Big Data Infrastructure

Session 4: MapReduce – Structured and Unstructured Data

Jimmy Lin
University of Maryland
Monday, February 23, 2015

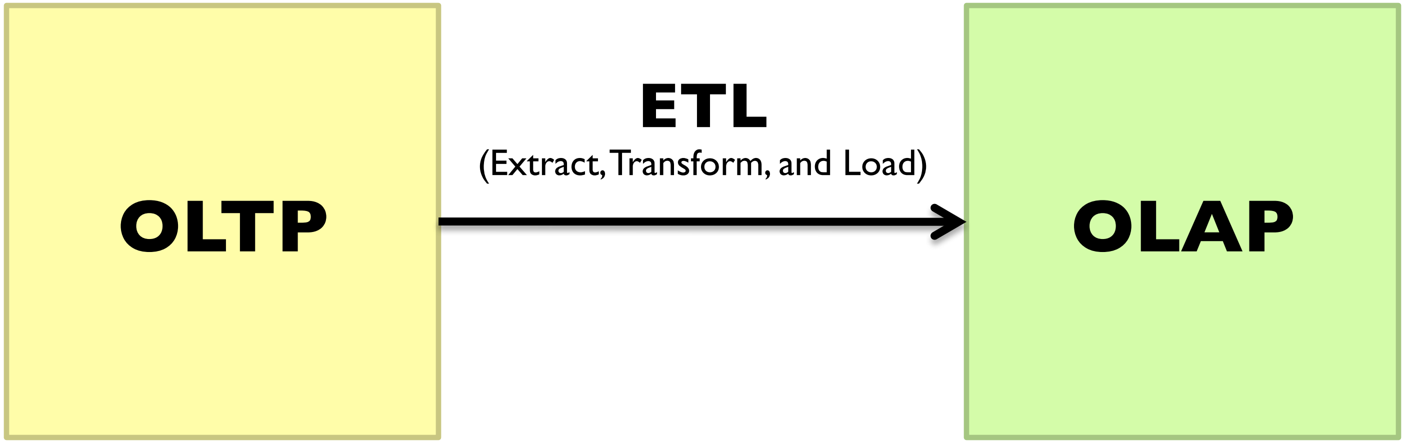
Today's Agenda

- Structured data
  - Processing relational data with MapReduce
- Unstructured data
  - Basics of indexing and retrieval
  - Inverted indexing in MapReduce

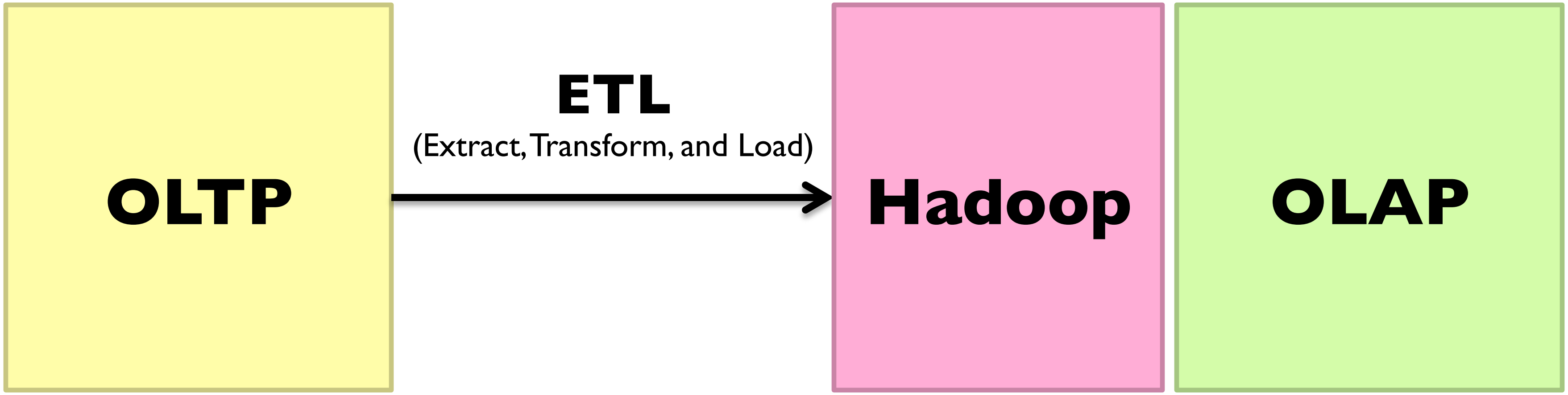
Relational Databases

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

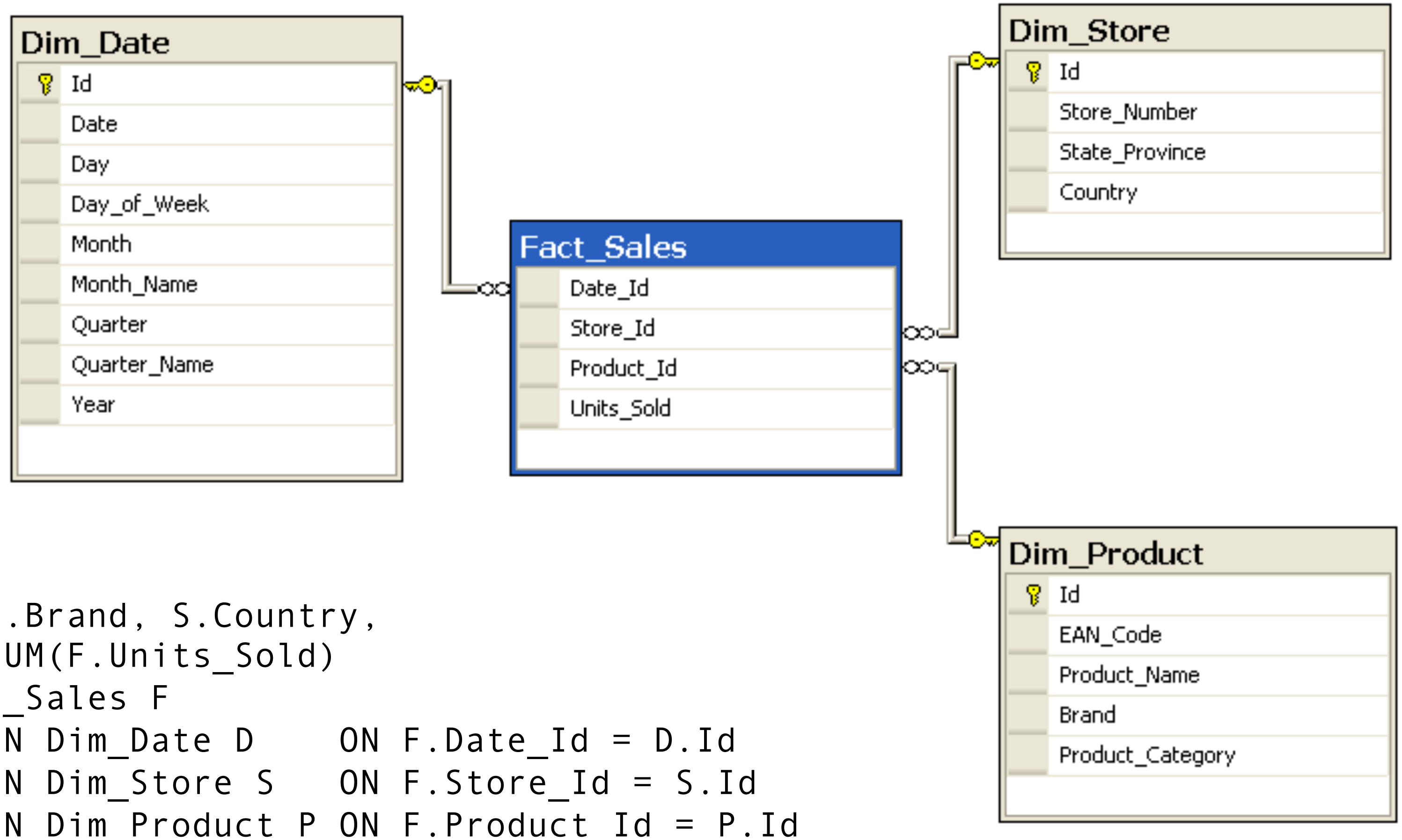
OLTP/OLAP Architecture



OLTP/OLAP/Hadoop Architecture



Structure of Data Warehouses



Dim_Date
- Id
- Date
- Day
- Day_of_Week
- Month
- Month_Name
- Quarter
- Quarter_Name
- Year

Fact_Sales
- Date_Id
- Store_Id
- Product_Id
- Units_Sold

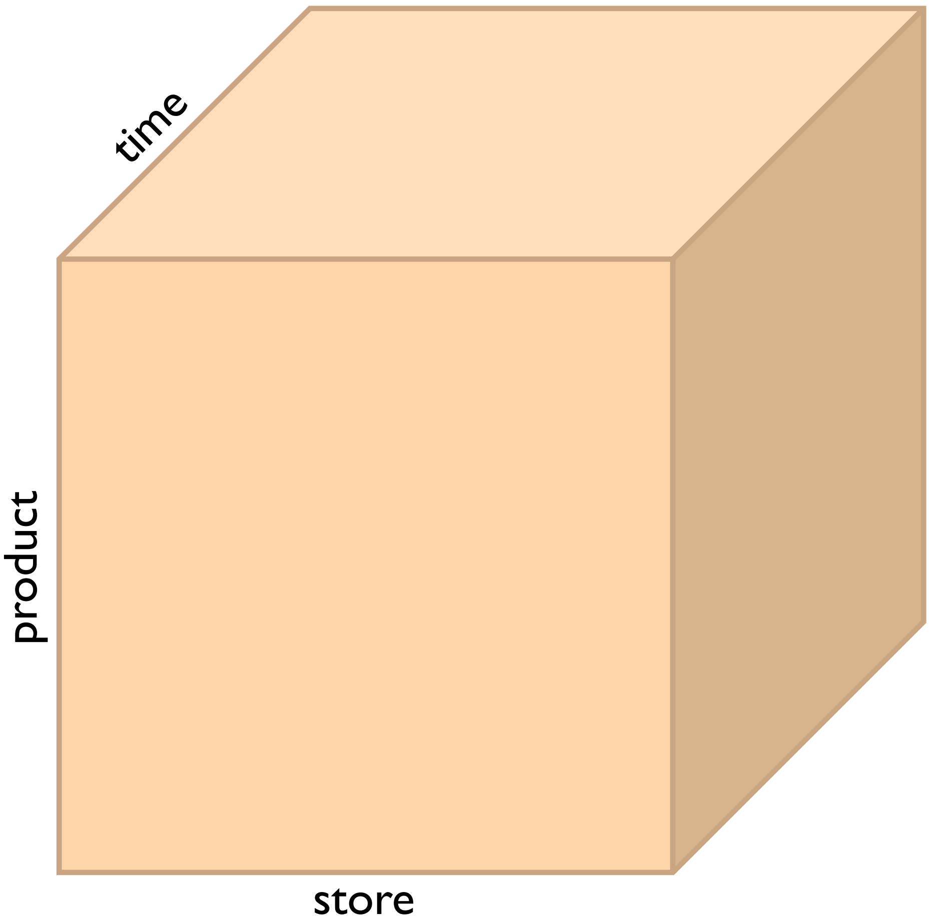
Dim_Store
- Id
- Store_Number
- State_Province
- Country

Dim_Product
- Id
- EAN_Code
- Product_Name
- Brand
- Product_Category

```
SELECT  P.Brand, S.Country,
        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)

OLAP Cubes



Common operations

slice and dice

roll up/drill down

pivot

MapReduce algorithms for processing relational data

Source: www.flickr.com/photos/stikatphotography/1590190676/

Source: www.flickr.com/photos/7518432@N06/2237536651/

Design Pattern: Secondary Sorting

- MapReduce sorts input to reducers by key
  - Values are arbitrarily ordered
- What if want to sort value also?
  - E.g., k → $(v_1, r), (v_3, r), (v_4, r), (v_8, r)\ldots$

Secondary Sorting: Solutions

- Solution 1:
  - Buffer values in memory, then sort
  - Why is this a bad idea?
- Solution 2:
  - "Value-to-key conversion" design pattern: form composite intermediate key, (k, v_1)
  - Let execution framework do the sorting
  - Preserve state across multiple key-value pairs to handle processing
  - Anything else we need to do?

Value-to-Key Conversion

Before

$k \rightarrow (v_1, r), (v_4, r), (v_8, r), (v_3, r)\ldots$

Values arrive in arbitrary order…

After

$(k, v_1) \rightarrow (v_1, r)$

$(k, v_3) \rightarrow (v_3, r)$

$(k, v_4) \rightarrow (v_4, r)$

$(k, v_8) \rightarrow (v_8, r)$

$\ldots$

Values arrive in sorted order…

Process by preserving state across multiple keys

Remember to partition correctly!

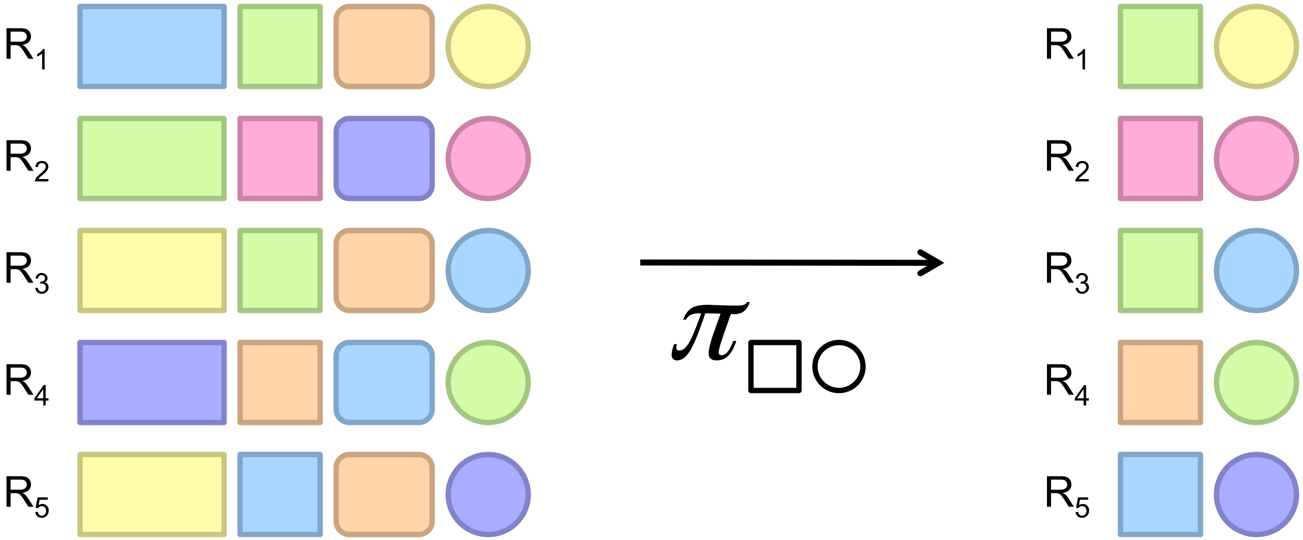
Working Scenario

- Two tables:
  - User demographics (gender, age, income, etc.)
  - User page visits (URL, time spent, etc.)
- Analyses we might want to perform:
  - Statistics on demographic characteristics
  - Statistics on page visits
  - Statistics on page visits by URL
  - Statistics on page visits by demographic characteristic
  - …

Relational Algebra

- Primitives
  - Projection (π)
  - Selection (σ)
  - Cartesian product (×)
  - Set union (∪)
  - Set difference (-)
  - Rename (ρ)
- Other operations
  - Join (⋈)
  - Group by… aggregation
  - …

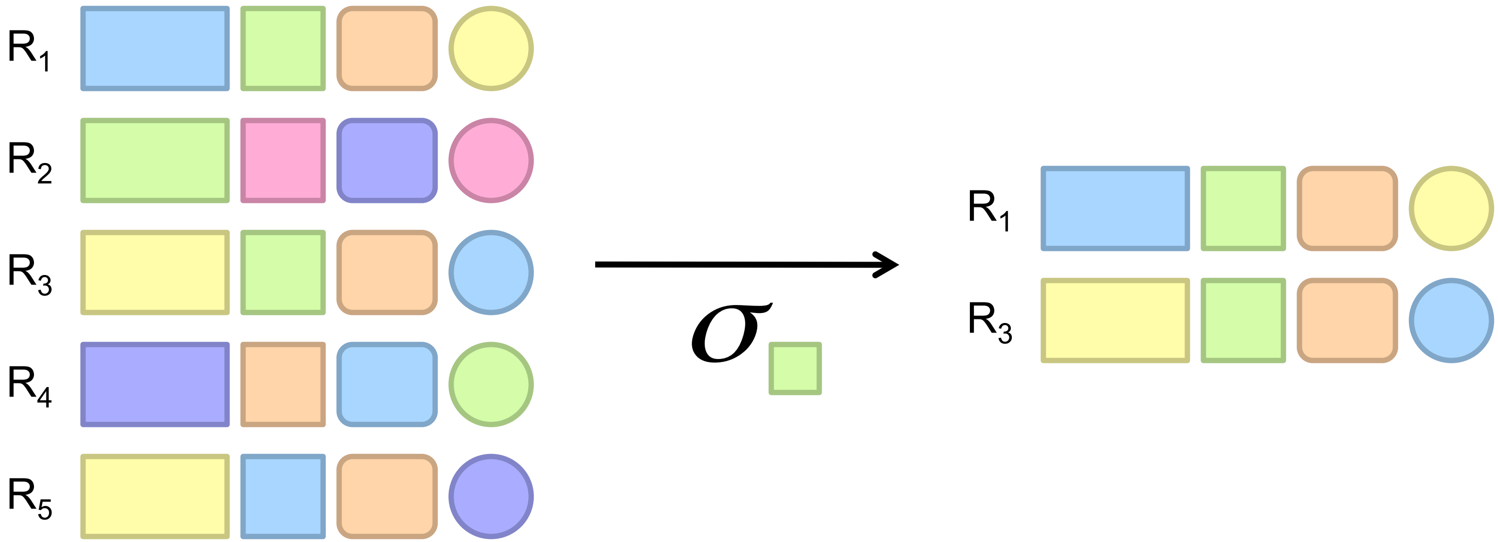
Projection



Projection in MapReduce

- Easy!
  - Map over tuples, emit new tuples with appropriate attributes
  - No reducers, unless for regrouping or resorting tuples
  - Alternatively: perform in reducer, after some other processing
- Basically limited by HDFS streaming speeds
  - Speed of encoding/decoding tuples becomes important
  - Take advantage of compression when available
  - Semistructured data? No problem!

Selection



Selection in MapReduce

- Easy!
  - Map over tuples, emit only tuples that meet criteria
  - No reducers, unless for regrouping or resorting tuples
  - Alternatively: perform in reducer, after some other processing
- Basically limited by HDFS streaming speeds
  - Speed of encoding/decoding tuples becomes important
  - Take advantage of compression when available
  - Semistructured data? No problem!

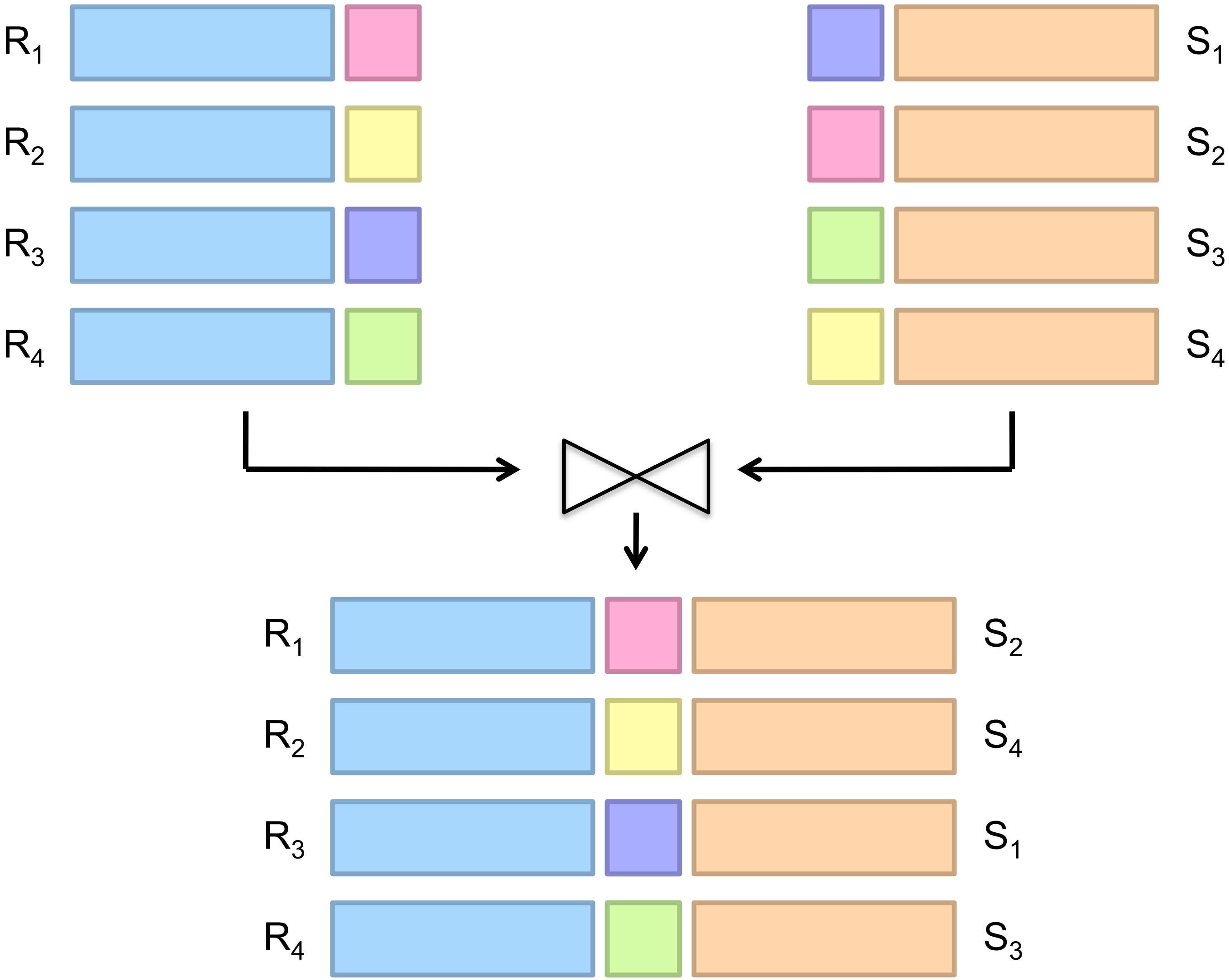
Group by… Aggregation

- Example: What is the average time spent per URL?
- In SQL:
  - SELECT url, AVG(time) FROM visits GROUP BY url
- In MapReduce:
  - Map over tuples, emit time, keyed by url
  - Framework automatically groups values by keys
  - Compute average in reducer
  - Optimize with combiners

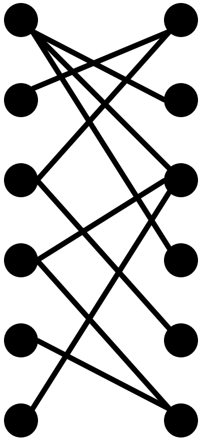
Relational Joins

Source: Microsoft Office Clip Art

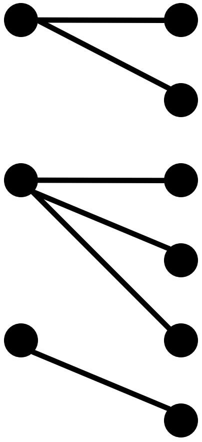
Relational Joins



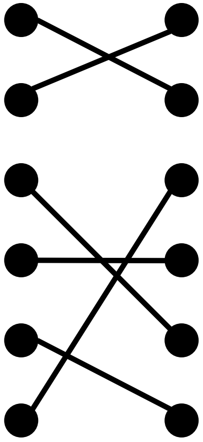
Types of Relationships



Many-to-Many



One-to-Many



One-to-One

Join Algorithms in MapReduce

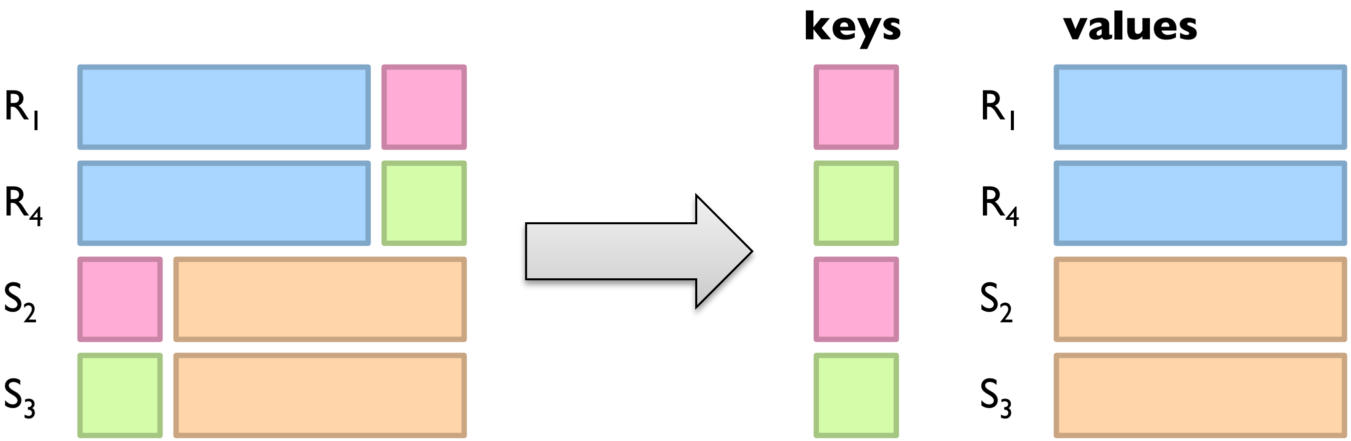
- Reduce-side join
- Map-side join
- In-memory join
  - Striped variant
  - Memcached variant

Reduce-side Join

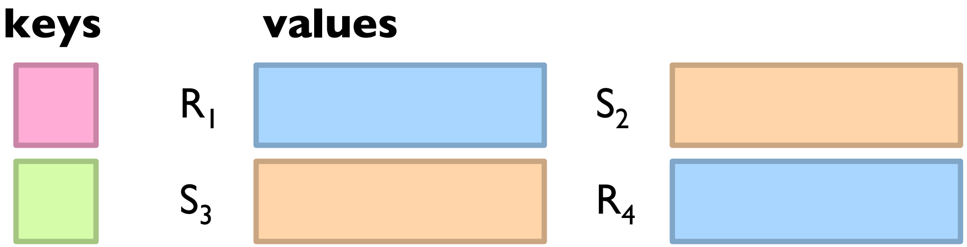
- Basic idea: group by join key
  - Map over both sets of tuples
  - Emit tuple as value with join key as the intermediate key
  - Execution framework brings together tuples sharing the same key
  - Perform actual join in reducer
  - Similar to a “sort-merge join” in database terminology
- Two variants
  - 1-to-1 joins
  - 1-to-many and many-to-many joins

Reduce-side Join: 1-to-1

Map



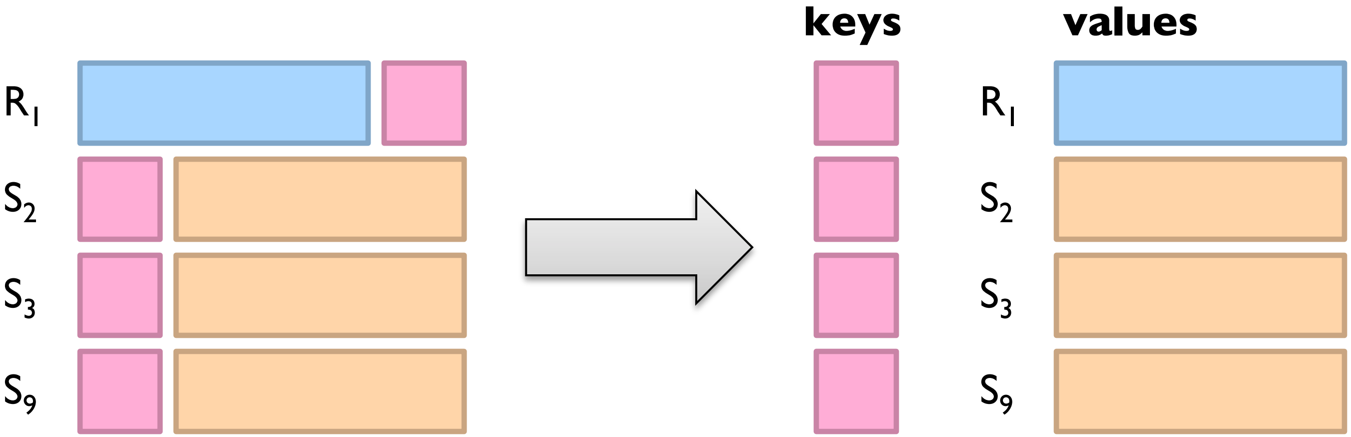
Reduce



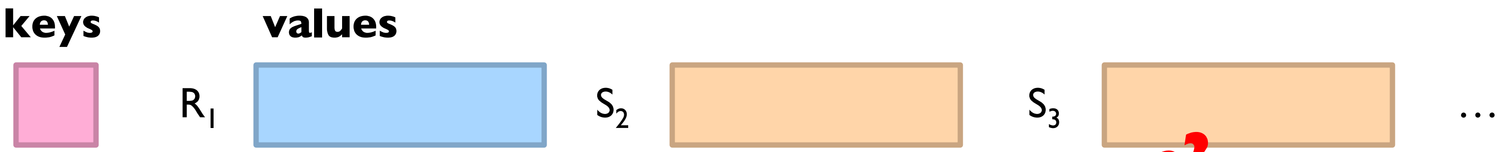
Note: no guarantee if R is going to come first or S

Reduce-side Join: 1-to-many

Map



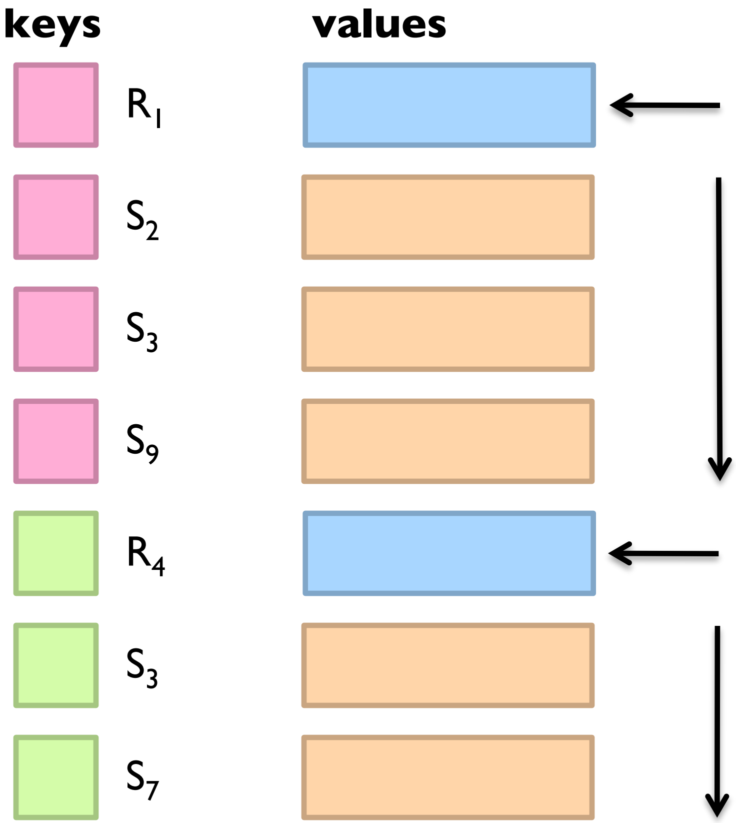
Reduce



What's the problem?

Reduce-side Join: V-to-K Conversion

In reducer...



New key encountered: hold in memory

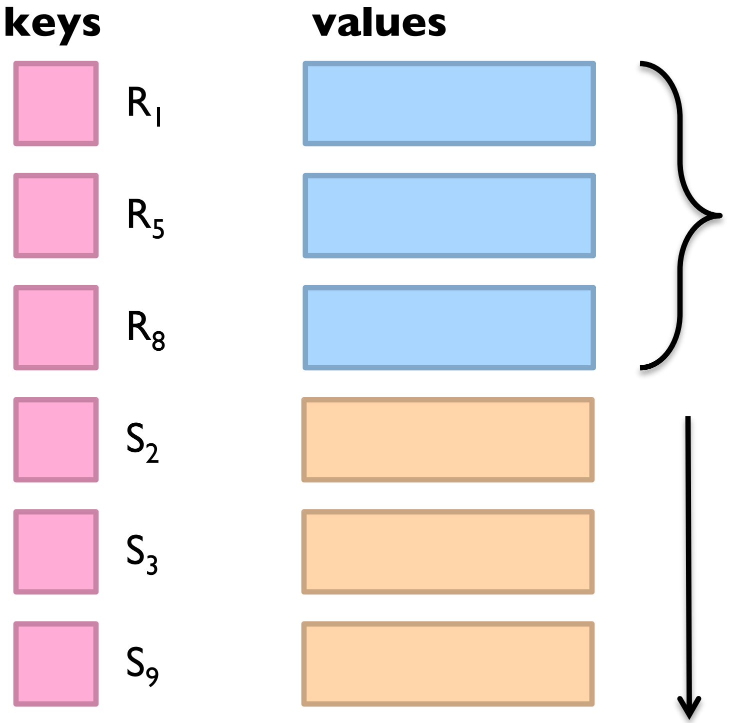
Cross with records from other set

New key encountered: hold in memory

Cross with records from other set

Reduce-side Join: many-to-many

In reducer…



keys

values

R_1

R_5

R_8

S_2

S_3

S_9

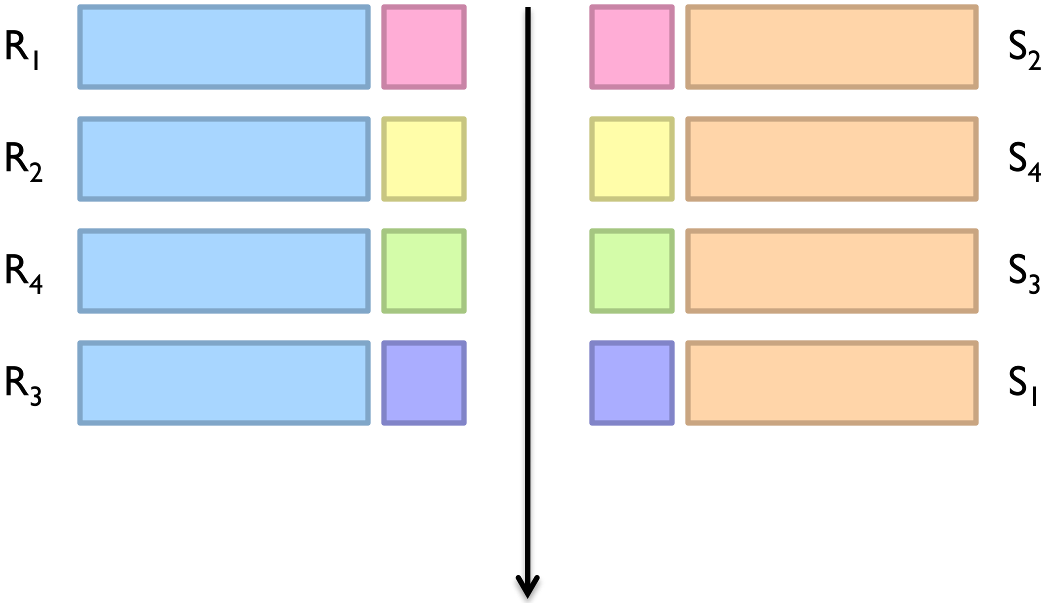
Hold in memory

Cross with records from other set

What's the problem?

Map-side Join: Basic Idea

Assume two datasets are sorted by the join key:



A sequential scan through both datasets to join
(called a "merge join" in database terminology)

Map-side Join: Parallel Scans

- If datasets are sorted by join key, join can be accomplished by a scan over both datasets
- How can we accomplish this in parallel?
  - Partition and sort both datasets in the same manner
- In MapReduce:
  - Map over one dataset, read from other corresponding partition
  - No reducers necessary (unless to repartition or resort)
- Consistently partitioned datasets: realistic to expect?

In-Memory Join

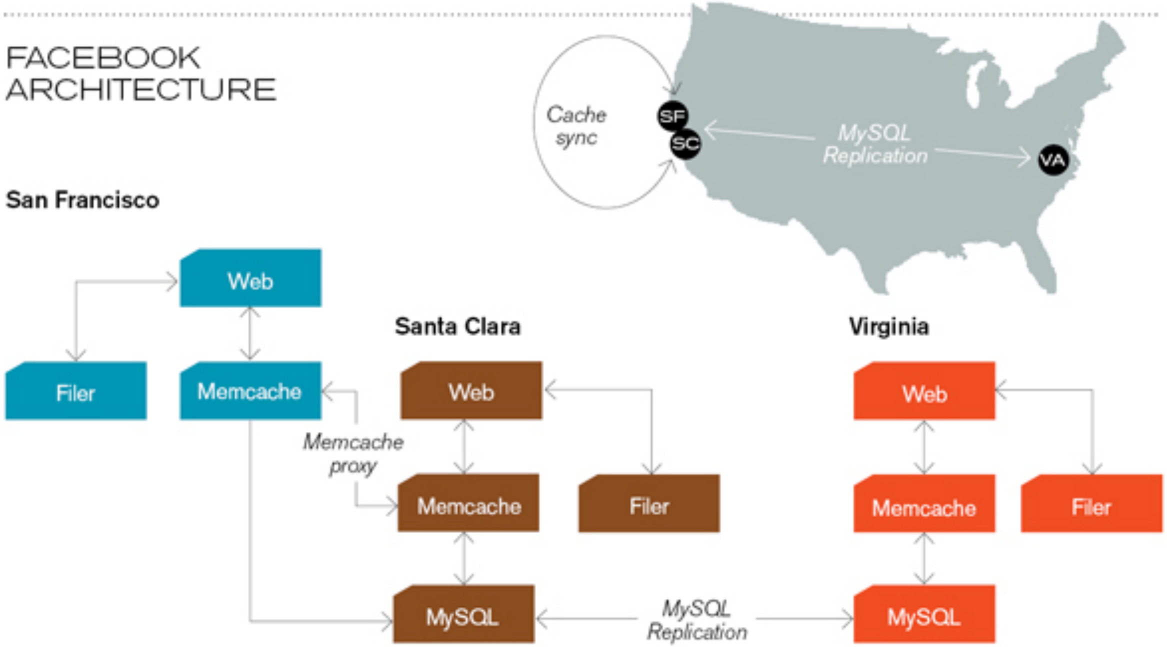
- Basic idea: load one dataset into memory, stream over other dataset
  - Works if R << S and R fits into memory
  - Called a “hash join” in database terminology
- MapReduce implementation
  - Distribute R to all nodes
  - Map over S, each mapper loads R in memory, hashed by join key
  - For every tuple in S, look up join key in R
  - No reducers, unless for regrouping or resorting tuples

In-Memory Join: Variants

- Striped variant:
  - R too big to fit into memory?
  - Divide R into R_1, R_2, R_3, … s.t. each R_n fits into memory
  - Perform in-memory join: $\forall n$, $R_n \bowtie S$
  - Take the union of all join results
- Memcached join:
  - Load R into memcached
  - Replace in-memory hash lookup with memcached lookup

Memcached

Circa 2008 Architecture



Caching servers: 15 million requests per second, 95% handled by memcache (15 TB of RAM)

Database layer: 800 eight-core Linux servers running MySQL (40 TB user data)

Source: Technology Review (July/August, 2008)

Memcached Join

- Memcached join:
    - Load R into memcached
    - Replace in-memory hash lookup with memcached lookup
- Capacity and scalability?
    - Memcached capacity >> RAM of individual node
    - Memcached scales out with cluster
- Latency?
    - Memcached is fast (basically, speed of network)
    - Batch requests to amortize latency costs

Source: See tech report by Lin et al. (2009)

Which join to use?

- In-memory join > map-side join > reduce-side join
  - Why?
- Limitations of each?
  - In-memory join: memory
  - Map-side join: sort order and partitioning
  - Reduce-side join: general purpose

Processing Relational Data: Summary

- MapReduce algorithms for processing relational data:
  - Group by, sorting, partitioning are handled automatically by shuffle/sort in MapReduce
  - Selection, projection, and other computations (e.g., aggregation), are performed either in mapper or reducer
  - Multiple strategies for relational joins
- Complex operations require multiple MapReduce jobs
  - Example: top ten URLs in terms of average time spent
  - Opportunities for automatic optimization

Source: www.flickr.com/photos/7518432@N06/2237536651/

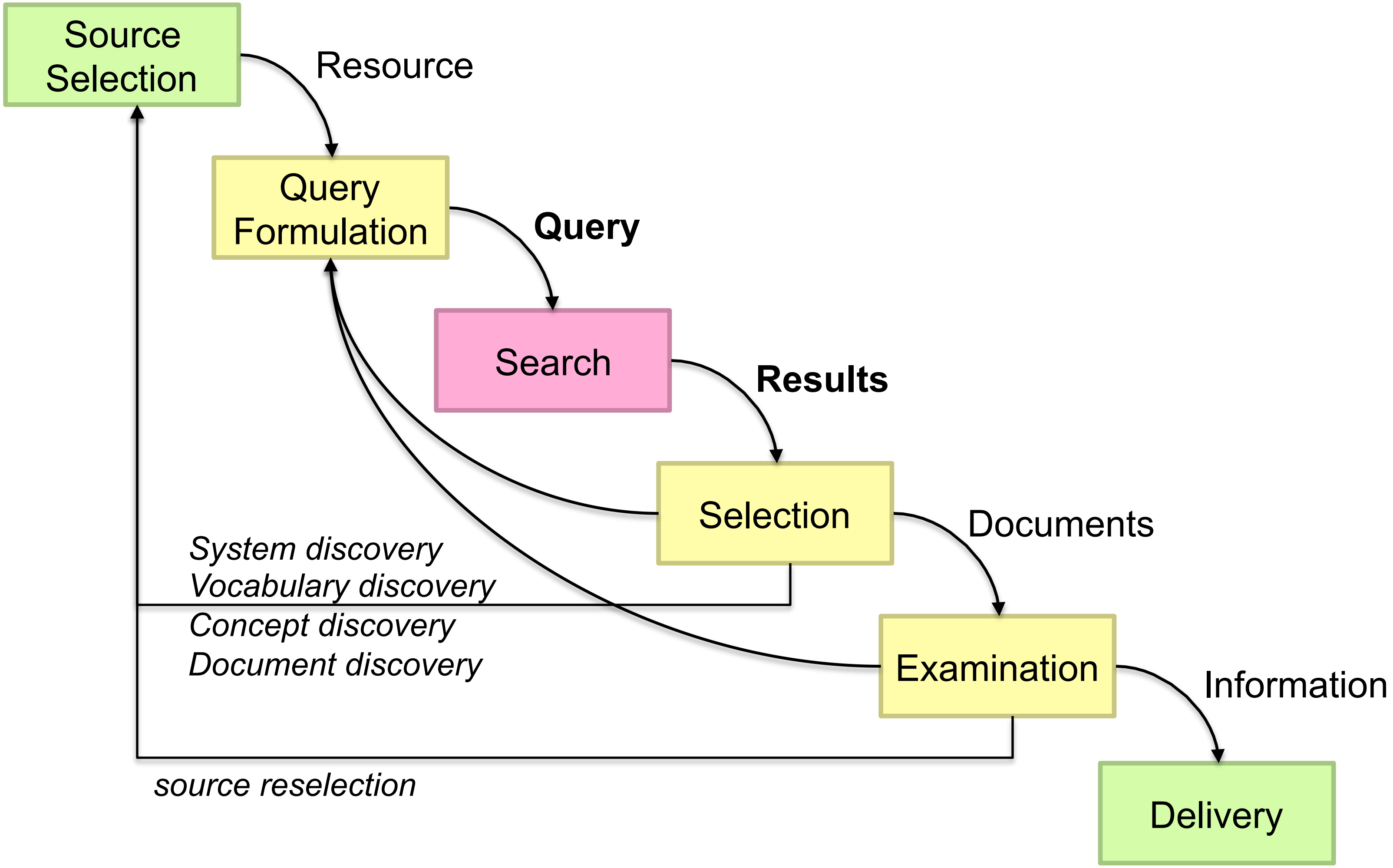
Today's Agenda

- Structured data
  - Processing relational data with MapReduce
- Unstructured data
  - Basics of indexing and retrieval
  - Inverted indexing in MapReduce

First, nomenclature…

- Information retrieval (IR)
  - Focus on textual information (= text/document retrieval)
  - Other possibilities include image, video, music, …
- What do we search?
  - Generically, “collections”
  - Less-frequently used, “corpora”
- What do we find?
  - Generically, “documents”
  - Even though we may be referring to web pages, PDFs, PowerPoint slides, paragraphs, etc.

Information Retrieval Cycle



The Central Problem in Search

Searcher

Author

Concepts → Query Terms

"tragic love story"

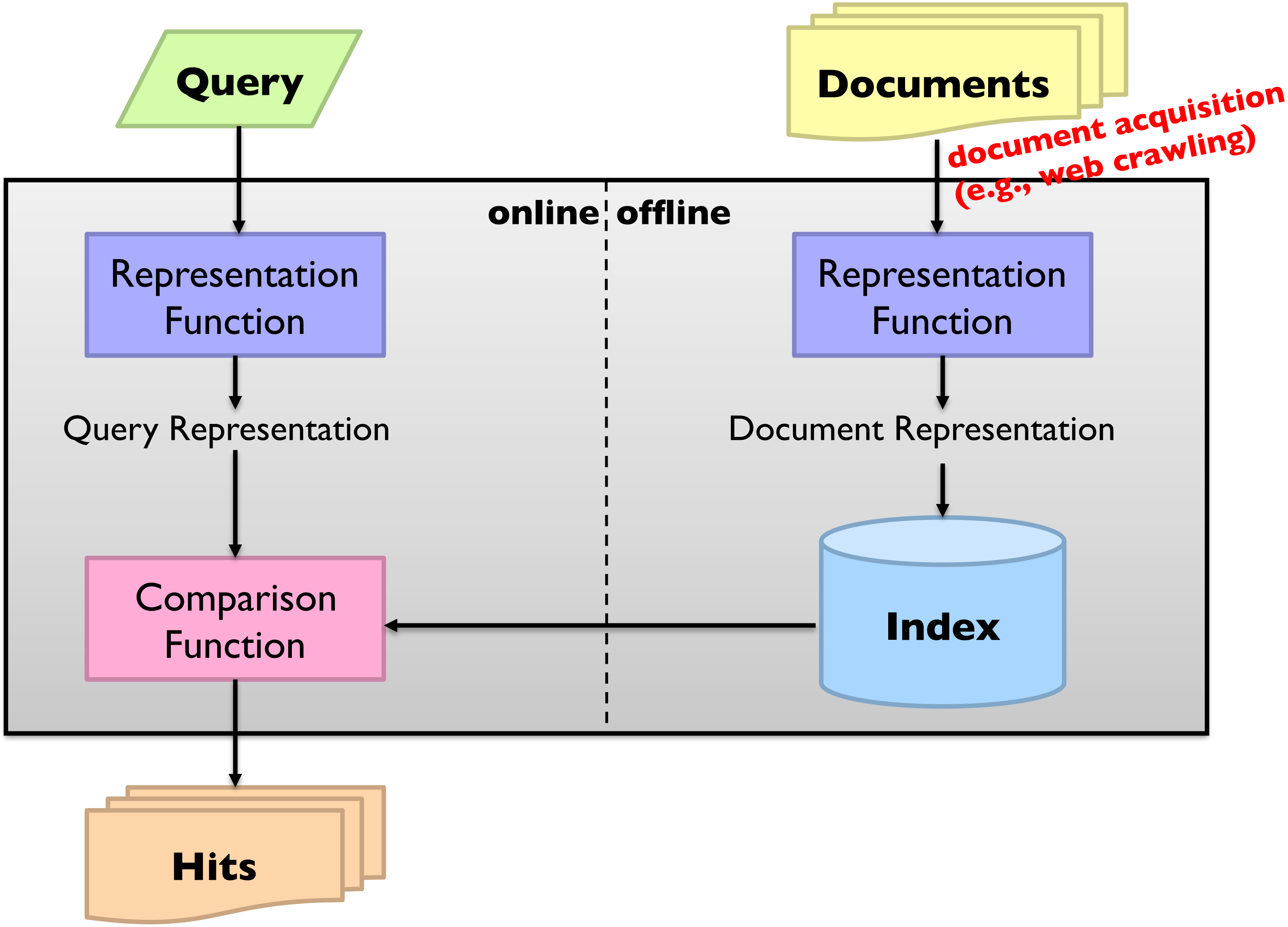
Concepts → Document Terms

"fateful star-crossed romance"

Query Terms ↔ Document Terms

Do these represent the same concepts?

Abstract IR Architecture



How do we represent text?

- Remember: computers don’t “understand” anything!
- “Bag of words”
  - Treat all the words in a document as index terms
  - Assign a “weight” to each term based on “importance” (or, in simplest case, presence/absence of word)
  - Disregard order, structure, meaning, etc. of the words
  - Simple, yet effective!
- Assumptions
  - Term occurrence is independent
  - Document relevance is independent
  - “Words” are well-defined

What’s a word?

天主教教宗若望保祿二世因感冒再度住進醫院。
這是他今年第二度因同樣的病因住院。

وقال مارك ريجيف - الناطق باسم
الخارجية الإسرائيلية - إن شارون قبل
الدعوة وسيقوم للمرة الأولى بزيارة
تونس، التي كانت لفترة طويلة المقر
الرسمي لمنظمة التحرير الفلسطينية بعد خروجها من لبنان عام 1982.

Выступая в Мещанском суде Москвы экс-глава ЮКОСа
заявил не совершал ничего противозаконного, в чем
обвиняет его генпрокуратура России.

भारत सरकार ने आर्थिक सर्वेक्षण में वित्तीय वर्ष 2005-06 में सात फ़ीसदी
विकास दर हासिल करने का आकलन किया है और कर सुधार पर ज़ोर दिया है

日米連合で台頭中国に対処…アーミテージ前副長官提言

조재영 기자= 서울시는 25일 이명박 시장이 `행정중심복합도시'' 건설안에 대해 `
군대라도 동원해 막고싶은 심정''이라고 말했다는 일부 언론의 보도를 부인했다.

Sample Document

McDonald's slims down spuds

Fast-food chain to reduce certain types of fat in its french fries with new cooking oil.

NEW YORK (CNN/Money) - McDonald's Corp. is cutting the amount of "bad" fat in its french fries nearly in half, the fast-food chain said Tuesday as it moves to make all its fried menu items healthier.

But does that mean the popular shoestring fries won't taste the same? The company says no. "It's a win-win for our customers because they are getting the same great french-fry taste along with an even healthier nutrition profile," said Mike Roberts, president of McDonald's USA.

But others are not so sure. McDonald's will not specifically discuss the kind of oil it plans to use, but at least one nutrition expert says playing with the formula could mean a different taste.

Shares of Oak Brook, Ill.-based McDonald's (MCD: down $0.54 to $23.22, Research, Estimates) were lower Tuesday afternoon. It was unclear Tuesday whether competitors Burger King and Wendy's International (WEN: down $0.80 to $34.91, Research, Estimates) would follow suit. Neither company could immediately be reached for comment.

…

"Bag of Words"

14 × McDonalds

12 × fat

11 × fries

8 × new

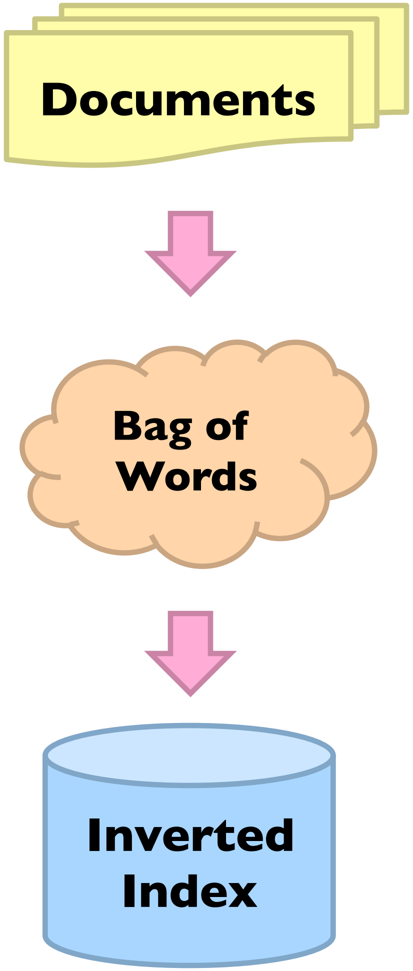
7 × french

6 × company, said, nutrition

5 × food, oil, percent, reduce, taste, Tuesday

…

Counting Words…



Documents

Bag of Words

Inverted Index

case folding, tokenization, stopword removal, stemming

~~syntax~~, ~~semantics~~, ~~word knowledge~~, etc.

Boolean Retrieval

- Users express queries as a Boolean expression
  - AND, OR, NOT
  - Can be arbitrarily nested
- Retrieval is based on the notion of sets
  - Any given query divides the collection into two sets: retrieved, not-retrieved
  - Pure Boolean systems do not define an ordering of the results

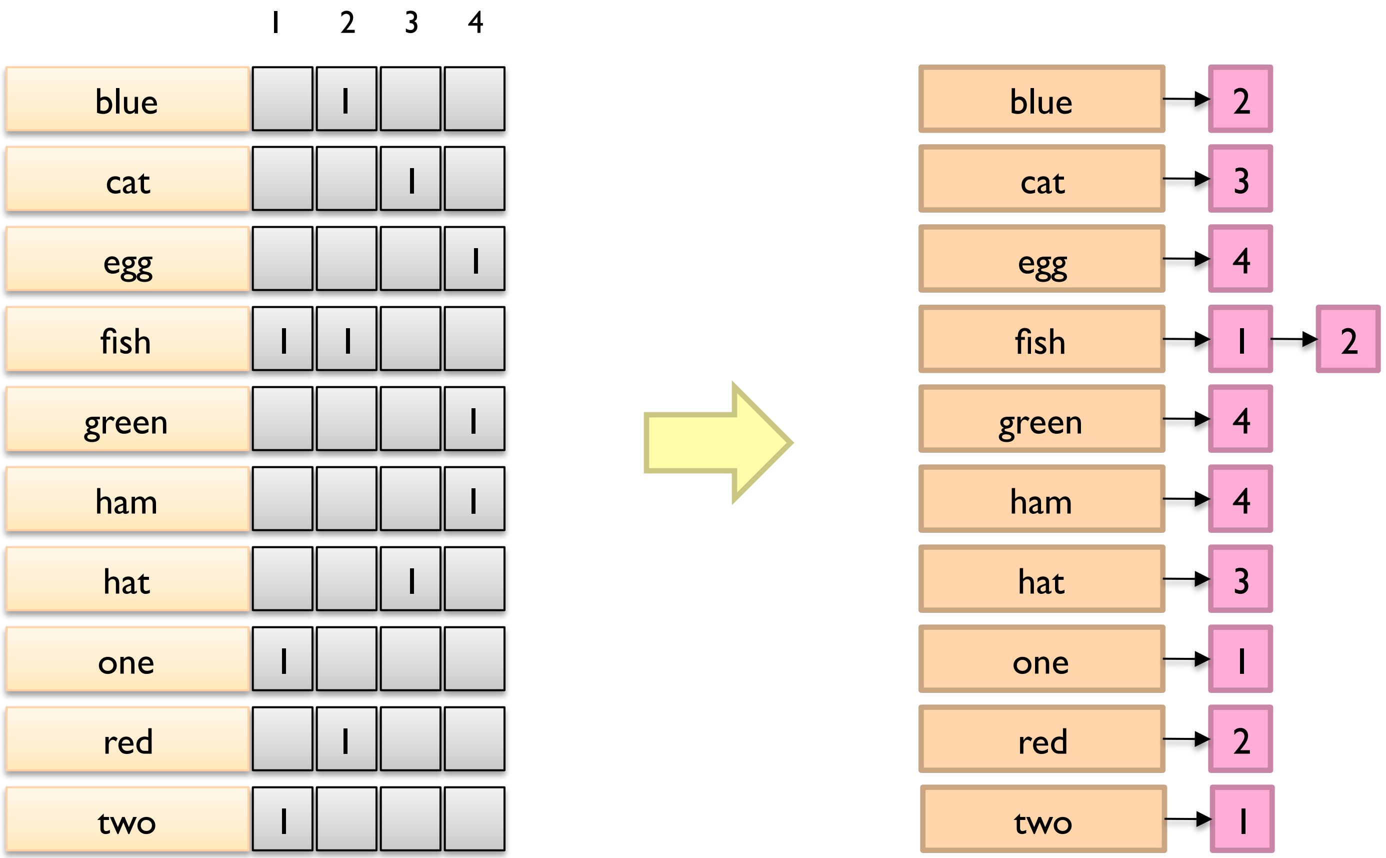
Inverted Index: Boolean Retrieval

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham



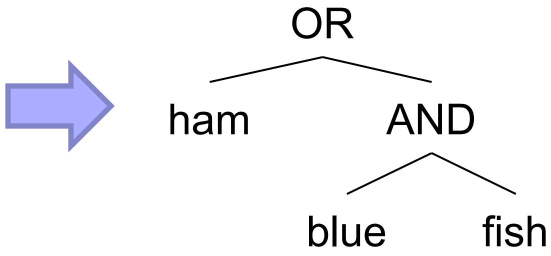
	1	2	3	4
blue		1		
cat			1	
egg				1
fish	1	1		
green				1
ham				1
hat			1	
one	1			
red		1		
two	1			

blue → 2
cat → 3
egg → 4
fish → 1 → 2
green → 4
ham → 4
hat → 3
one → 1
red → 2
two → 1

Boolean Retrieval

- To execute a Boolean query:
  - Build query syntax tree

(blue AND fish) OR ham



OR

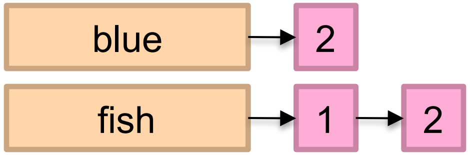
ham

AND

blue

fish

  - For each clause, look up postings



blue → 2

fish → 1 → 2

  - Traverse postings and apply Boolean operator
- Efficiency analysis
  - Postings traversal is linear (assuming sorted postings)
  - Start with shortest posting first

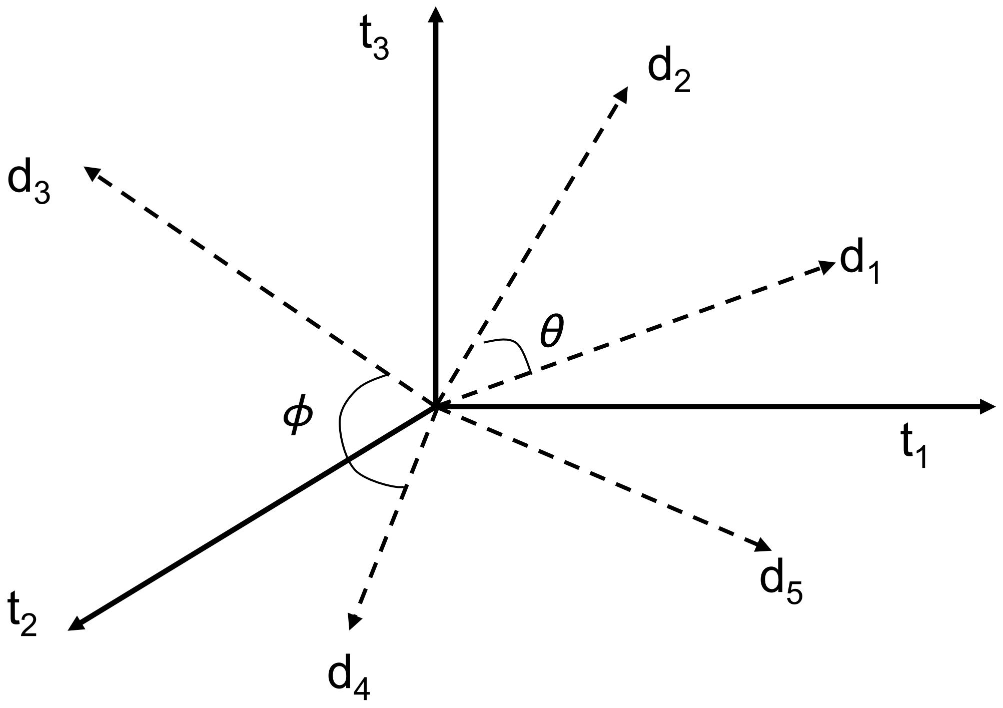
Strengths and Weaknesses

- Strengths
  - Precise, if you know the right strategies
  - Precise, if you have an idea of what you're looking for
  - Implementations are fast and efficient
- Weaknesses
  - Users must learn Boolean logic
  - Boolean logic insufficient to capture the richness of language
  - No control over size of result set: either too many hits or none
  - When do you stop reading? All documents in the result set are considered "equally good"
  - What about partial matches? Documents that "don't quite match" the query may be useful also

Ranked Retrieval

- Order documents by how likely they are to be relevant
  - Estimate relevance(q, d_i)
  - Sort documents by relevance
  - Display sorted results
- User model
  - Present hits one screen at a time, best results first
  - At any point, users can decide to stop looking
- How do we estimate relevance?
  - Assume document is relevant if it has a lot of query terms
  - Replace relevance(q, d_i) with sim(q, d_i)
  - Compute similarity of vector representations

Vector Space Model



Assumption: Documents that are "close together" in vector space "talk about" the same things

Therefore, retrieve documents based on how close the document is to the query (i.e., similarity ~ "closeness")

Similarity Metric

- Use "angle" between the vectors:

$$d_j = [w_{j,1}, w_{j,2}, w_{j,3}, \dots w_{j,n}]$$
$$d_k = [w_{k,1}, w_{k,2}, w_{k,3}, \dots w_{k,n}]$$

$$\cos\theta = \frac{d_j \cdot d_k}{|d_j||d_k|}$$

$$\mathrm{sim}(d_j, d_k) = \frac{d_j \cdot d_k}{|d_j||d_k|} = \frac{\sum_{i=0}^{n} w_{j,i} w_{k,i}}{\sqrt{\sum_{i=0}^{n} w_{j,i}^2}\sqrt{\sum_{i=0}^{n} w_{k,i}^2}}$$

- Or, more generally, inner products:

$$\mathrm{sim}(d_j, d_k) = d_j \cdot d_k = \sum_{i=0}^{n} w_{j,i} w_{k,i}$$

Term Weighting

- Term weights consist of two components
  - Local: how important is the term in this document?
  - Global: how important is the term in the collection?
- Here’s the intuition:
  - Terms that appear often in a document should get high weights
  - Terms that appear in many documents should get low weights
- How do we capture this mathematically?
  - Term frequency (local)
  - Inverse document frequency (global)

TF.IDF Term Weighting

$$w_{i,j} = \mathrm{tf}_{i,j} \cdot \log \frac{N}{n_i}$$

$w_{i,j}$	weight assigned to term *i* in document *j*
$\mathrm{tf}_{i,j}$	number of occurrence of term *i* in document *j*
N	number of documents in entire collection
n_i	number of documents with term *i*

Inverted Index: TF.IDF

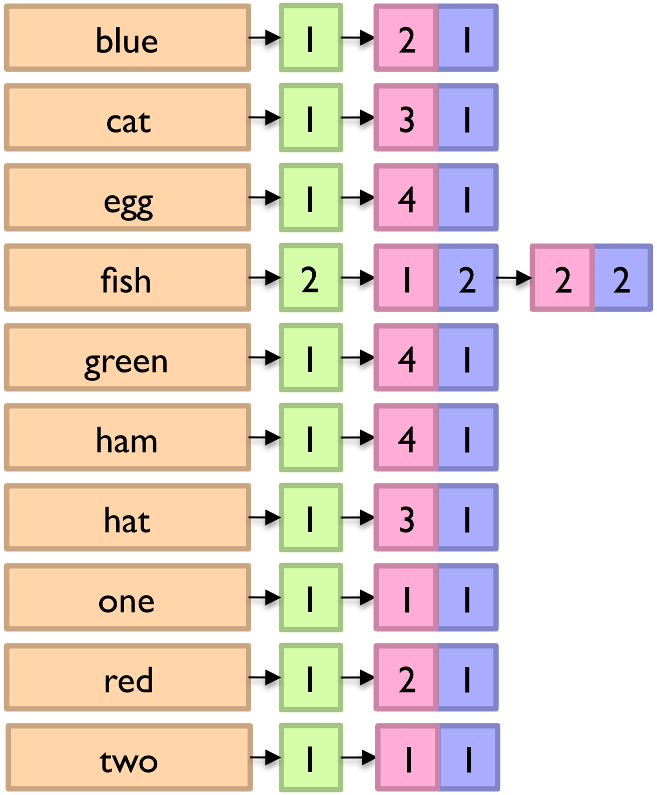
Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham

	tf				
	1	2	3	4	*df*
blue		1			1
cat			1		1
egg				1	1
fish	2	2			2
green				1	1
ham				1	1
hat			1		1
one	1				1
red		1			1
two	1				1



blue → 1 → 2 1
cat → 1 → 3 1
egg → 1 → 4 1
fish → 2 → 1 2 → 2 2
green → 1 → 4 1
ham → 1 → 4 1
hat → 1 → 3 1
one → 1 → 1 1
red → 1 → 2 1
two → 1 → 1 1

Positional Indexes

- Store term position in postings
- Supports richer queries (e.g., proximity)
- Naturally, leads to larger indexes…

Inverted Index: Positional Information

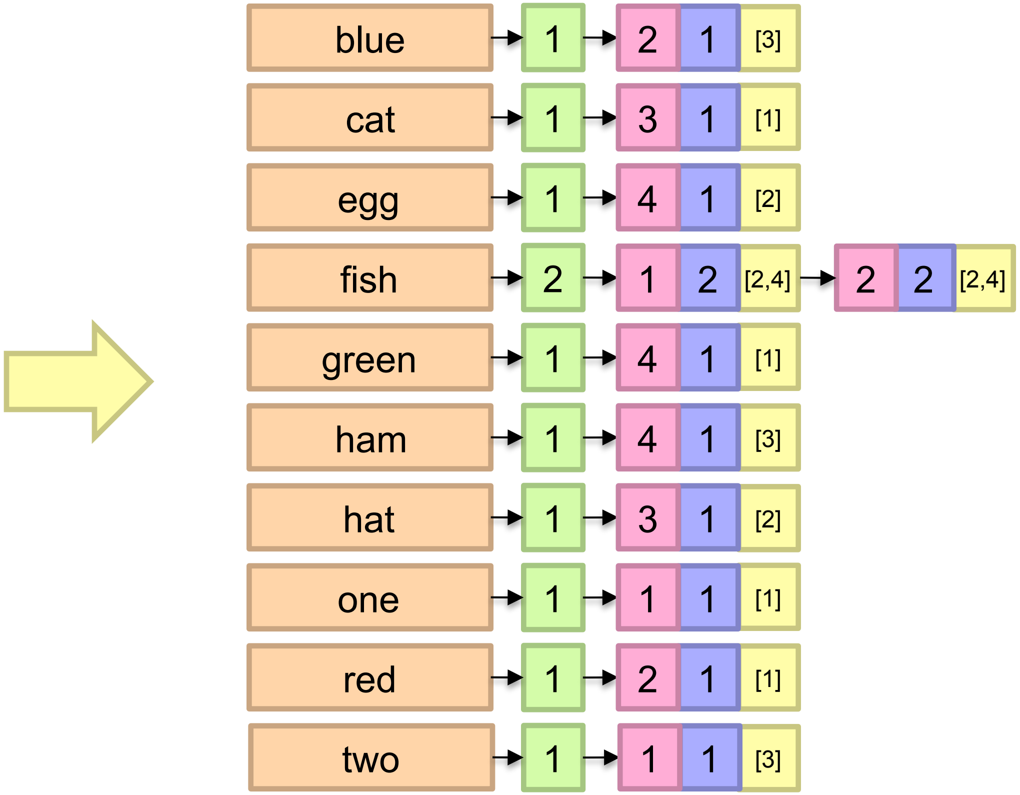
Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham

	tf				
	1	2	3	4	*df*
blue		1			1
cat			1		1
egg				1	1
fish	2	2			2
green				1	1
ham				1	1
hat			1		1
one	1				1
red		1			1
two	1				1



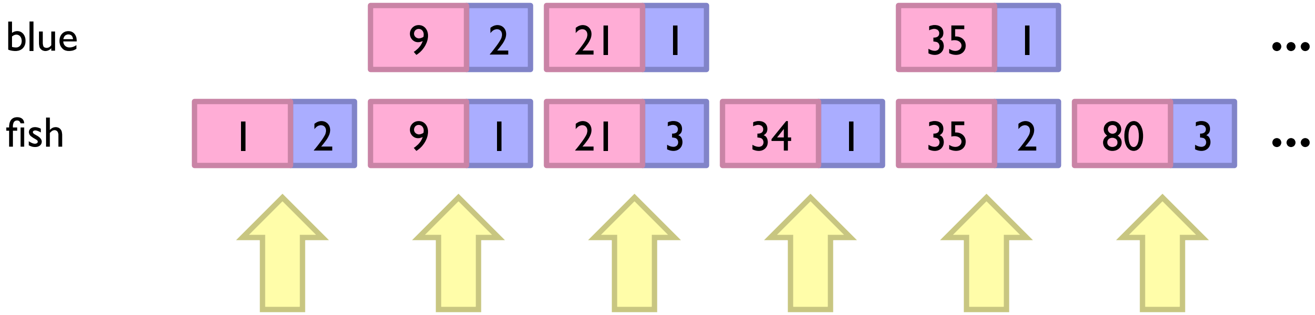
blue	1	2	1	[3]			
cat	1	3	1	[1]			
egg	1	4	1	[2]			
fish	2	1	2	[2,4]	2	2	[2,4]
green	1	4	1	[1]			
ham	1	4	1	[3]			
hat	1	3	1	[2]			
one	1	1	1	[1]			
red	1	2	1	[1]			
two	1	1	1	[3]			

Retrieval in a Nutshell

- Look up postings lists corresponding to query terms
- Traverse postings for each query term
- Store partial query-document scores in accumulators
- Select top *k* results to return

Retrieval: Document-at-a-Time

- Evaluate documents one at a time (score all query terms)



blue		9 2	21 1		35 1		…
fish	1 2	9 1	21 3	34 1	35 2	80 3	…

Accumulators
(e.g. min heap)

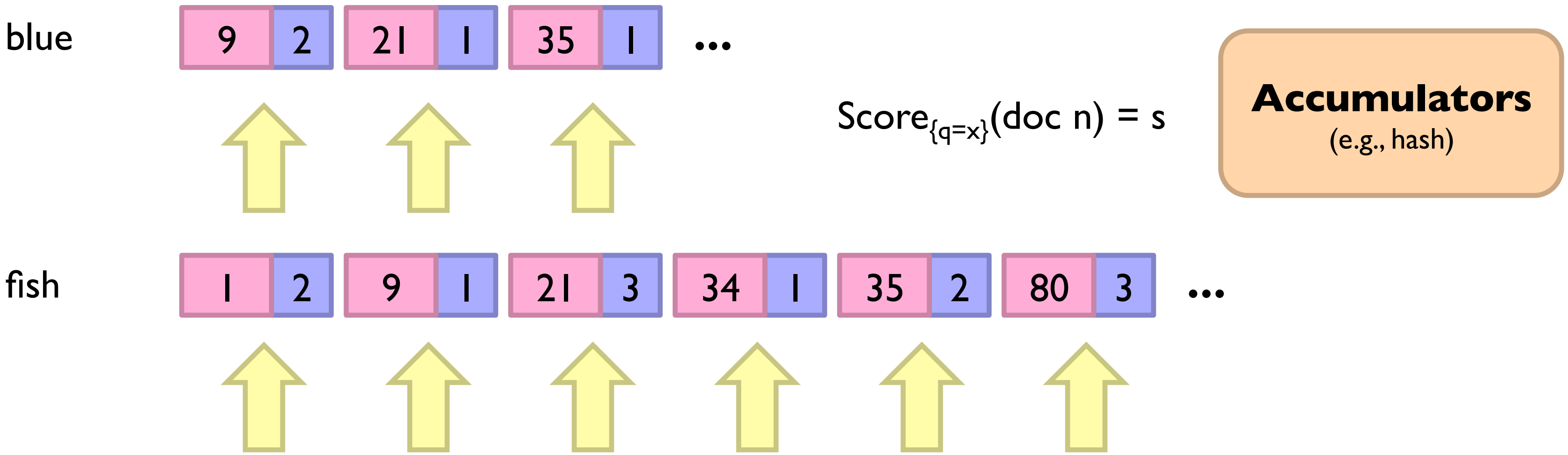
Document score in top k?

Yes: Insert document score, extract-min if heap too large
No: Do nothing

- Tradeoffs
  - Small memory footprint (good)
  - Skipping possible to avoid reading all postings (good)
  - More seeks and irregular data accesses (bad)

Retrieval: Query-At-A-Time

- Evaluate documents one query term at a time
  - Usually, starting from most rare term (often with tf-sorted postings)



blue	9	2	21	1	35	1	…						
fish	1	2	9	1	21	3	34	1	35	2	80	3	…

Score$_{\{q=x\}}$(doc n) = s

Accumulators
(e.g., hash)

- Tradeoffs
  - Early termination heuristics (good)
  - Large memory footprint (bad), but filtering heuristics possible

MapReduce it?

- The indexing problem
  - Scalability is critical
  - Must be relatively fast, but need not be real time
  - Fundamentally a batch operation
  - Incremental updates may or may not be important
  - For the web, crawling is a challenge in itself

Perfect for MapReduce!

- The retrieval problem
  - Must have sub-second response time
  - For the web, only need relatively few results

Uh... not so good...

Indexing: Performance Analysis

- Fundamentally, a large sorting problem
  - Terms usually fit in memory
  - Postings usually don't
- How is it done on a single machine?
- How can it be done with MapReduce?
- First, let's characterize the problem size:
  - Size of vocabulary
  - Size of postings

Vocabulary Size: Heaps' Law

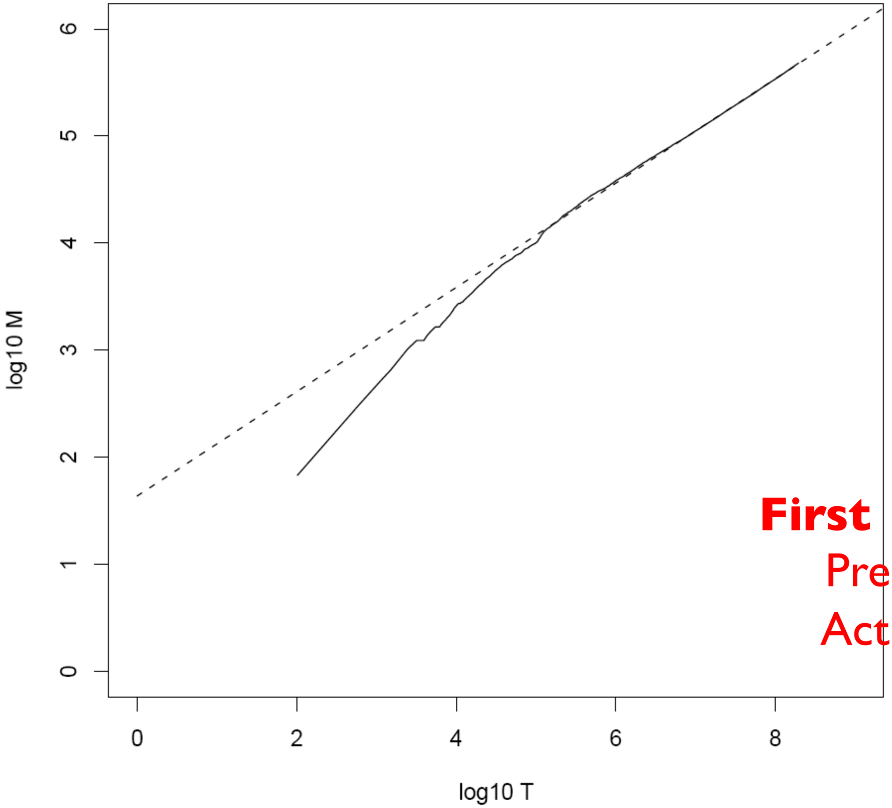
$$M = kT^b$$

M is vocabulary size
T is collection size (number of documents)
k and *b* are constants

Typically, *k* is between 30 and 100, *b* is between 0.4 and 0.6

- Heaps' Law: linear in log-log space
- Vocabulary size grows unbounded!

Heaps’ Law for RCV1



k = 44
b = 0.49

First 1,000,020 terms:
Predicted = 38,323
Actual = 38,365

Reuters-RCV1 collection: 806,791 newswire documents (Aug 20, 1996-August 19, 1997)

Manning, Raghavan, Schütze, Introduction to Information Retrieval (2008)

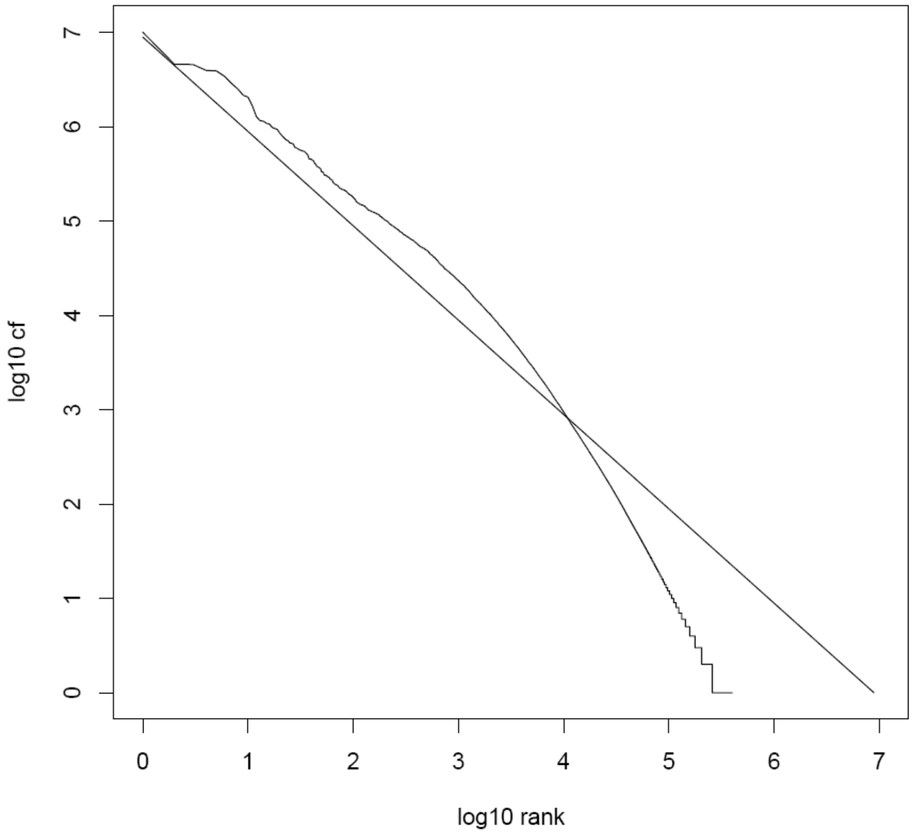
Postings Size: Zipf’s Law

$$\mathrm{cf}_i = \frac{c}{i}$$

cf is the collection frequency of *i*-th common term
c is a constant

- Zipf’s Law: (also) linear in log-log space
  - Specific case of Power Law distributions
- In other words:
  - A few elements occur very frequently
  - Many elements occur very infrequently

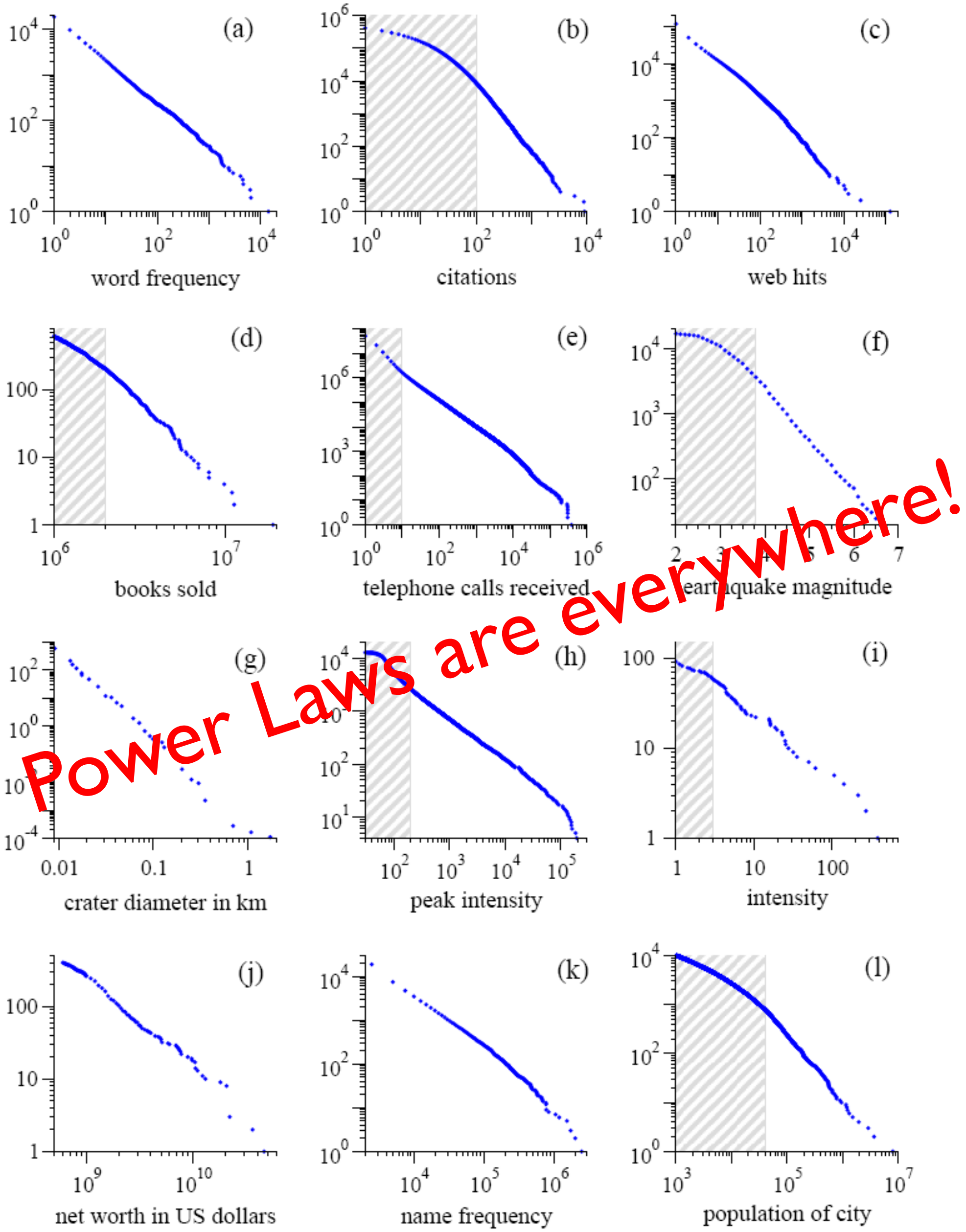
Zipf’s Law for RCV1



Fit isn’t that good… but good enough!

Reuters-RCV1 collection: 806,791 newswire documents (Aug 20, 1996-August 19, 1997)

Manning, Raghavan, Schütze, Introduction to Information Retrieval (2008)



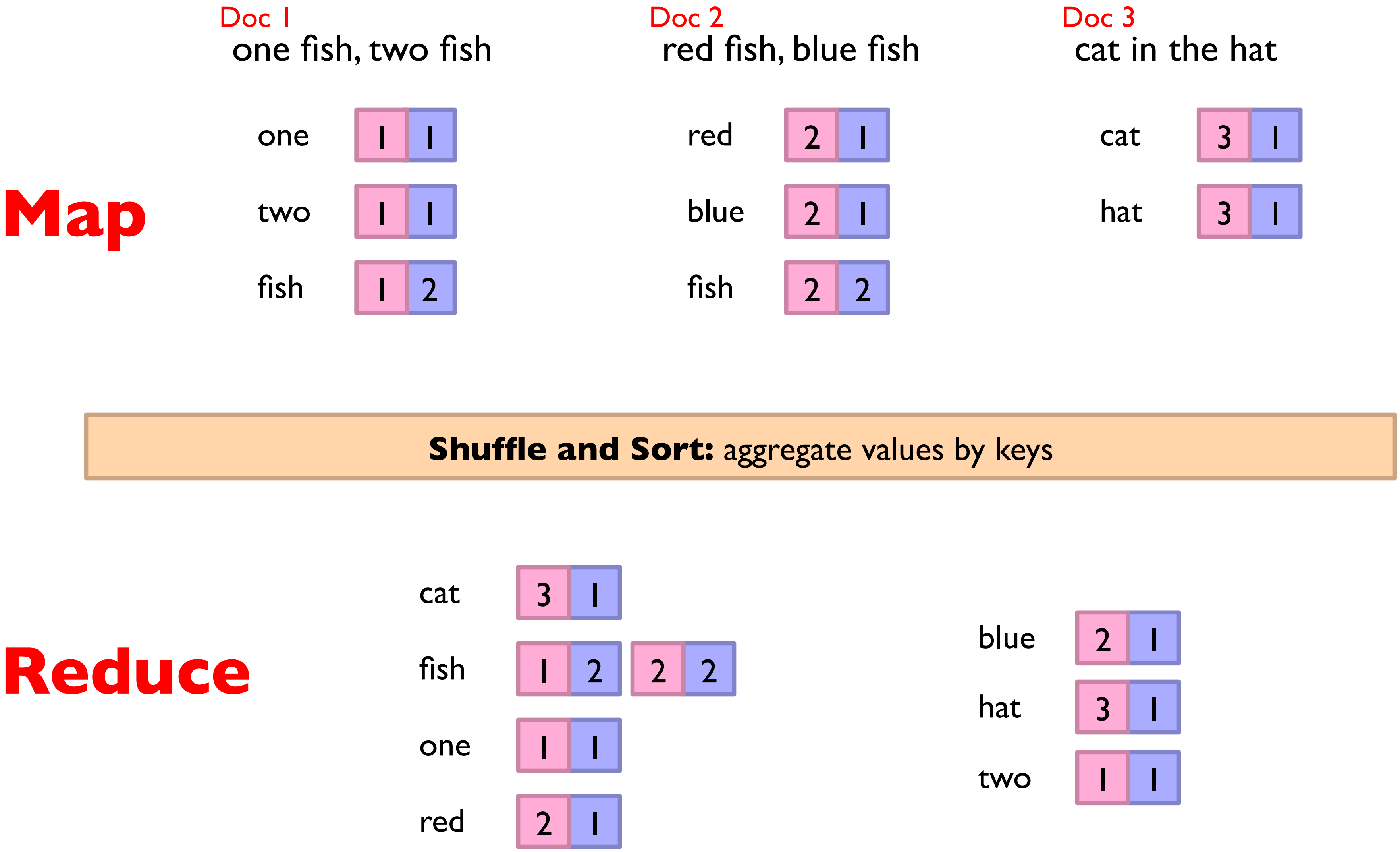
Power Laws are everywhere!

Figure from: Newman, M. E. J. (2005) “Power laws, Pareto distributions and Zipf's law.” Contemporary Physics 46:323–351.

MapReduce: Index Construction

- Map over all documents
  - Emit *term* as key, (*docno*, *tf*) as value
  - Emit other information as necessary (e.g., term position)
- Sort/shuffle: group postings by term
- Reduce
  - Gather and sort the postings (e.g., by *docno* or *tf*)
  - Write postings to disk
- MapReduce does all the heavy lifting!

Inverted Indexing with MapReduce



Map

Doc 1
one fish, two fish

one 1 1

two 1 1

fish 1 2

Doc 2
red fish, blue fish

red 2 1

blue 2 1

fish 2 2

Doc 3
cat in the hat

cat 3 1

hat 3 1

Shuffle and Sort: aggregate values by keys

Reduce

cat 3 1

fish 1 2 2 2

one 1 1

red 2 1

blue 2 1

hat 3 1

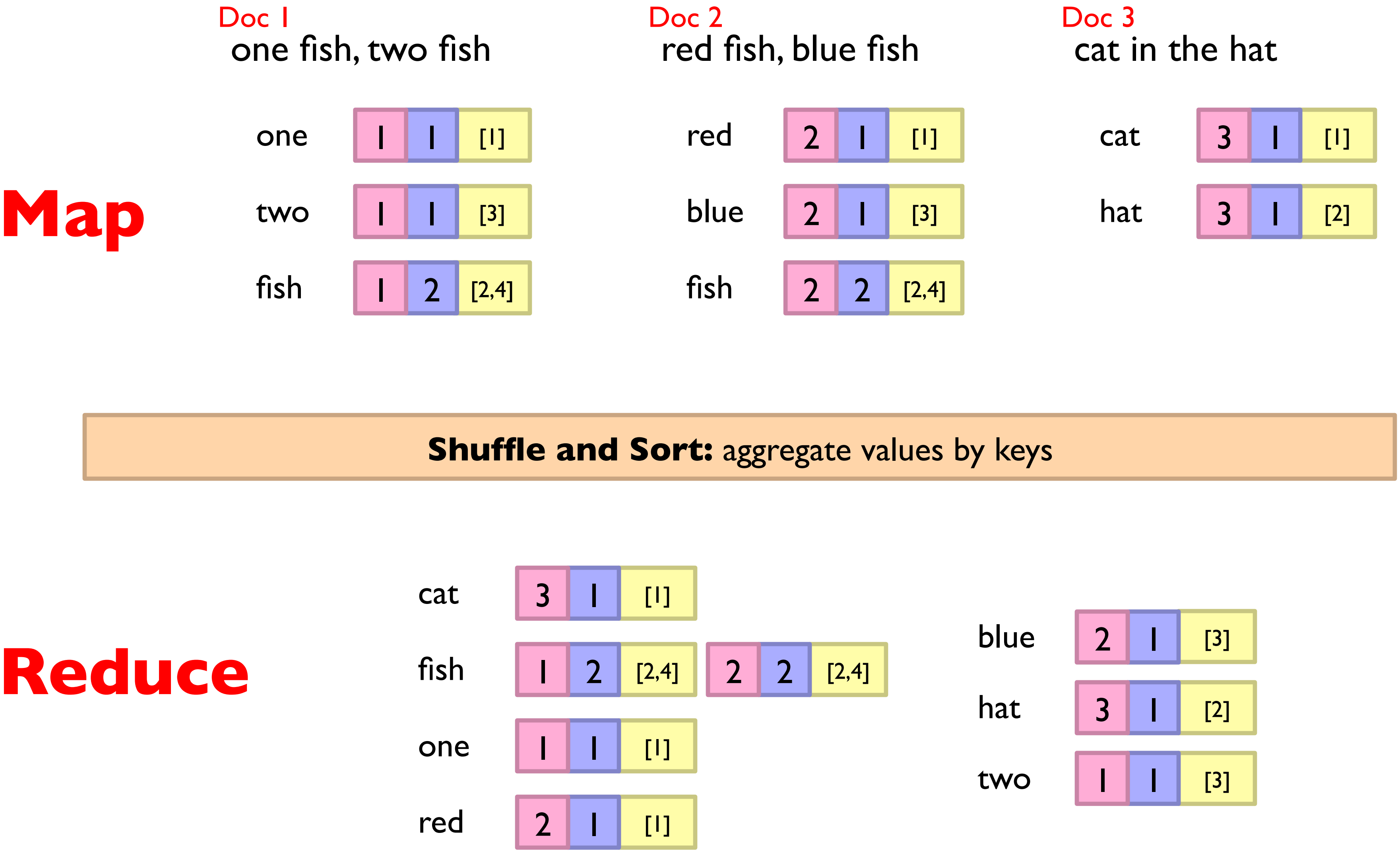
two 1 1

Inverted Indexing: Pseudo-Code

```
class Mapper
    method Map(docid $n$, doc $d$)
        $H \leftarrow$ new AssociativeArray                ▷ histogram to hold term frequencies
        for all term $t \in$ doc $d$ do       ▷ processes the doc, e.g., tokenization and stopword removal
            $H\{t\} \leftarrow H\{t\} + 1$
        for all term $t \in H$ do
            Emit(term $t$, posting $\langle n, H\{t\} \rangle$)          ▷ emits individual postings
```

```
class Reducer
    method Reduce(term $t$, postings $[\langle n_1, f_1 \rangle \ldots]$)
        $P \leftarrow$ new List
        for all $\langle n, f \rangle \in$ postings $[\langle n_1, f_1 \rangle \ldots]$ do
            $P$.Append($\langle n, f \rangle$)                  ▷ appends postings unsorted
        $P$.Sort()                                       ▷ sorts for compression
        Emit(term $t$, postingsList $P$)
```

Positional Indexes



Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Map

one	1	1	[1]
two	1	1	[3]
fish	1	2	[2,4]

red	2	1	[1]
blue	2	1	[3]
fish	2	2	[2,4]

cat	3	1	[1]
hat	3	1	[2]

Shuffle and Sort: aggregate values by keys

Reduce

cat	3	1	[1]			
fish	1	2	[2,4]	2	2	[2,4]
one	1	1	[1]			
red	2	1	[1]			

blue	2	1	[3]
hat	3	1	[2]
two	1	1	[3]

Inverted Indexing: Pseudo-Code

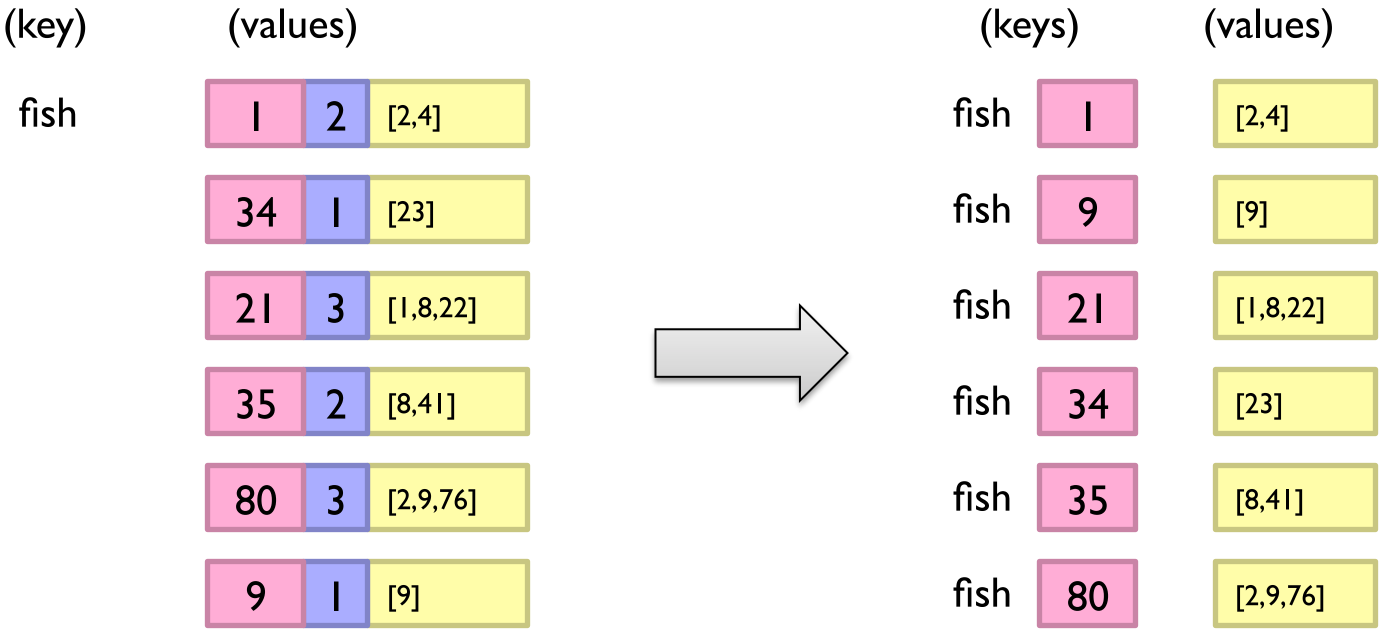
```
class Mapper
    method Map(docid $n$, doc $d$)
        $H \leftarrow$ new AssociativeArray                ▷ histogram to hold term frequencies
        for all term $t \in$ doc $d$ do      ▷ processes the doc, e.g., tokenization and stopword removal
            $H\{t\} \leftarrow H\{t\} + 1$
        for all term $t \in H$ do
            Emit(term $t$, posting $\langle n, H\{t\} \rangle$)            ▷ emits individual postings
class Reducer
    method Reduce(term $t$, postings $[\langle n_1, f_1 \rangle \ldots]$)
        $P \leftarrow$ new List
        for all $\langle n, f \rangle \in$ postings $[\langle n_1, f_1 \rangle \ldots]$ do
            $P$.Append($\langle n, f \rangle$)                ▷ appends postings unsorted
        $P$.Sort()                                          ▷ sorts for compression
        Emit(term $t$, postingsList $P$)
```

What's the problem?

Scalability Bottleneck

- Initial implementation: terms as keys, postings as values
  - Reducers must buffer all postings associated with key (to sort)
  - What if we run out of memory to buffer postings?
- Uh oh!

Another Try…



(key)	(values)		
fish	1	2	[2,4]
	34	1	[23]
	21	3	[1,8,22]
	35	2	[8,41]
	80	3	[2,9,76]
	9	1	[9]

(keys)		(values)
fish	1	[2,4]
fish	9	[9]
fish	21	[1,8,22]
fish	34	[23]
fish	35	[8,41]
fish	80	[2,9,76]

How is this different?

- Let the framework do the sorting
- Term frequency implicitly stored

Where have we seen this before?

Inverted Indexing: Pseudo-Code

```
class Mapper
    method Map(docid $n$, doc $d$)
        $H \leftarrow$ new AssociativeArray
        for all term $t \in$ doc $d$ do                         ▷ builds a histogram of term frequencies
            $H\{t\} \leftarrow H\{t\} + 1$
        for all term $t \in H$ do
            Emit(tuple $\langle t, n \rangle$, tf $H\{t\}$)      ▷ emits individual postings, with a tuple as the key
class Partitioner
    method Partition(tuple $\langle t, n \rangle$, tf $f$)
        return Hash($t$) mod $NumOfReducers$                     ▷ keys of same term are sent to same reducer
class Reducer
    method Initialize
        $t_{prev} \leftarrow \emptyset$
        $P \leftarrow$ new PostingsList
    method Reduce(tuple $\langle t, n \rangle$, tf $[f]$)
        if $t \neq t_{prev} \wedge t_{prev} \neq \emptyset$ then
            Emit(term $t$, postings $P$)                          ▷ emits postings list of term $t_{prev}$
            $P$.Reset()
        $P$.Append($\langle n, f \rangle$)                        ▷ appends postings in sorted order
        $t_{prev} \leftarrow t$
    method Close
        Emit(term $t$, postings $P$)                              ▷ emits last postings list from this reducer
```

Inverted Index (Again)

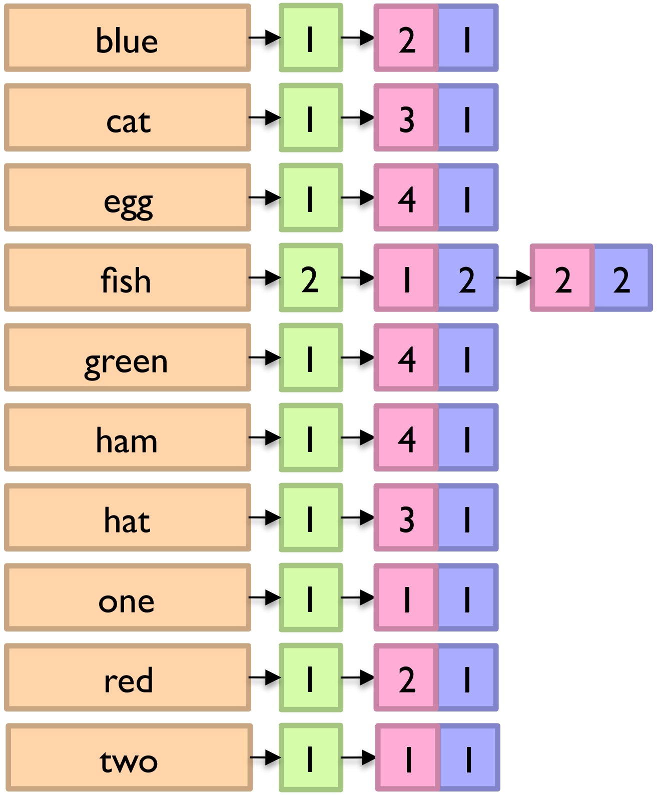
Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

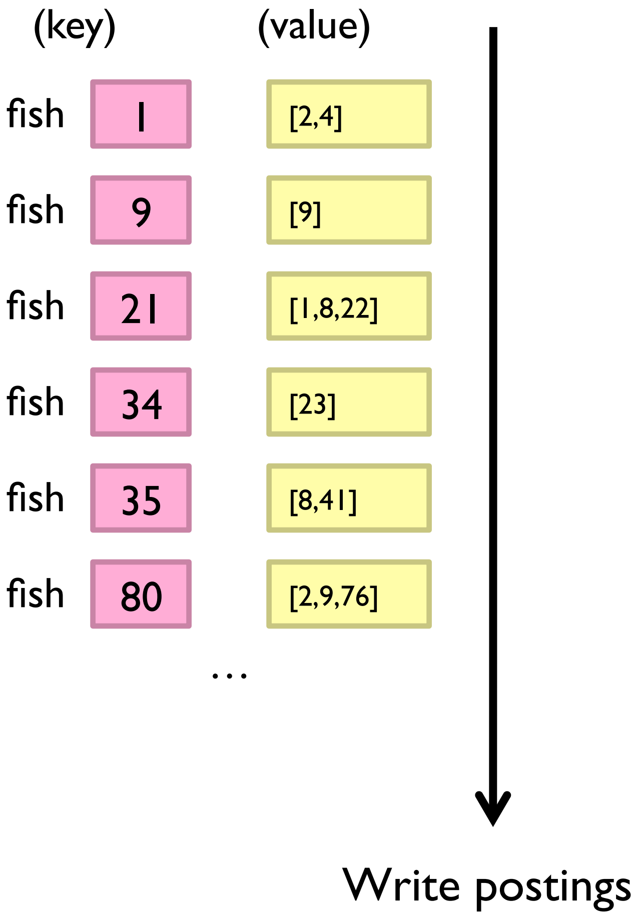
Doc 4
green eggs and ham

	tf				
	1	2	3	4	*df*
blue		1			1
cat			1		1
egg				1	1
fish	2	2			2
green				1	1
ham				1	1
hat			1		1
one	1				1
red		1			1
two	1				1



blue → 1 → 2 1
cat → 1 → 3 1
egg → 1 → 4 1
fish → 2 → 1 2 → 2 2
green → 1 → 4 1
ham → 1 → 4 1
hat → 1 → 3 1
one → 1 → 1 1
red → 1 → 2 1
two → 1 → 1 1

Chicken and Egg?



	(key)	(value)
fish	1	[2,4]
fish	9	[9]
fish	21	[1,8,22]
fish	34	[23]
fish	35	[8,41]
fish	80	[2,9,76]
	…	

Write postings

We'd like to store the *df* at the front of the postings list

But we don't know the *df* until we've seen all postings!

Sound familiar?

Getting the *df*

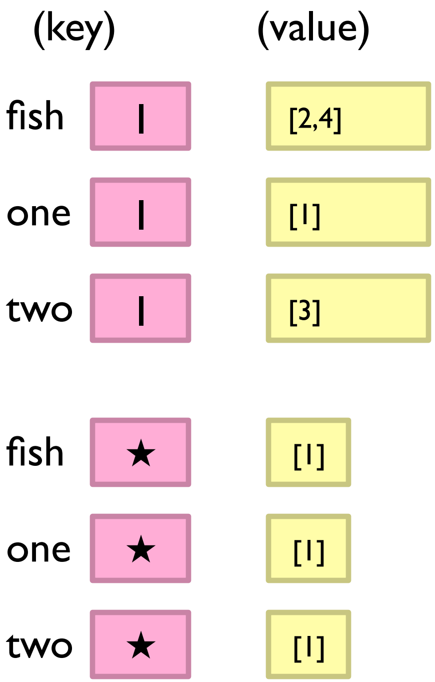
- In the mapper:
  - Emit “special” key-value pairs to keep track of *df*
- In the reducer:
  - Make sure “special” key-value pairs come first: process them to determine *df*
- Remember: proper partitioning!

Getting the *df*: Modified Mapper

Doc 1

one fish, two fish

Input document…

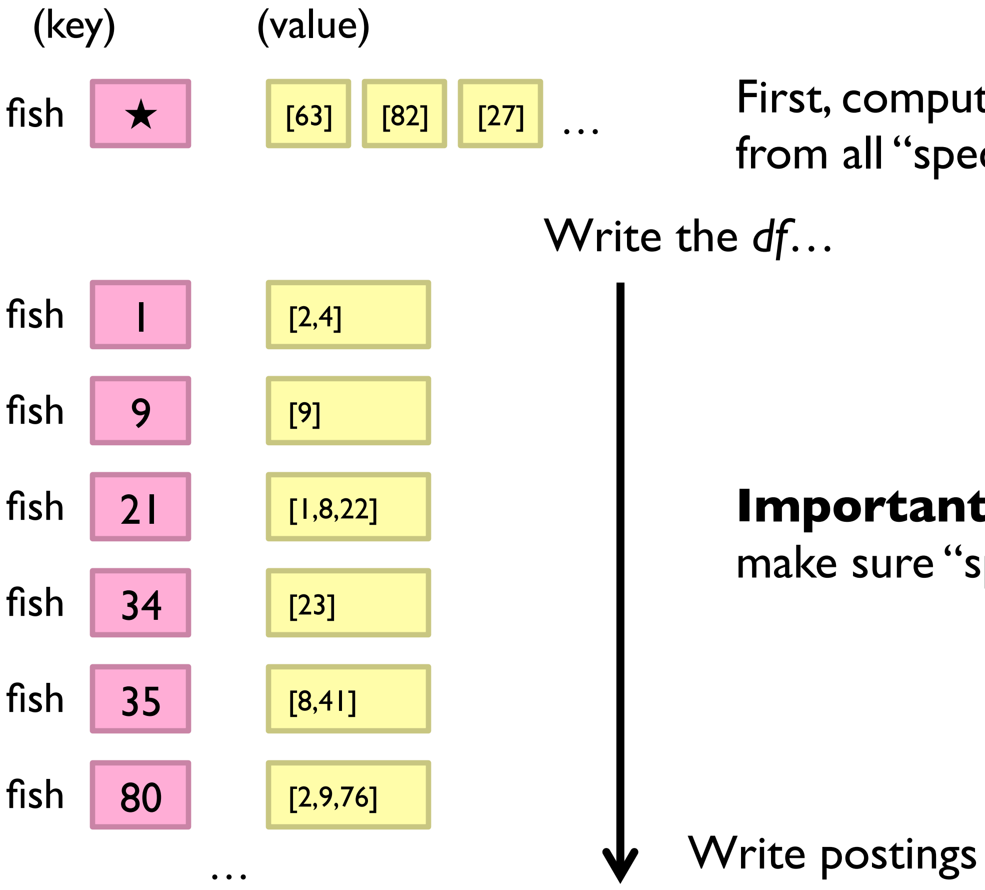


	(key)	(value)
fish	1	[2,4]
one	1	[1]
two	1	[3]
fish	★	[1]
one	★	[1]
two	★	[1]

Emit normal key-value pairs…

Emit “special” key-value pairs to keep track of *df*…

Getting the *df*: Modified Reducer



	(key)	(value)
fish	★	[63] [82] [27] …
fish	1	[2,4]
fish	9	[9]
fish	21	[1,8,22]
fish	34	[23]
fish	35	[8,41]
fish	80	[2,9,76]
	…	

Write the *df*…

Write postings

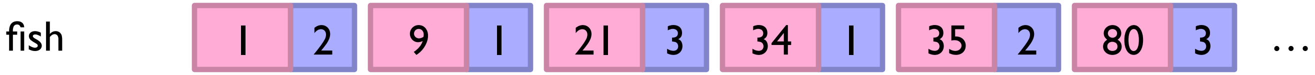
First, compute the *df* by summing contributions from all "special" key-value pair…

Important: properly define sort order to make sure "special" key-value pairs come first!

Where have we seen this before?

Postings Encoding

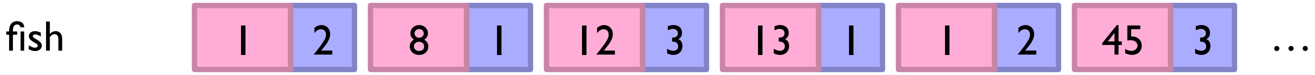
Conceptually:



fish 1 2 | 9 1 | 21 3 | 34 1 | 35 2 | 80 3 | …

In Practice:

- Don’t encode docnos, encode gaps (or *d*-gaps)
- But it’s not obvious that this save space…



fish 1 2 | 8 1 | 12 3 | 13 1 | 1 2 | 45 3 | …

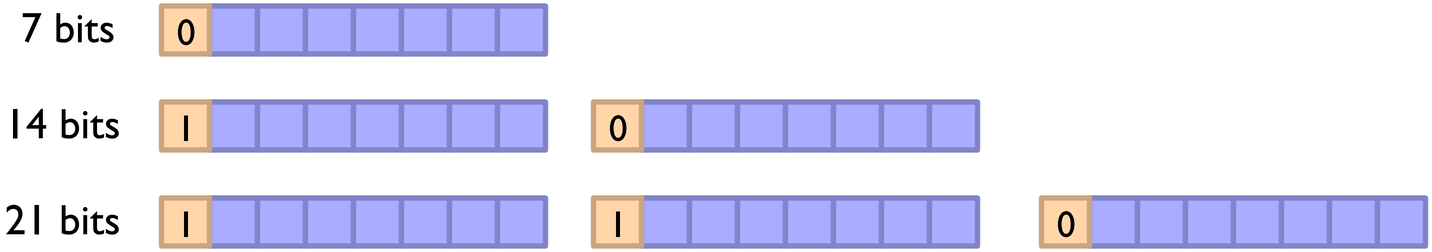
Overview of Index Compression

- Byte-aligned vs. bit-aligned
- Byte-aligned technique
    - VByte
    - Simple9 and variants
    - PForDelta
- Bit-aligned
    - Unary codes
    - γ codes
    - δ codes
    - Golomb codes (local Bernoulli model)

Want more detail? Read *Managing Gigabytes* by Witten, Moffat, and Bell!

VByte

- Simple idea: use only as many bytes as needed
  - Need to reserve one bit per byte as the “continuation bit”
  - Use remaining bits for encoding value



- Works okay, easy to implement…

Inverted Indexing: IP

```
class Mapper
    method Map(docid $n$, doc $d$)
        $H \leftarrow$ new AssociativeArray
        for all term $t \in$ doc $d$ do                ▷ builds a histogram of term frequencies
            $H\{t\} \leftarrow H\{t\} + 1$
        for all term $t \in H$ do
            Emit(tuple $\langle t, n\rangle$, tf $H\{t\}$)        ▷ emits individual postings, with a tuple as the key

class Partitioner
    method Partition(tuple $\langle t, n\rangle$, tf $f$)
        return Hash($t$) mod $NumOfReducers$        ▷ keys of same term are sent to same reducer

class Reducer
    method Initialize
        $t_{prev} \leftarrow \emptyset$
        $P \leftarrow$ new PostingsList
    method Reduce(tuple $\langle t, n\rangle$, tf $[f]$)
        if $t \neq t_{prev} \wedge t_{prev} \neq \emptyset$ then
            Emit(term $t$, postings $P$)        ▷ emits postings list of term $t_{prev}$
            $P$.Reset()
        $P$.Append($\langle n, f\rangle$)        ▷ appends postings in sorted order
        $t_{prev} \leftarrow t$
    method Close
        Emit(term $t$, postings $P$)        ▷ emits last postings list from this reducer
```

What's the assumption?

Inverted Indexing: LP

```
class Mapper
    method Initialize
        $M \leftarrow$ new AssociativeArray                         ▷ holds partial lists of postings
    method Map(docid $n$, doc $d$)
        $H \leftarrow$ new AssociativeArray                 ▷ builds a histogram of term frequencies
        for all term $t \in$ doc $d$ do
            $H\{t\} \leftarrow H\{t\} + 1$
        for all term $t \in H$ do
            $M\{t\}$.Add(posting $\langle n, H\{t\}\rangle$)       ▷ adds a posting to partial postings lists
        if MemoryFull() then
            Flush()
    method Flush                   ▷ flushes partial lists of postings as intermediate output
        for all term $t \in M$ do
            $P \leftarrow$ SortAndEncodePostings($M\{t\}$)
            Emit(term $t$, postingsList $P$)
        $M$.Clear()
    method Close
        Flush()
```

Inverted Indexing: LP

```
class Reducer
    method Reduce(term $t$, postingsLists $[P_1, P_2, \ldots]$)
        $P_f \leftarrow$ new List                        ▷ temporarily stores partial lists of postings
        $R \leftarrow$ new List                          ▷ stores merged partial lists of postings
        for all $P \in$ postingsLists $[P_1, P_2, \ldots]$ do
            $P_f$.Add($P$)
            if MemoryNearlyFull() then
                $R$.Add(MergeLists($P_f$))
                $P_f$.Clear()
        $R$.Add(MergeLists($P_f$))
        Emit(term $t$, postingsList MergeLists($R$))     ▷ emits fully merged postings list of term $t$
```

MapReduce it?

- The indexing problem ← **Just covered**
  - Scalability is paramount
  - Must be relatively fast, but need not be real time
  - Fundamentally a batch operation
  - Incremental updates may or may not be important
  - For the web, crawling is a challenge in itself
- The retrieval problem ← **Now**
  - Must have sub-second response time
  - For the web, only need relatively few results

Retrieval with MapReduce?

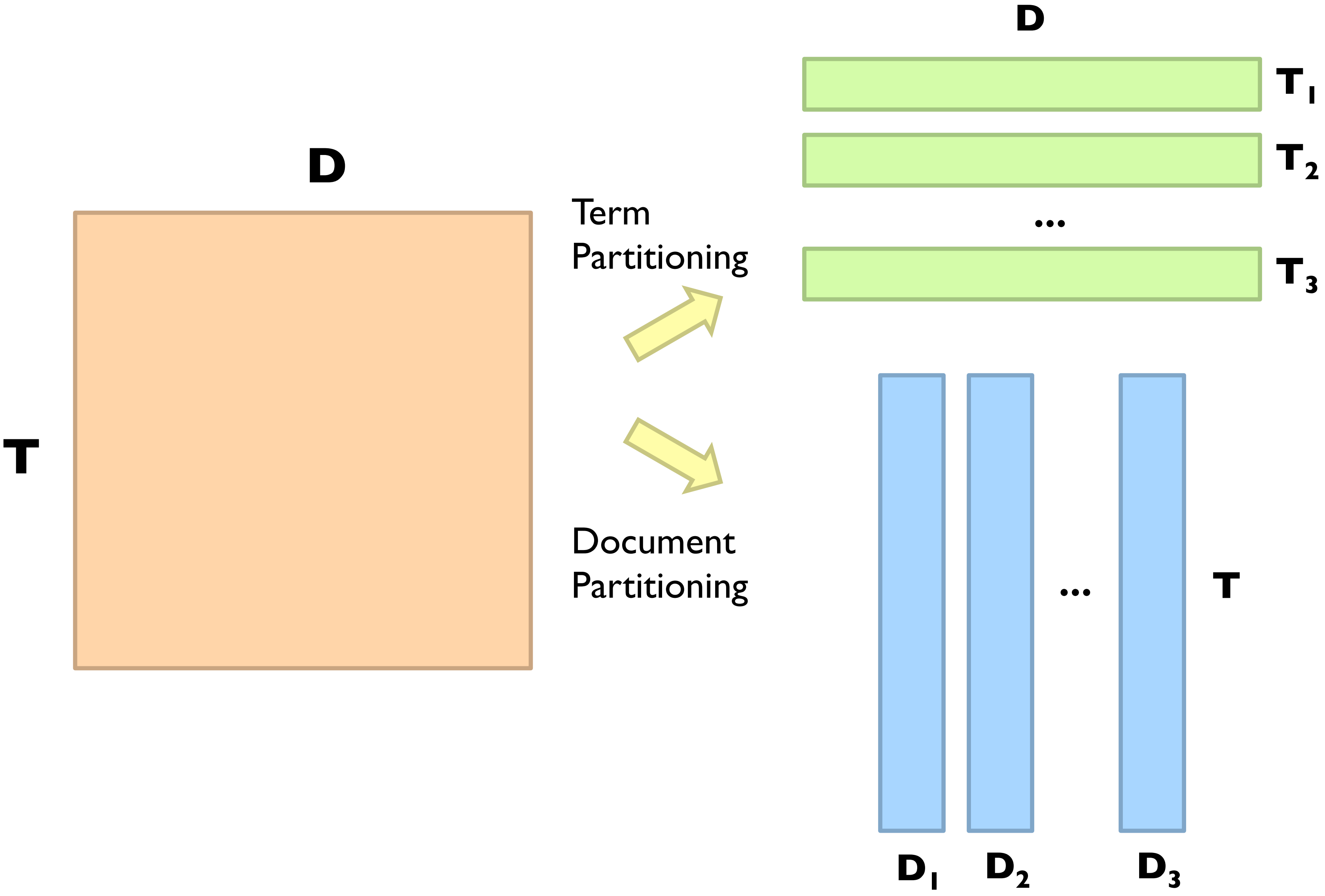
- MapReduce is fundamentally batch-oriented
  - Optimized for throughput, not latency
  - Startup of mappers and reducers is expensive
- MapReduce is not suitable for real-time queries!
  - Use separate infrastructure for retrieval…

Important Ideas

- Partitioning (for scalability)
- Replication (for redundancy)
- Caching (for speed)
- Routing (for load balancing)

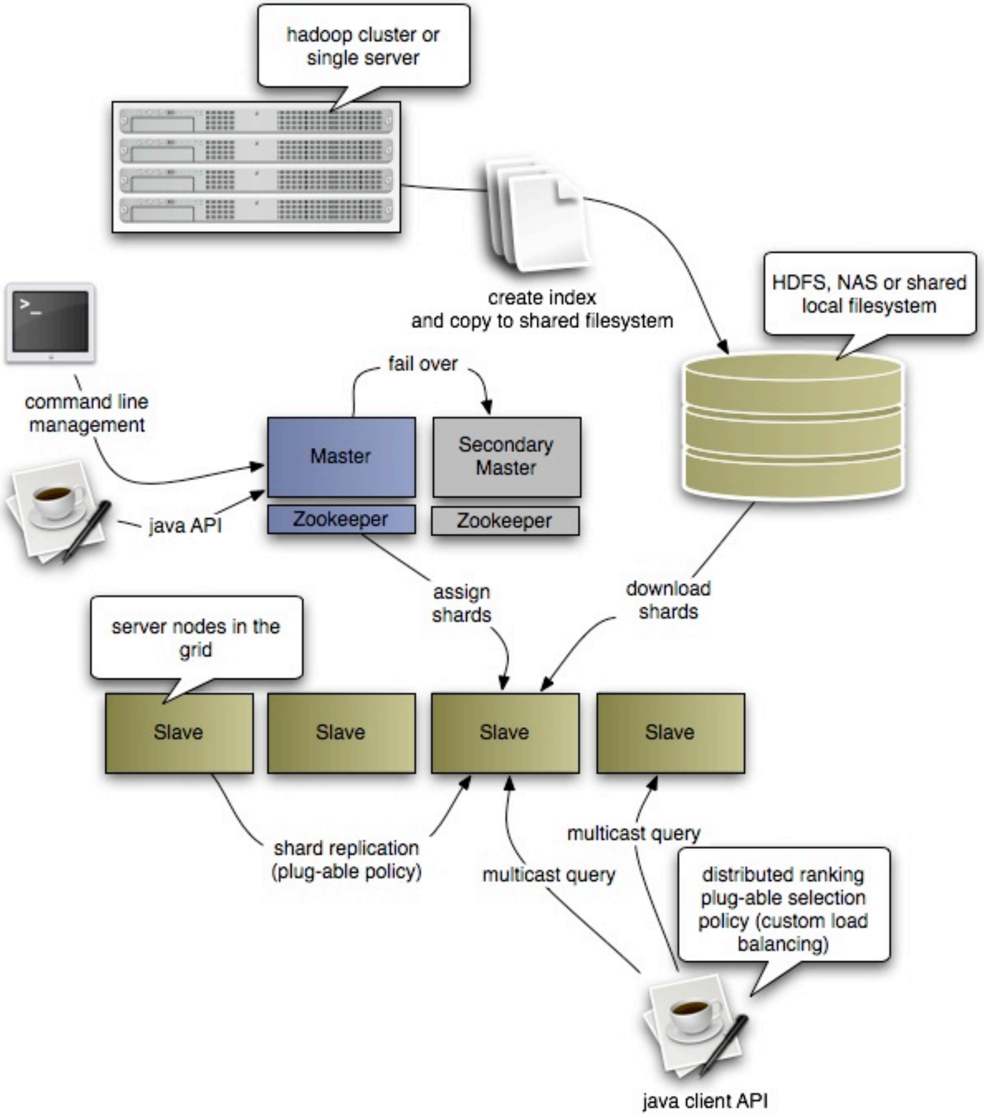
The rest is just details!

Term vs. Document Partitioning



Katta Architecture

(Distributed Lucene)



hadoop cluster or single server
create index and copy to shared filesystem
HDFS, NAS or shared local filesystem
command line management
fail over
Master
Secondary Master
Zookeeper
Zookeeper
java API
assign shards
download shards
server nodes in the grid
Slave
Slave
Slave
Slave
shard replication (plug-able policy)
multicast query
multicast query
distributed ranking plug-able selection policy (custom load balancing)
java client API

http://katta.sourceforge.net/

Questions?

Source: Wikipedia (Japanese rock garden)