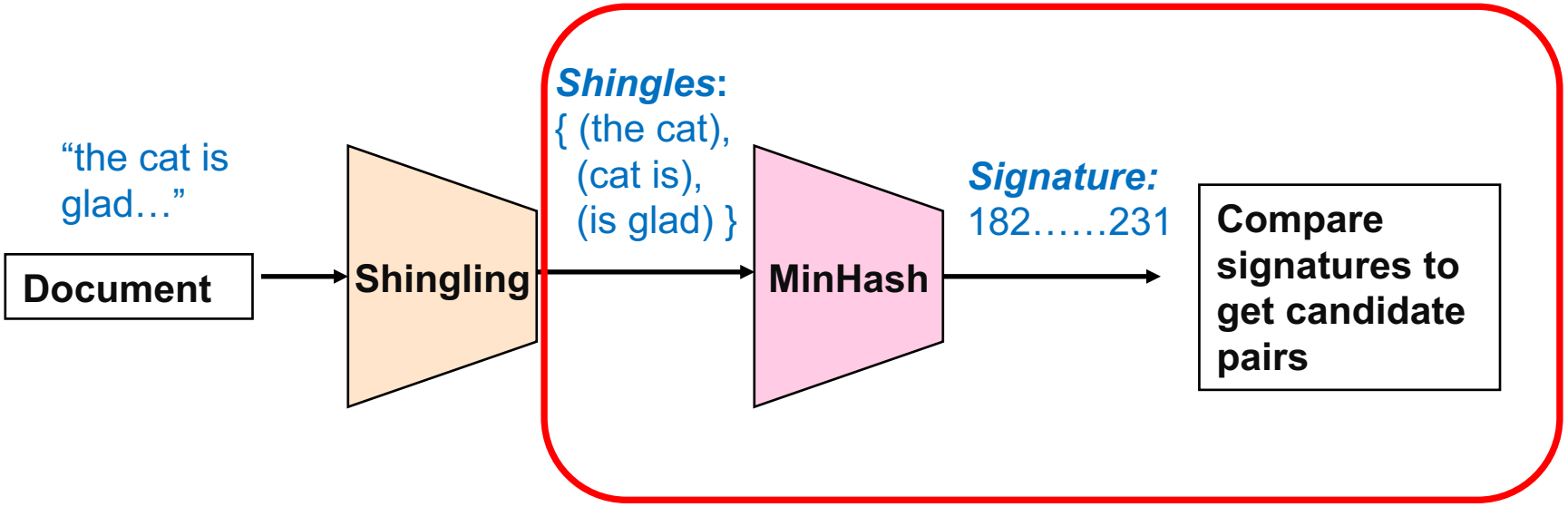


Documents

Shingles	D_1	D_2	…	
"the cat"	1	1	1	0
"cat is"	1	1	0	1
…	0	1	0	1
	0	0	0	1
	1	0	0	1
	1	1	1	0
	1	0	1	0

Motivation for MinHash

- Suppose we have $N = 1$ million documents
- Naively, we would have to compute **pairwise Jaccard similarities** for **every pair of docs**
  - $\boldsymbol{N(N-1)/2 \approx 5*10^{11}}$ comparisons: too slow!
- MinHash gives us a fast approximation to the result of using Jaccard similarities to compare all pairs of documents



MinHashing

Step 2: *MinHash:* Convert **large sets** to **short signatures**, while **preserving similarity**

MinHash: Overview

- **Key idea:** hash each column ***C*** to a small ***signature h(C)***, such that:
  - **(1) *h(C)*** is small enough that the signature fits in RAM
  - **(2)** highly similar documents usually have the same signature
- **Goal: Find a hash function $h(\cdot)$ such that:**
  - If $sim(C_1,C_2)$ is high, then with high probability, $h(C_1) = h(C_2)$
  - If $sim(C_1,C_2)$ is low, then with high probability, $h(C_1) \neq h(C_2)$

MinHash: Steps

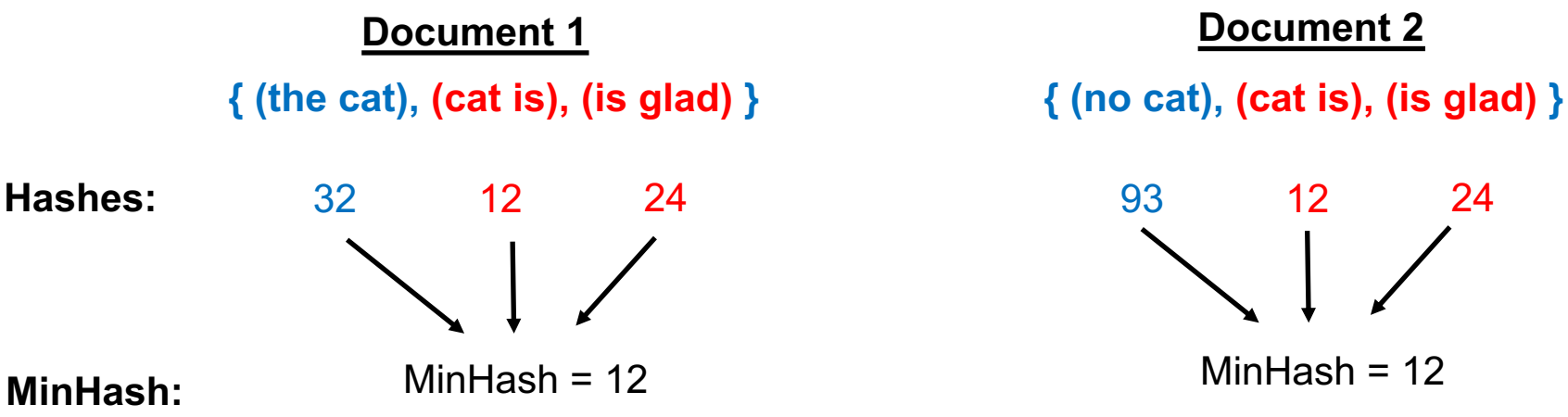
- Given a set of shingles, { (the cat), (cat is), (is glad) }

1. We have a **hash function** h that maps each shingle to an integer:

   h("the cat") = 12, h("cat is") = 74, h("is glad") = 48

2. Then compute the **minimum** of these: min(12, 74, 48) = 12

Example: MinHash on 2 sets of shingles



- Recall that we want to ensure that highly similar documents have high probability to have the same MinHash signature

Key Property of MinHash

- **Key property**: the probability that two documents have the same MinHash signature is equal to their Jaccard similarity!
- Formally: **Pr**[$h(C_1) = h(C_2)$] = *Jaccard-Sim*(C_1, C_2)
- **Proof** (not required knowledge):

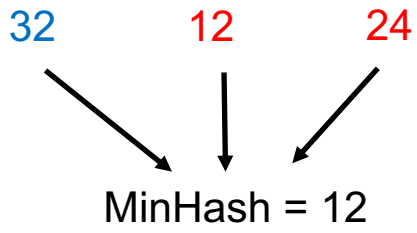
Document 1

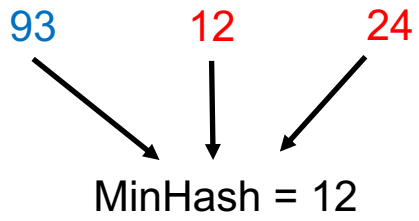
{ (the cat), (cat is), (is glad) }

Document 2

{ (no cat), (cat is), (is glad) }

Hashes:





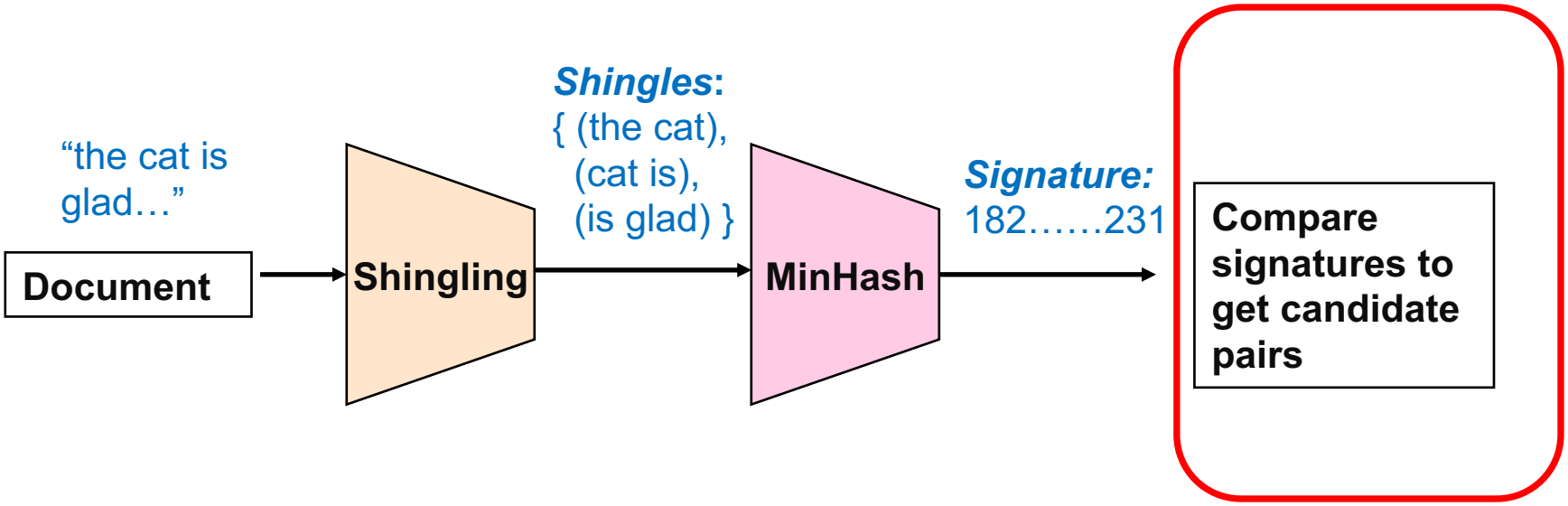
Set of all our tuples: {(the cat), (no cat), (cat is), (is glad)}

(2 non-shared shingles) (2 shared shingles)

Jaccard sim = 2/4 = 0.5

Among these 4 tuples, each has the same probability of having the smallest hash value.
if one of the red shingles has the smallest hash, the documents will have the same MinHash.
Otherwise, they will have different MinHashes.
=> **Pr**[$h(C_1) = h(C_2)$] = (red shingles / total shingles) = (intersection / union) = Jaccard similarity

Putting it all together



- **Candidate pairs**: the documents with the same final signature are "candidate pairs". We can either directly use them as our final output, or compare them one by one to check if they are actually similar pairs.
- **Extension to multiple hashes:** in practice, we usually use multiple hash functions (e.g. N=100), and generate N signatures for each document. "Candidate pairs" can be defined as those matching a 'sufficient number' (e.g. at least 50) among these signatures.

MapReduce Implementation

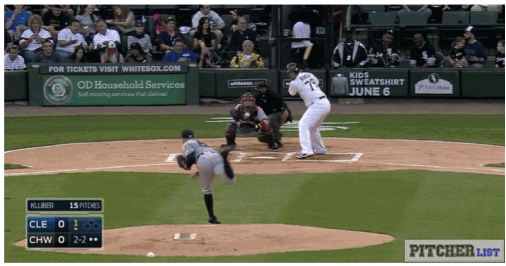
- Map
  - Read over the document and extract its shingles
  - Hash each shingle and take the min of them to get the MinHash signature
  - Emit <signature, document_id>
- Reduce
  - Receive all documents with a given MinHash signature
  - Generate all candidate pairs from these documents
  - (Optional): compare each such pair to check if they are actually similar

Overview

1. Relational Databases and MapReduce
2. Similarity Search
3. **Clustering**

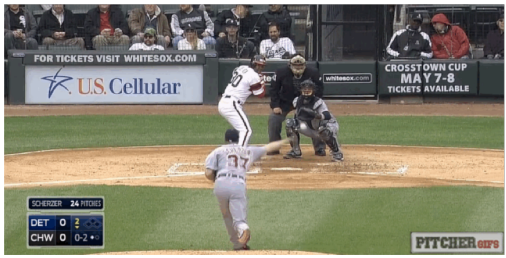
Source: www.flickr.com/photos/rheinitz/6158837748/


SFGIANTS.COM/VO
at&t
FRA
TEMI

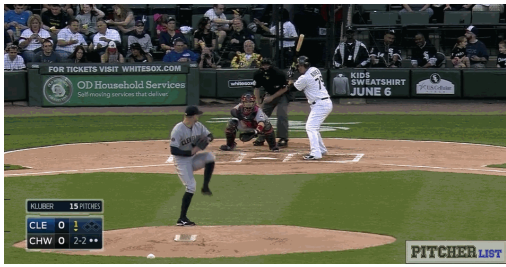

FOR TICKETS VISIT WHITESOX.COM
OD Household Services
KIDS SWEATSHIRT
JUNE 6
KLUBER 15 PITCHES
CLE 0
CHW 0 2-2
PITCHER LIST


KIA 4
5 삼성
삼성카드 삼성증권 삼성생
SPOTV
KCW
대구은행
하루100
무항생제
치킨더홈
땅땅치킨


mariners.com
T-Mobile
T-Mobile
IWAKUMA 33 PITCHES
TB 0
SEA 1 0-2
PITCHER GIFS


FOR TICKETS VISIT WHITESOX.COM
U.S. Cellular
CROSSTOWN CUP
MAY 7-8
TICKETS AVAILABLE
SCHERZER 24 PITCHES
DET 0
CHW 0 0-2
PITCHER GIFS


SFGIANTS.COM/VOT
at&t

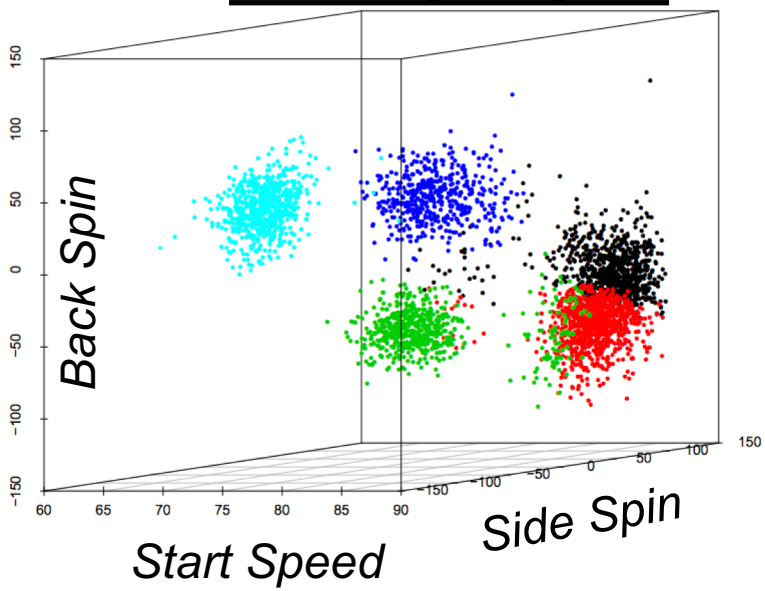

FOR TICKETS VISIT WHITESOX.COM
OD Household Services
KIDS SWEATSHIRT JUNE 6
KLUBER 15 PITCHES
CLE 0
CHW 0 2-2
PITCHER LIST

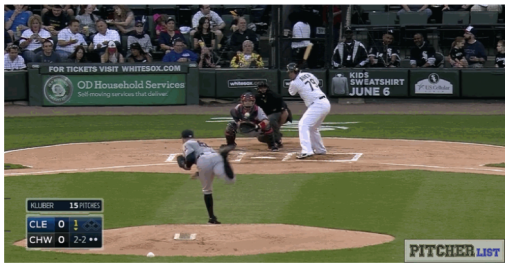

KIA 4 5 삼성
KCW
대구은행
하림100 무항생제 치킨너겟
삼성카드 삼성증권
SPOTV

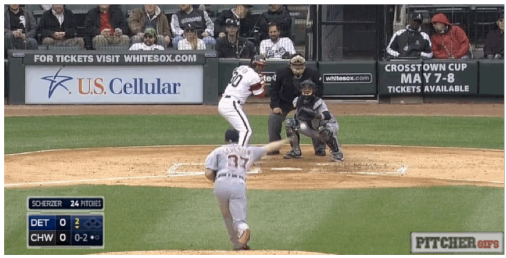

mariners.com
T-Mobile
MANUMA 33 PITCHES
TB 0
SEA 1 0-2
PITCHER OPS


FOR TICKETS VISIT WHITESOX.COM
U.S. Cellular
CROSSTOWN CUP MAY 7-8 TICKETS AVAILABLE
SCHERZER 24 PITCHES
DET 0
CHW 0 0-2
PITCHER OPS

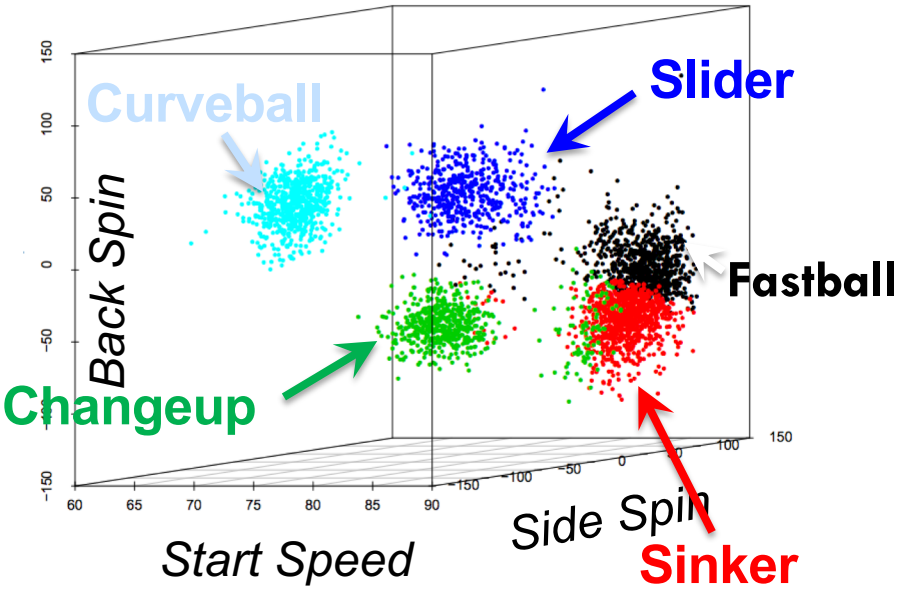
Pitches by Barry Zito

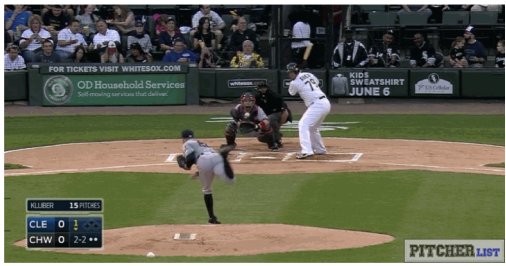


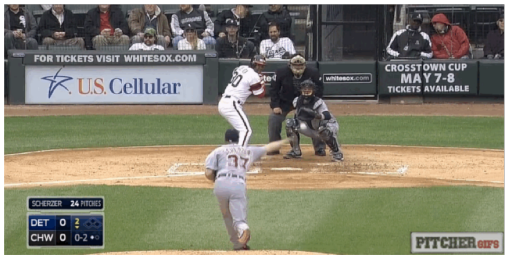




Pitches by Barry Zito



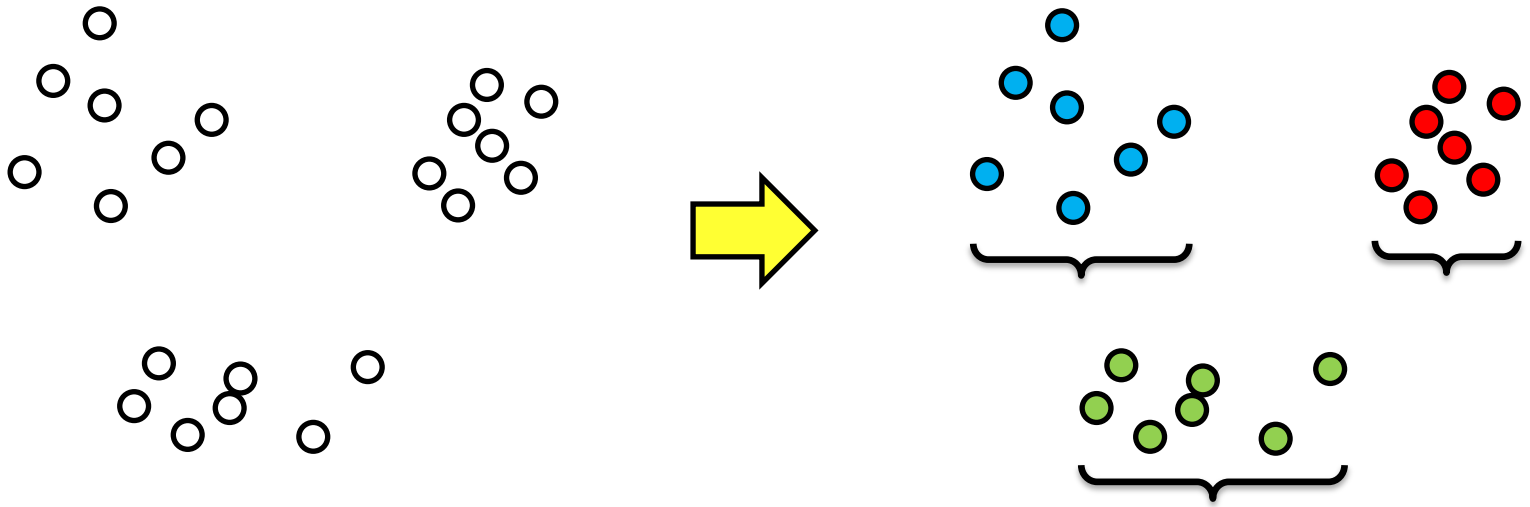




Goal of clustering

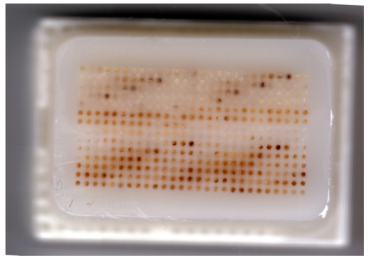
Clustering separates unlabelled data into groups of similar points.

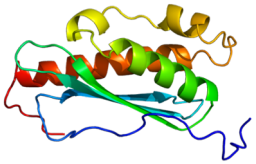
Clusters should have high intra-cluster similarity, and low inter-cluster similarity.



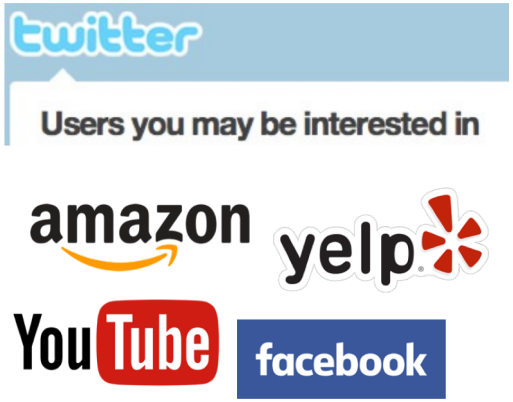
Applications of clustering

Many applications:

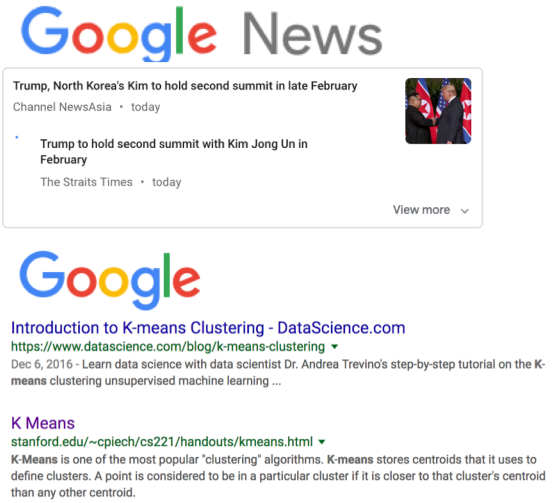




Microbiology: find groups of related genes (or proteins etc.)



Recommendation & Social Networks: find groups of similar users



Search & Information Retrieval: grouping similar search (or news etc.) results

K-Means Algorithm: Steps

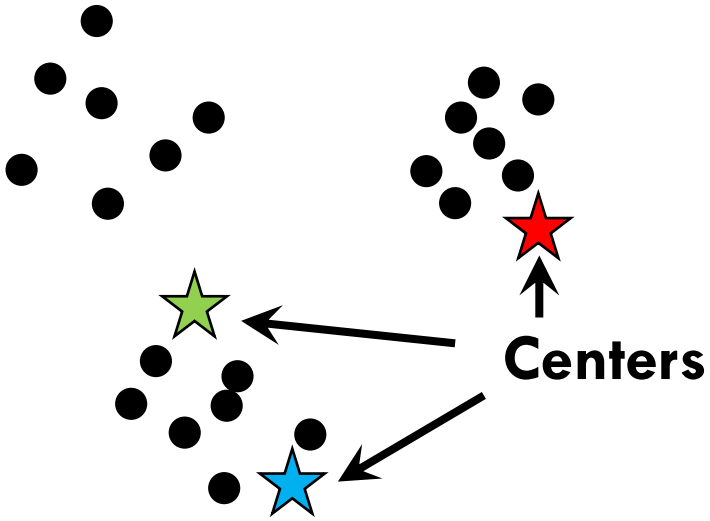
1. Initialization: Pick K random points as centers

2. Repeat:

   a) **Assignment:** assign each point to nearest cluster

   b) **Update:** move each cluster center to average of its assigned points

   Stop if no assignments change



K-Means Algorithm: Steps

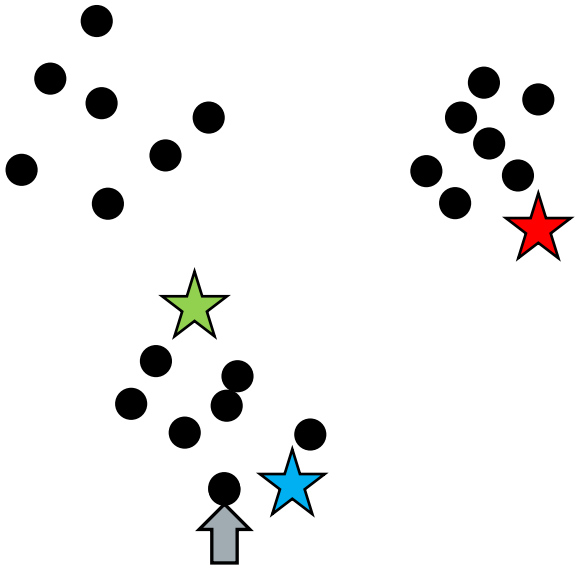
1. **Initialization**: Pick K random points as centers

2. **Repeat:**

   a) **Assignment:** assign each point to nearest cluster

   b) **Update:** move each cluster center to average of its assigned points

   Stop if no assignments change



K-Means Algorithm: Steps

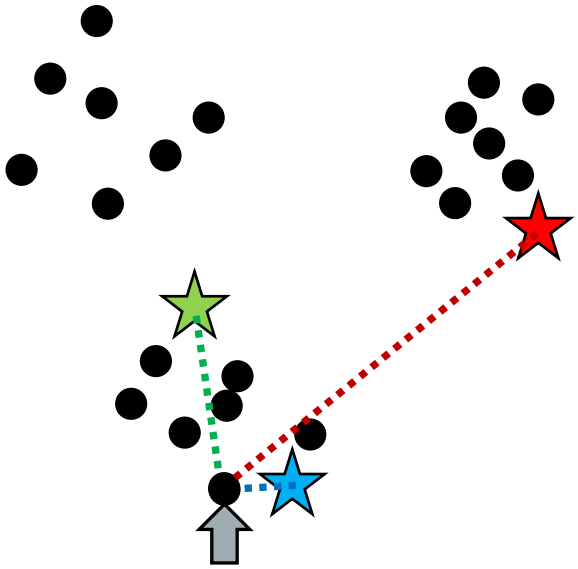
1. **Initialization**: Pick K random points as centers

2. **Repeat:**

   a) **Assignment:** assign each point to nearest cluster

   b) **Update:** move each cluster center to average of its assigned points

   Stop if no assignments change



K-Means Algorithm: Steps

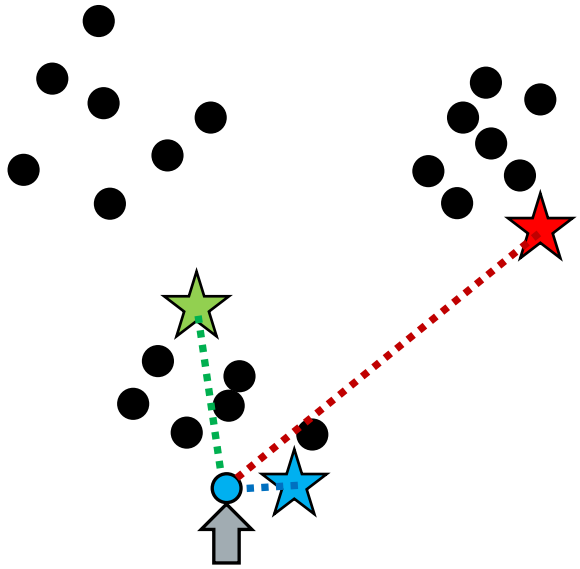
1. **Initialization**: Pick K random points as centers

2. **Repeat:**

   a) **Assignment:** assign each point to nearest cluster

   b) **Update:** move each cluster center to average of its assigned points

   Stop if no assignments change



K-Means Algorithm: Steps

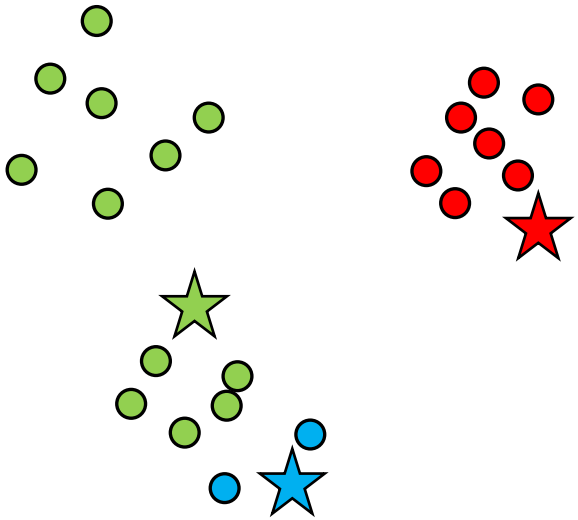
1. **Initialization**: Pick K random points as centers

2. Repeat:

a) **Assignment:** assign each point to nearest cluster

b) **Update:** move each cluster center to average of its assigned points

Stop if no assignments change



K-Means Algorithm: Steps

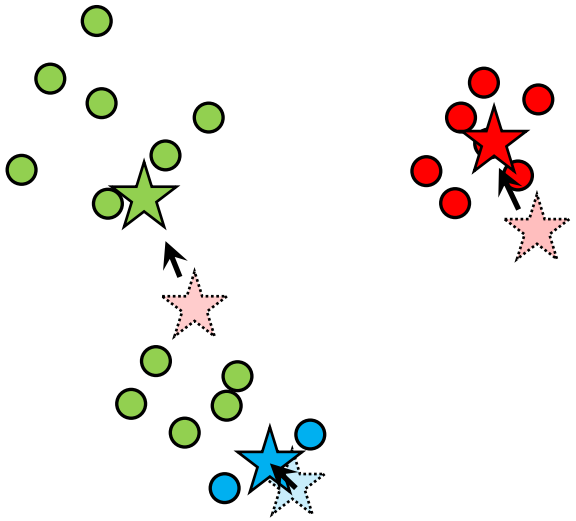
1. **Initialization**: Pick K random points as centers

2. **Repeat:**

   a) **Assignment:** assign each point to nearest cluster

   b) **Update:** move each cluster center to average of its assigned points

   Stop if no assignments change



K-Means Algorithm: Steps

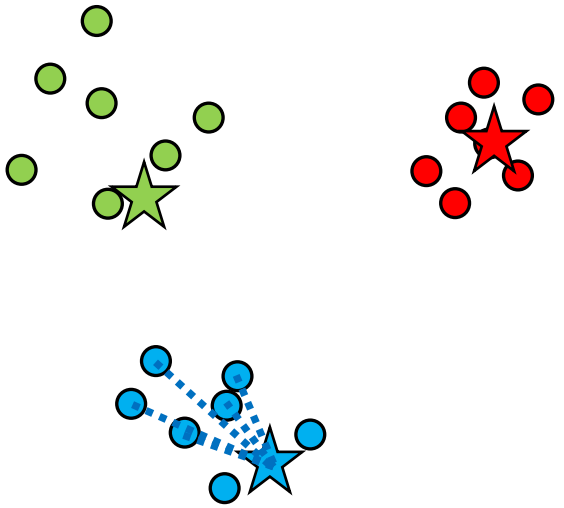
1. **Initialization**: Pick K random points as centers

2. Repeat:

a) **Assignment:** assign each point to nearest cluster

b) **Update:** move each cluster center to average of its assigned points

Stop if no assignments change



K-Means Algorithm: Steps

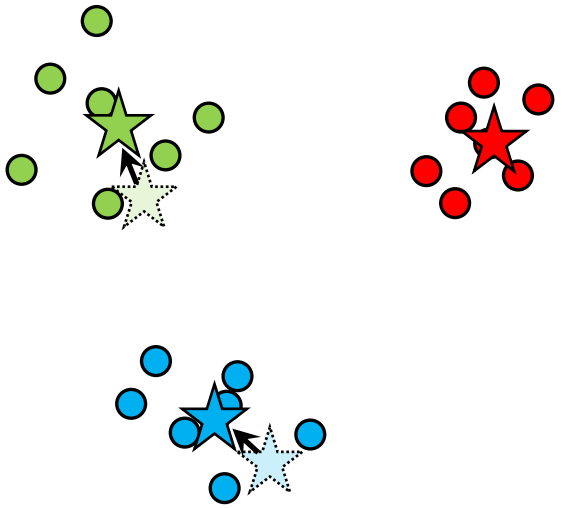
1. **Initialization**: Pick K random points as centers

2. **Repeat:**

   a) **Assignment:** assign each point to nearest cluster

   b) **Update:** move each cluster center to average of its assigned points

   Stop if no assignments change



K-Means Algorithm: Steps

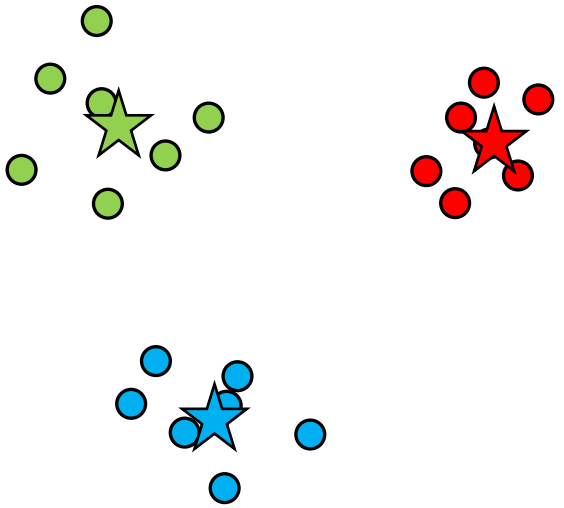
1. **Initialization**: Pick K random points as centers

2. Repeat:

a) **Assignment:** assign each point to nearest cluster

b) **Update:** move each cluster center to average of its assigned points

Stop if no assignments change



K-Means Algorithm: Steps

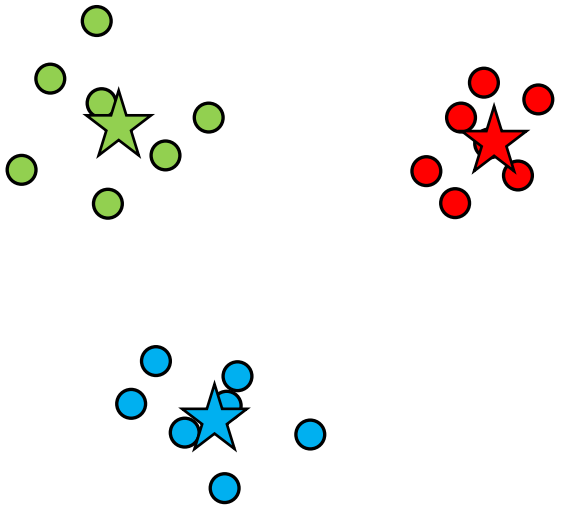
1. Initialization: Pick K random points as centers

2. Repeat:

a) **Assignment:** assign each point to nearest cluster

b) **Update:** move each cluster center to average of its assigned points

Stop if no assignments change



MapReduce Implementation v1

This MapReduce job performs a **single iteration** of k-means. It receives as input the current positions of the cluster centers (i.e. stars)

```
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!

What is the problem?

Q: How much disk I/O is exchanged between the mappers and reducers, in MapReduce K-means v1?

(let n = no. of points, m = no. of iterations, d = dimensionality, k = no. of centers)

1. O(n)
2. O(md)
3. O(nmd)
4. O(kmd)

Q: How much disk I/O is exchanged between the mappers and reducers, in MapReduce K-means v1?

(let n = no. of points, m = no. of iterations, d = dimensionality, k = no. of centers)

1. O(n)
2. O(md)
3. O(nmd)
4. O(kmd)

MapReduce Implementation v2 (with 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)
```

Q: How much disk I/O is exchanged between the mappers and reducers, in MapReduce K-means v2?

(let n = no. of points, m = no. of iterations, d = dimensionality, k = no. of centers)

(you may assume there is only 1 map task)

1. O(n)
2. O(md)
3. O(nmd)
4. O(kmd)

Q: How much disk I/O is exchanged between the mappers and reducers, in MapReduce K-means v2?

(let n = no. of points, m = no. of iterations, d = dimensionality, k = no. of centers)

(you may assume there is only 1 map task)

1. O(n)
2. O(md)
3. O(nmd)
4. O(kmd)