Mining Massive Datasets

Lecture 8

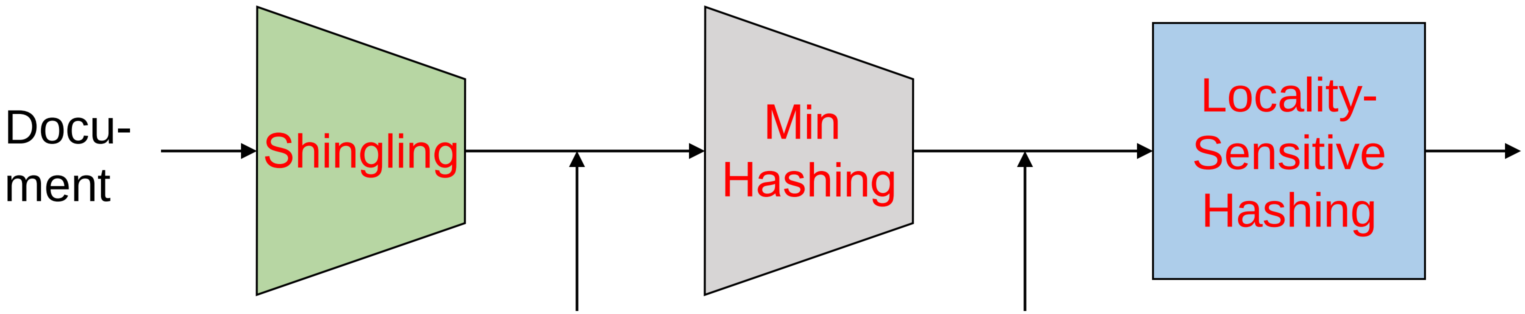
Artur Andrzejak
http://pvs.ifi.uni-heidelberg.de

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Note on Slides

A substantial part of these slides come (either verbatim or in a modified form) from the book Mining of Massive Datasets by Jure Leskovec, Anand Rajaraman, Jeff Ullman (Stanford University).
For more information, see the website accompanying the book: http://www.mmds.org.

Recall



The set of strings of length ***k*** that appear in the document

Signatures: short integer vectors that represent the sets, and reflect their similarity

Candidate pairs**:** those pairs of signatures that we need to test for similarity

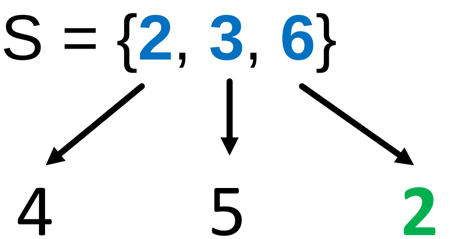
MinHashing

Finishing …

MinHash: Repetition

- Def.: Let **h** be a hash function that maps the members of S to distinct integers, then for any set S define **MinHash**$_h$(S) = **h**$_{min}$(S) to be the minimum value of **h**(x)

- **Example**:
  - Assume S = {**2**, **3**, **6**} and
  - **h**(**2**) = 4, **h**(**3**) = 5, **h**(**6**) = 2



- $h_{min}(S) = 2$

Min-Hashing: Other Interpretation

- We represent each set as a Boolean vector **C** (here: S = {**2**, **3**, **6**})
- Assume that a hash function **h** is given by a (random) permutation π of the rows of the Boolean vector
  - **h** fulfills: "… maps the members of S to distinct integers"

1	0
2	**1**
3	**1**
4	0
5	0
6	**1**

Row perm.

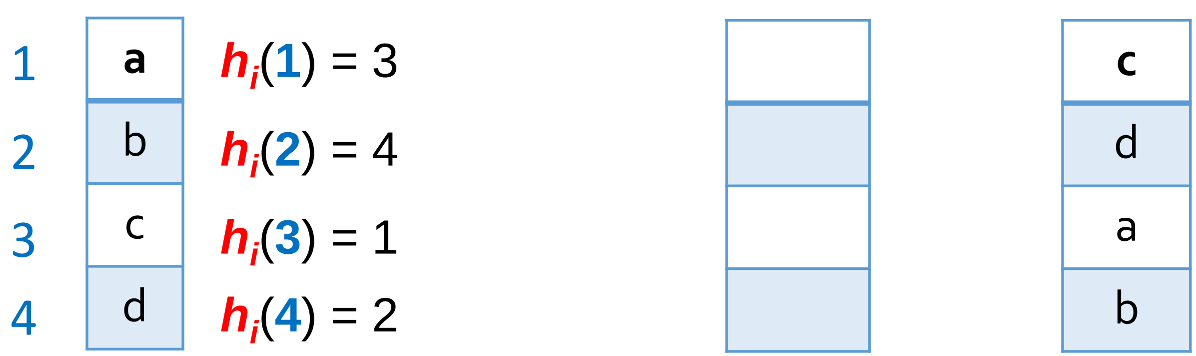
$\pi(\mathbf{1}) = 3$
$\pi(\mathbf{2}) = 4$
$\pi(\mathbf{3}) = 5$
$\pi(\mathbf{4}) = 6$
$\pi(\mathbf{5}) = 1$
$\pi(\mathbf{6}) = 2$

1	
<u>2</u>	**1**
3	
<u>4</u>	**1**
<u>5</u>	**1**
6	

- Then **MinHash$_h$**(S) is the <u>index</u> of the **first row** of the permuted column ***C*** with value **1**
- => Again, **h**$_{\pi,\min}$(S) = 2

Implementation /1

- Permuting rows even once is <u>very expensive</u>
- Approximate permutation by a <u>hash function</u> h_i
  - $h_i(x) = [((a \cdot x + b) \bmod p) \bmod N] + 1$
  - a, b: random integers;
    p: a prime (p > N);   N: #rows in the matrix
  - h_i is possibly not injective, but errors are rare => OK
- Pick about **K = 100** such hash functions h_i



Implementation /2

- Pick about **K = 100** such hash functions h_i

Intuition: for fixed C and h_i, find the smallest value $h_i(r)$ over all rows r with C(r) = 1

<u>One-pass implementation</u>:

- For each column ***C*** and hash-function h_i prepare a "slot" (variable) ***sig(C)[i]*** for the min-hash value
- Initialize all ***sig(C)[i]*** $= \infty$
- **Scan rows looking for 1s**
  - If row ***q*** has 1 in column ***C***, then for each h_i (i=1..100):
    - If $h_i(q) < sig(C)[i]$, then $sig(C)[i] \leftarrow h_i(q)$

Implementation /3

For fixed C and h_i:

Find the smallest value $h_i(r)$ over all rows r with C(r) = 1

Scan rows looking for 1s

- If row ***q*** has 1 in column ***C***, then for each h_i :
  - If $h_i(q)$ < ***sig(C)[i]***, then ***sig(C)[i]*** $\leftarrow h_i(q)$

Example: fixed ***C*** and h_i

S = {2, 3}

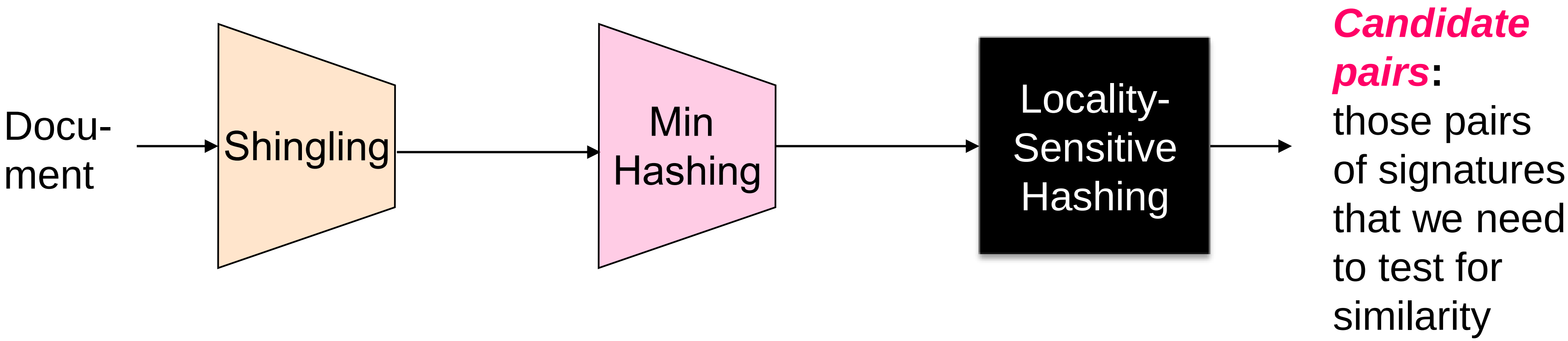
$2 \rightarrow 4$, $3 \rightarrow 1$

4 1

Scan

	C		sig(C)[i]	sig(C)[i]
1	**0**	$h_i(1) = 3$		∞
2	1	$h_i(2) = 4$		4
3	1	$h_i(3) = 1$		1
4	0	$h_i(4) = 2$		1

Locality Sensitive Hashing



Step 3: **Locality-Sensitive Hashing**:
Focus on pairs of signatures likely to be from similar documents

LSH: First Cut

2	1	4	1
1	2	1	2
2	1	2	1

- **Goal:** Find documents with Jaccard similarity at least ***s*** (for some similarity threshold, e.g., ***s***=0.8)

- **LSH – General idea:** Use a function ***f(x,y)*** that tells whether ***x*** and ***y*** is a ***candidate pair:*** a pair of elements whose similarity must be evaluated

- **For Min-Hash matrices:**
  - Hash columns of signature matrix ***M*** to many buckets
  - Each pair of documents that hashes into the same bucket is a candidate pair

Candidates from Min-Hash

2	1	4	1
1	2	1	2
2	1	2	1

- **Pick a similarity threshold *s* (0 < s < 1)**
- Columns ***x*** and ***y*** of ***M*** are a **candidate pair** if their signatures agree on at least fraction ***s*** of their rows:
  $M(i, x) = M(i, y)$ for at least frac. s values of i
  - We expect documents ***x*** and ***y*** to have the same (Jaccard) similarity as their signatures

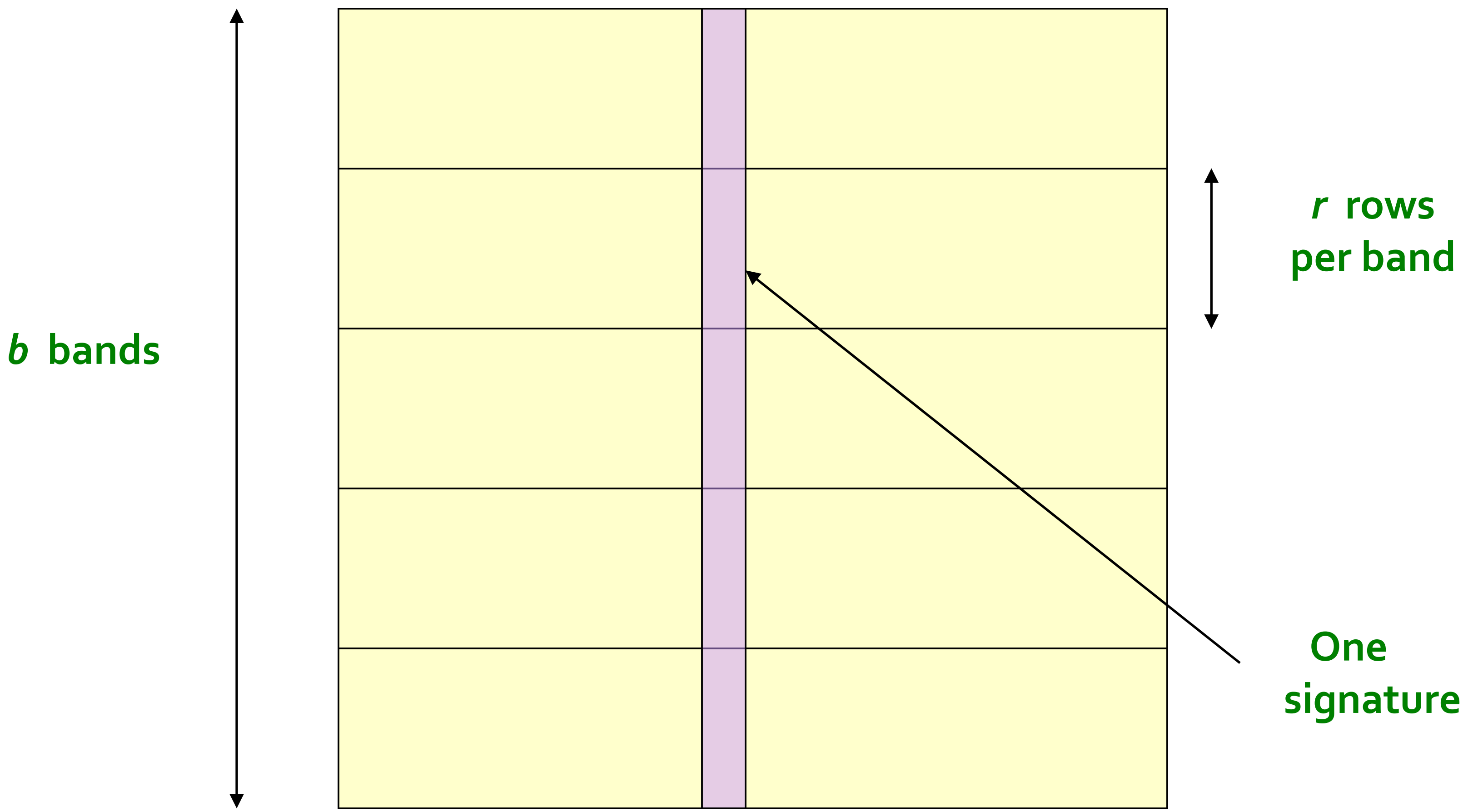
LSH for Min-Hash

2	1	4	1
1	2	1	2
2	1	2	1

- **Big idea: Hash columns of signature matrix *M* several times**
- Arrange that (only) **similar columns** are likely to **hash to the same bucket**, with high probability
- **Candidate pairs are those that hash to the same bucket**

Partition *M* into *b* Bands

2	1	4	1
1	2	1	2
2	1	2	1

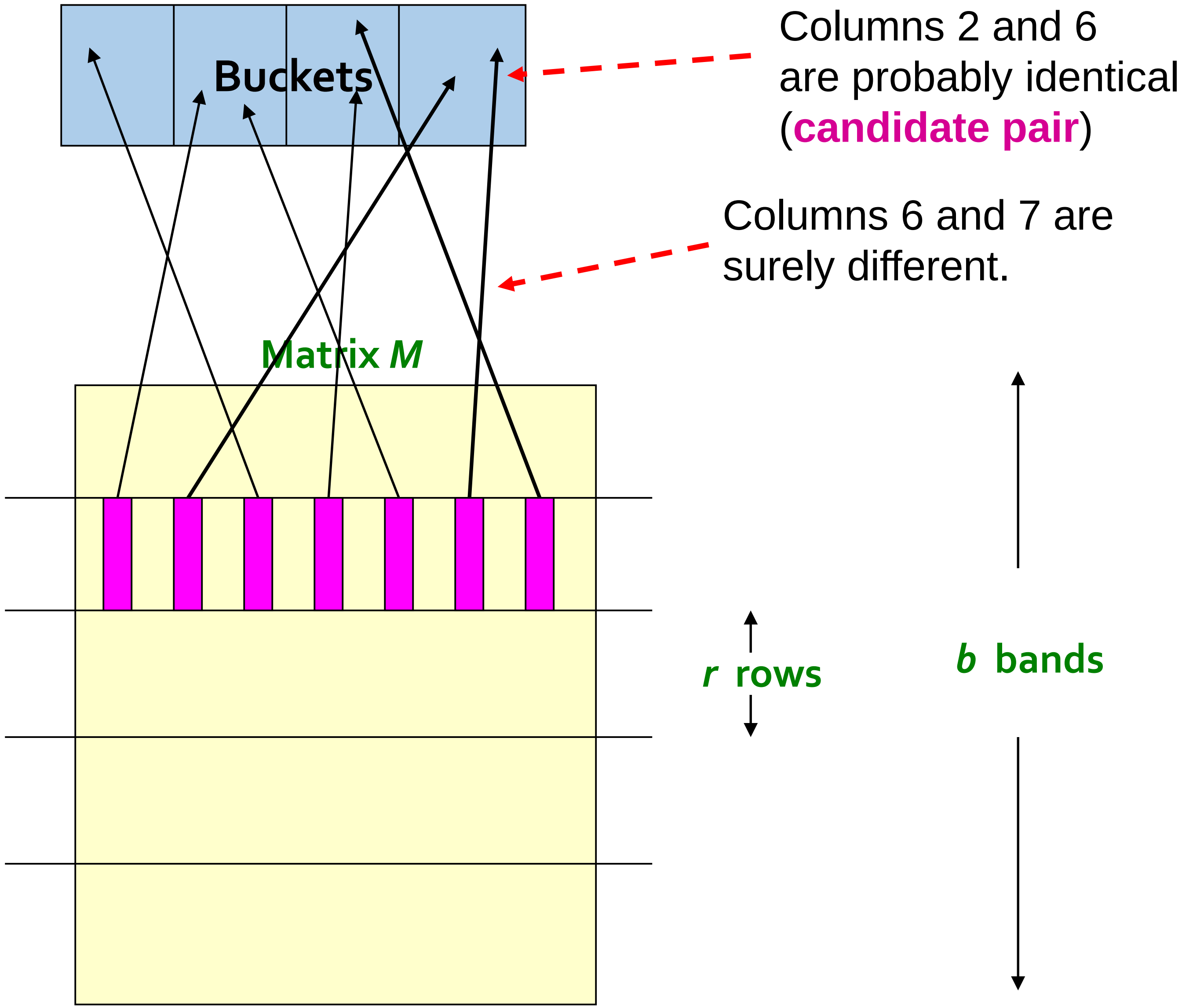


Signature matrix *M*

Partition M into Bands

- Divide matrix ***M*** into ***b*** bands of ***r*** rows
- For each band, hash its portion of each column to a hash table with ***k*** buckets
  - Make ***k*** as large as possible
- ***Candidate*** column pairs are those that hash to the same bucket for ≥ **1** band
- Tune ***b*** and ***r*** to catch most similar pairs, but few non-similar pairs

Hashing Bands



Simplifying Assumption

- There are **enough buckets** that columns are unlikely to hash to the same bucket unless they are **identical** in a particular band
- Hereafter, we assume that "**same bucket**" means "**identical in that band**"
- Assumption needed only to simplify analysis, not for correctness of algorithm

Example of Bands

2	1	4	1
1	2	1	2
2	1	2	1

Assume the following case:

- Suppose 100,000 columns of ***M*** (100k docs)
- Signatures of 100 integers (rows)
- Therefore, signatures take 40Mb
- Choose ***b*** = 20 bands of ***r*** = 5 integers/band

- **Goal:** Find pairs of documents that are at least ***s*** = *0.8* similar

C_1, C_2 are 80% Similar

2	1	4	1
1	2	1	2
2	1	2	1

- **Find pairs of** $\geq$ ***s***=0.8 similarity, set **b**=20, **r**=5
- **Assume:** sim(C_1, C_2) = 0.8
  - Since sim(C_1, C_2) $\geq$ **s**, we want C_1, C_2 to be a **candidate pair**: We want them to hash to at **least 1 common bucket** (at least one band is identical)
- **Probability C_1, C_2 identical in one particular band:** $(0.8)^5 = 0.328$
- Probability C_1, C_2 are ***not*** similar in all of the 20 bands: $(1-0.328)^{20} = 0.00035$
  - i.e., about 1/3000th of the 80%-similar column pairs are **false negatives** (we miss them)
  - **We would find 99.965% pairs of truly similar documents**

C_1, C_2 are 30% Similar

2	1	4	1
1	2	1	2
2	1	2	1

- **Find pairs of ≥ s=0.8 similarity, set b=20, r=5**
- **Assume:** $\text{sim}(C_1, C_2) = 0.3$
  - Since $\text{sim}(C_1, C_2) < \mathbf{s}$ we want C_1, C_2 to hash to **NO common buckets** (all bands should be different)
- **Probability C_1, C_2 identical in one particular band:** $(0.3)^5 = 0.00243$
- Probability C_1, C_2 identical in at least 1 of 20 bands: $1 - (1 - 0.00243)^{20} = 0.0474$
  - In other words, approximately 4.74% pairs of docs with similarity 0.3% end up becoming **candidate pairs**
    - They are **false positives** since we will have to examine them (they are candidate pairs) but then it will turn out their similarity is below threshold **s**

LSH Involves a Tradeoff

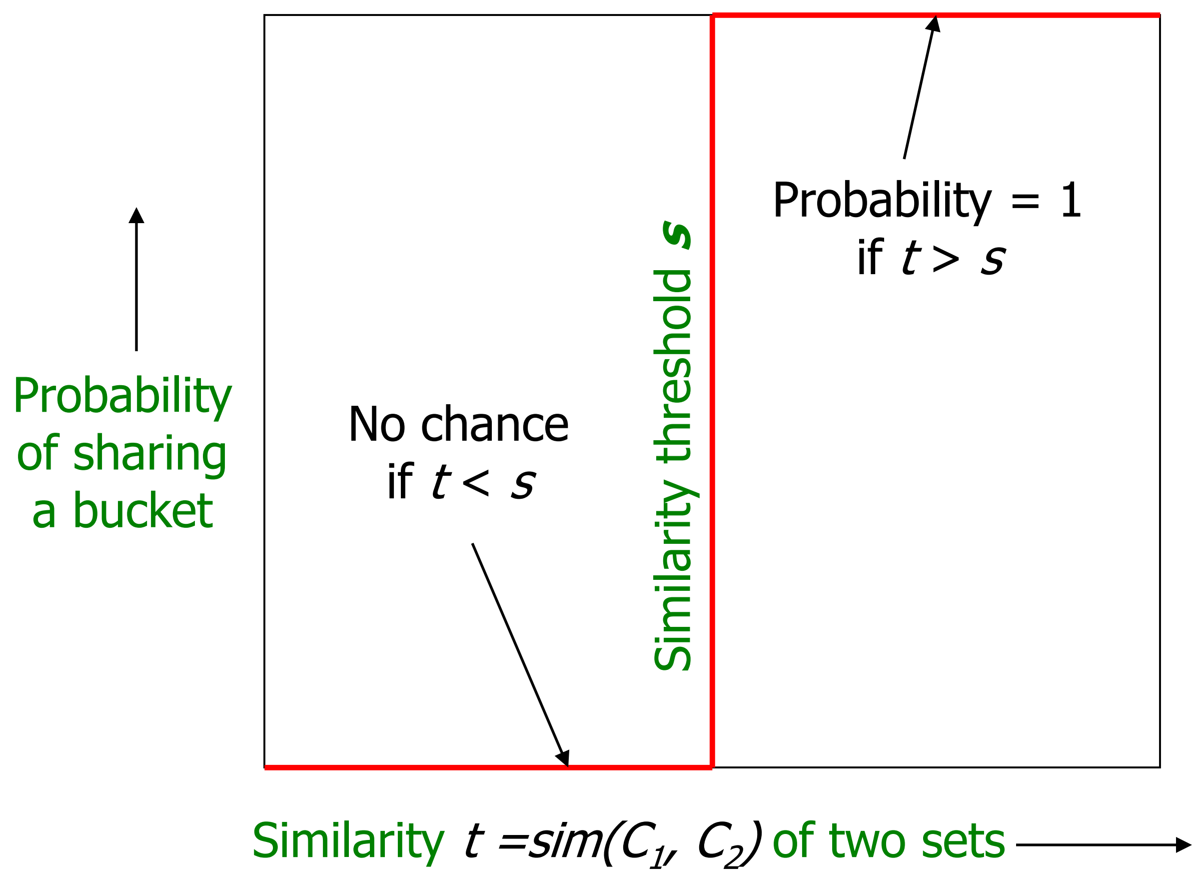
2	1	4	1
1	2	1	2
2	1	2	1

- **Pick:**
  - The number of Min-Hashes (rows of ***M***)
  - The number of bands ***b***, and
  - The number of rows ***r*** per band

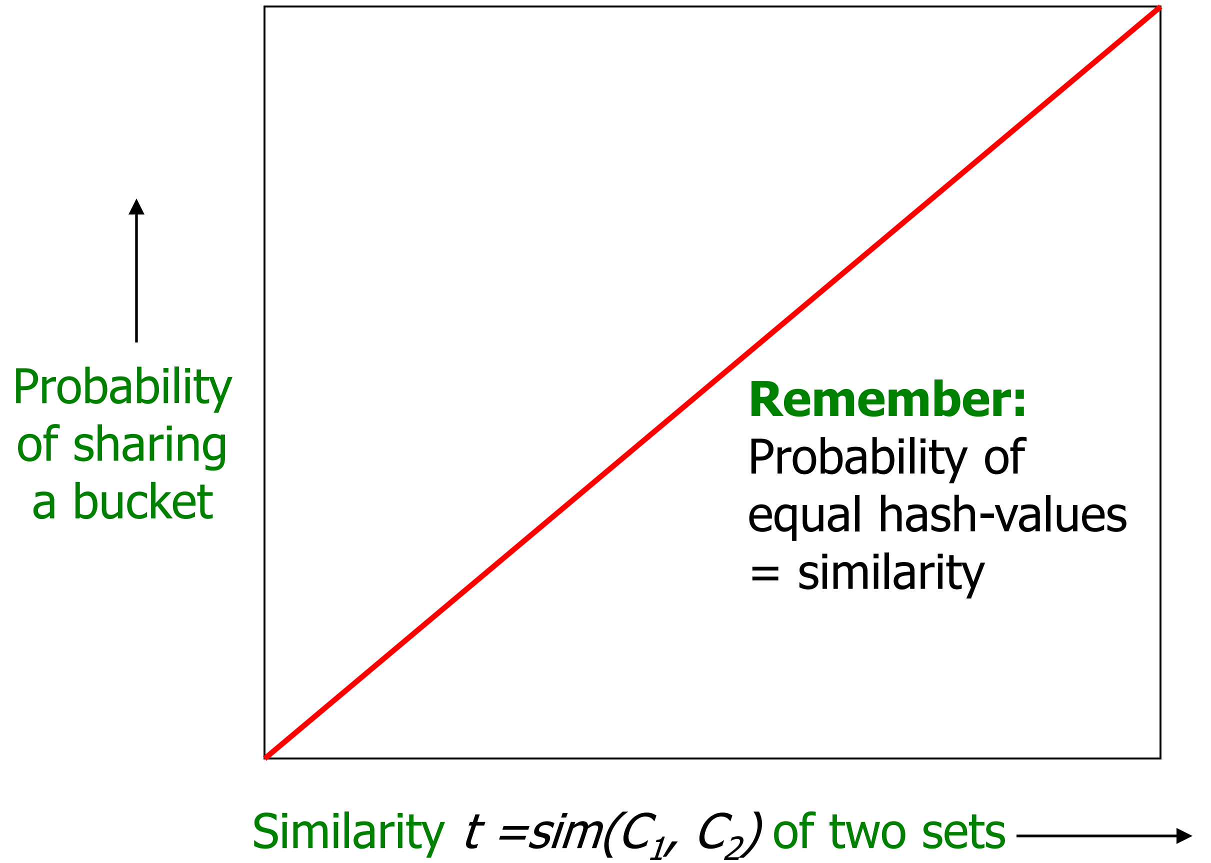
  to balance false positives/negatives

- **Example:** If we had only 15 bands of 5 rows, the number of false positives would go down, but the number of false negatives would go up

Analysis of LSH – What We Want



What 1 Band of 1 Row Gives You



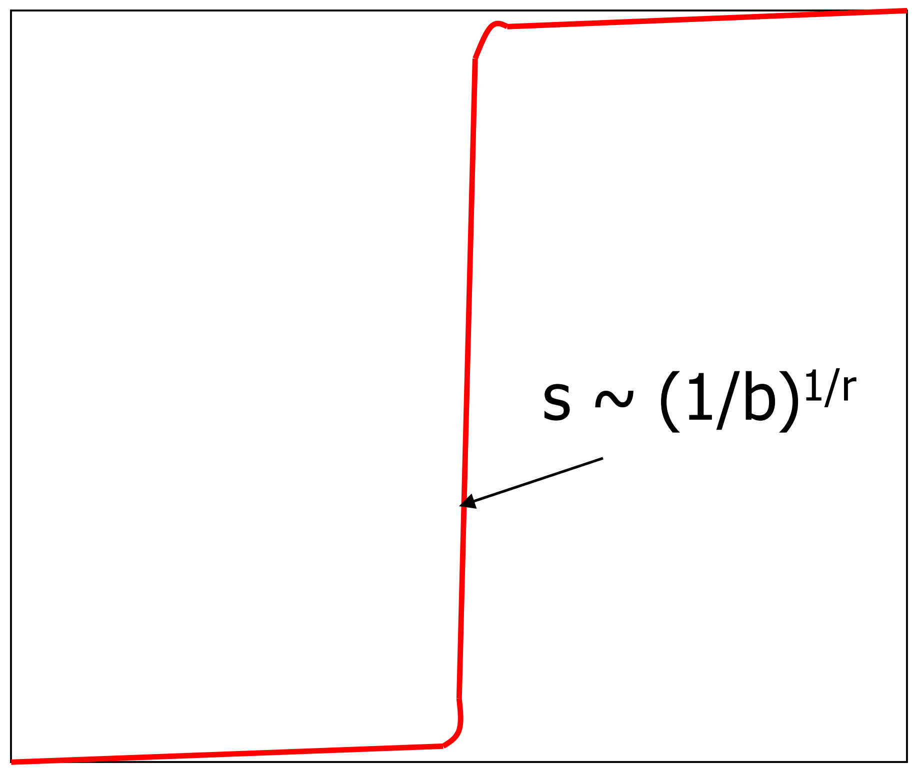
b bands, r rows/band

- Columns C_1 and C_2 have similarity t
- Pick any band (r rows)
  - Prob. that all rows in band equal = t^r
  - Prob. that some row in band unequal = $1 - t^r$
- Prob. that no band identical = $(1 - t^r)^b$
- Prob. that at least 1 band identical =

$$1 - (1 - t^r)^b$$

What b Bands of r Rows Gives You

Probability of sharing a bucket



Similarity $t=sim(C_1, C_2)$ of two sets

At least one band identical

No bands identical

$$\mathbf{1 - (1 - } \boldsymbol{t}^{\boldsymbol{r}}\mathbf{)}^{\boldsymbol{b}}$$

Some row of a band unequal

All rows of a band are equal

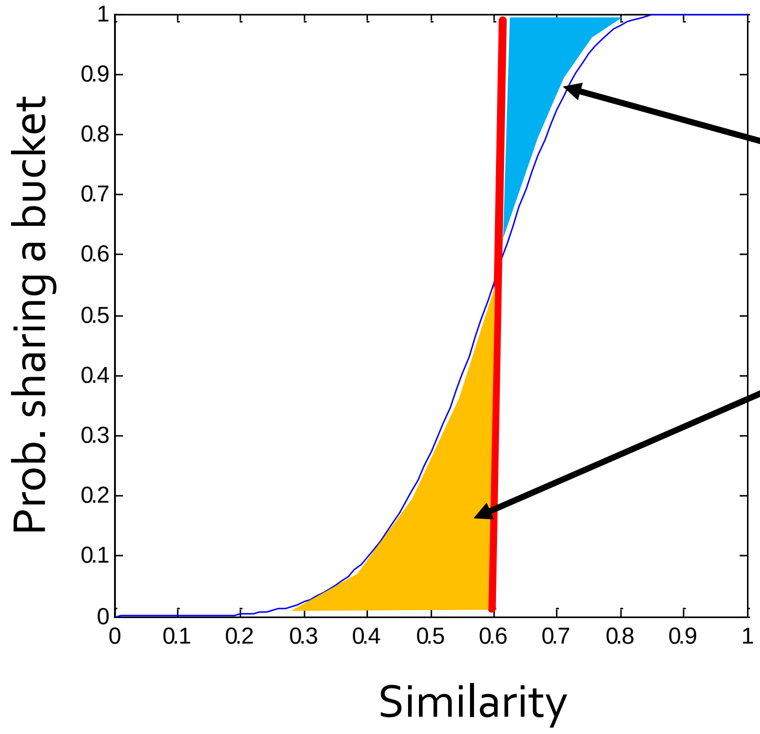
Example: $b = 20; r = 5$

- **Similarity threshold s**
- **Prob. that at least 1 band is identical:**

s	$1-(1-s^r)^b$
.2	.006
.3	.047
.4	.186
.5	.470
.6	.802
.7	.975
.8	.9996

Picking *r* and *b*: The S-curve

- **Picking *r* and *b* to get the best S-curve**
  - 50 hash-functions (r=5, b=10)



Blue area: False Negative rate

Orange area: False Positive rate

LSH Summary

- Tune ***M, b, r*** to get almost all pairs with similar signatures, but eliminate most pairs that do not have similar signatures
- Check in main memory that **candidate pairs** really do have **similar signatures**
- **Optional:** In another pass through data, check that the remaining candidate pairs really represent similar documents

Summary: 3 Steps

- **Shingling:** Convert documents to sets
  - We used hashing to assign each shingle an ID
- **Min-Hashing:** Convert large sets to short signatures, while preserving similarity
  - We used **similarity preserving hashing** to generate signatures with property $\mathbf{Pr}[h_\pi(C_1) = h_\pi(C_2)] = sim(C_1, C_2)$
  - We used hashing to get around generating random permutations
- **Locality-Sensitive Hashing:** Focus on pairs of signatures likely to be from similar documents
  - We used hashing to find **candidate pairs** of similarity $\geq$ **s**

Spark:
DataFrames and Datasets

Repetition

DataFrames in Spark

- **DataFrames** (DF) are tables with named and typed data columns
  - Similar to a dataframe in R, or Pandas (Python), or tables in DBMS/SQL
  - Impose a structure and schema on data
- Example

	Partition 1			Partition 2			Partition 3			Partition 4		
	Time (Str)	Site (Str)	Req (Int)	Time (Str)	Site (Str)	Req (Int)	Time (Str)	Site (Str)	Req (Int)	Time (Str)	Site (Str)	Req (Int)
Row 1	ts	m	1304	ts	d	3901	ts	m	1172	ts	m	2538
Row 2	ts	d	2237	ts	d	2491	ts	m	2137	ts	d	2837
Row 3	ts	m	1600	ts	d	2288	ts	d	3176	ts	d	3400

From: Databricks, 7 Steps for a Developer to Learn Apache Spark, p. 14, 2017

DataFrames from RDDs

```
// Read file with rows: <name, age>
filePath = „/home/immd-user/spark-2 …/examples/src/main/resources/people.txt“
parts = sc.textFile(filePath).map( lambda line: line.split(“,“) )

// Each row should become a tuple (name, age)
peopleRDD = parts.map( lambda p: (p[0], p[1].strip() )

from pyspark.sql import SparkSession
spark = SparkSession.builder.appName(„Exmpl").getOrCreate()
schema = StructType( [
                StructField (“name“,  StringType(), True),
                StructField ( „age“,     StringType(), True) ] )

dfPeople = spark.createDataFrame (peopleRDD, schema)
print ( dfPeople.take(5) )
```

SQL: More User-Friendly

Standard DF API:

```
# Select people older than 25
dfPeople.filter(dfPeople['age'] > 25).show()
    # |age|name|
    # | 30|Andy|
```

Same result with **SQL**:

```
# Register the DataFrame as a SQL temporary view
dfPeople.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT * FROM people where age > 25")
sqlDF.show()
```

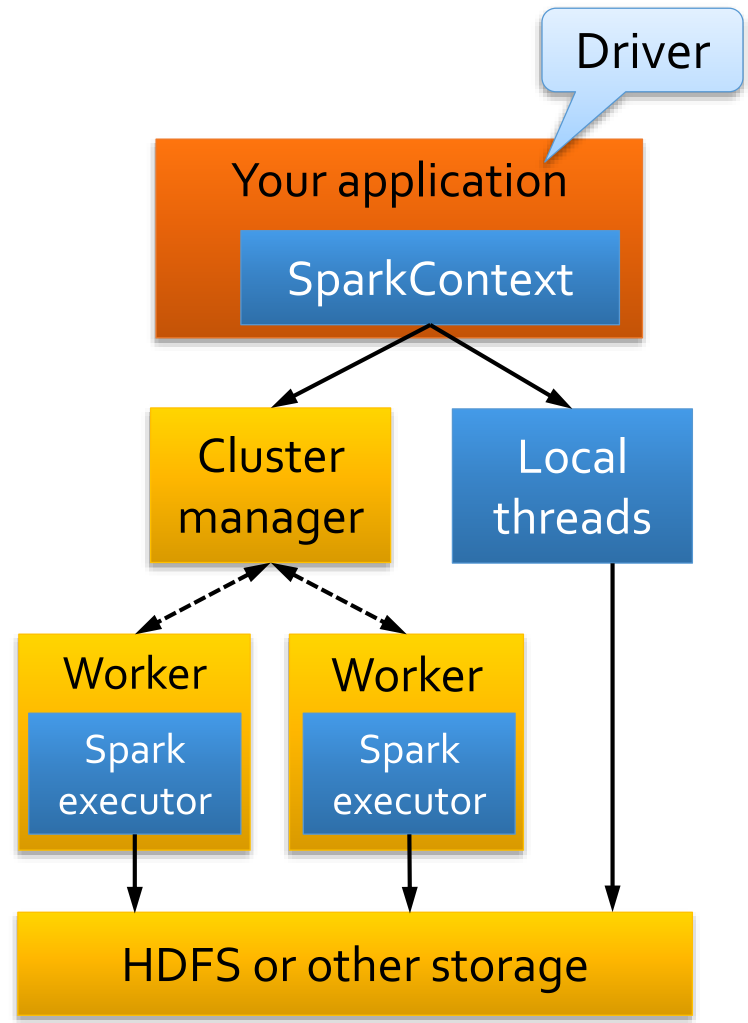
Video

- Coursera: Big Data Integration and Processing
  - https://www.coursera.org/learn/big-data-integration-processing/home/info
  - Week 5: Programming in Spark
  - Video: Hands-on: Data Processing in Spark => Exploring SparkSQL and Spark DataFrames
    - Link:
    - https://www.coursera.org/learn/big-data-integration-processing/lecture/aHc8E/exploring-sparksql-and-spark-dataframes

Spark: Execution Details

Software Components

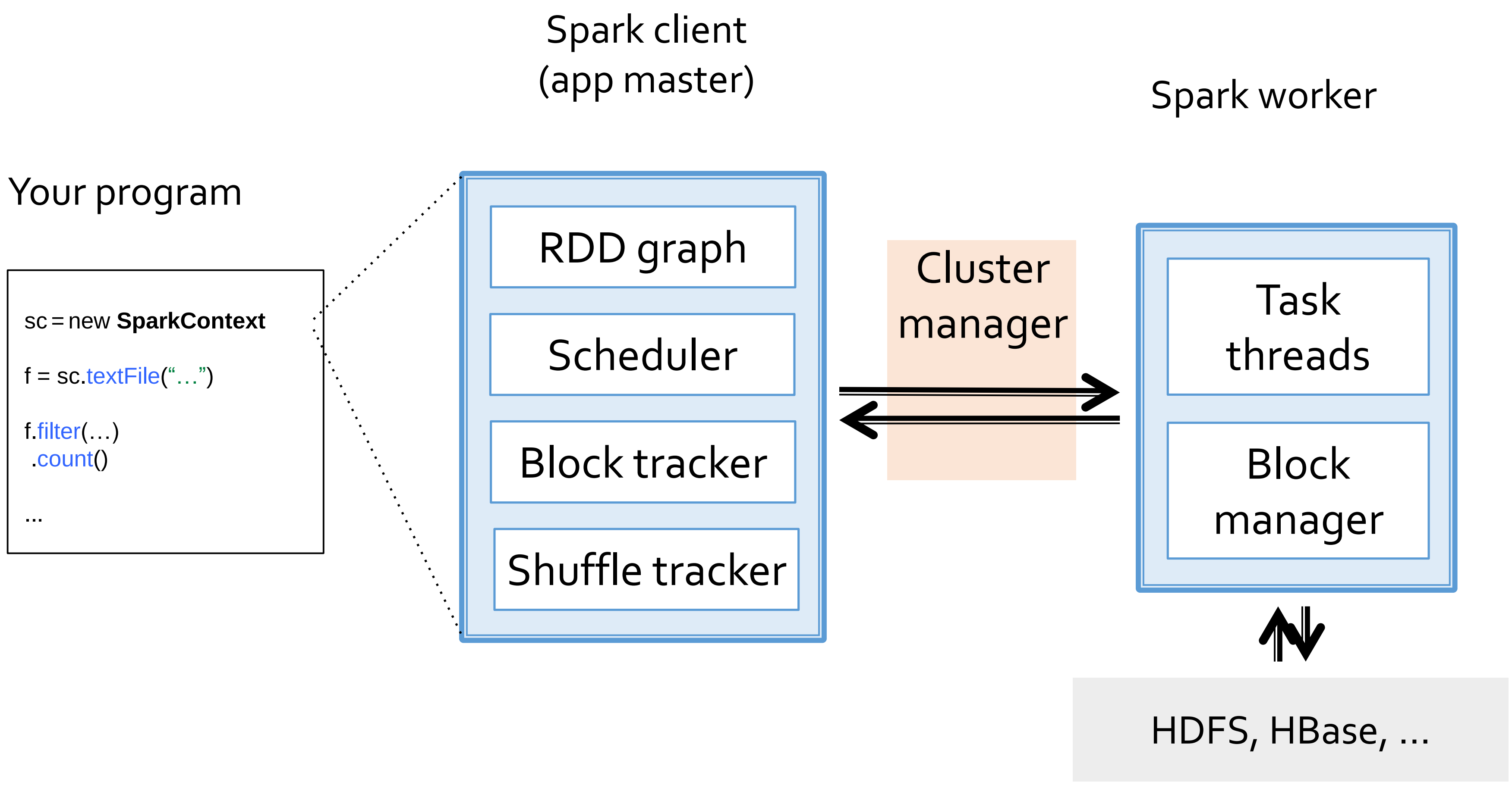
- Spark runs as a library in your program
- Runs tasks locally / on cluster*
  - Standalone, YARN, Mesos
  - See Cluster Mode Overview*
- Accesses storage systems via Hadoop API
  - Can use HBase, HDFS, S3, …



*=http://spark.apache.org/docs/latest/cluster-overview.html

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Components



For more info see video "Introduction to AmpLab Spark Internals" (https://www.youtube.com/watch?v=49Hr5xZyTEA) and read slides http://files.meetup.com/3138542/dev-meetup-dec-2012.pptx

From: *Parallel Programming with Spark*, **Matei Zaharia**, AmpCamp 2013

Example Job

```
sc = SparkContext (appName="PythonExample")

file = sc.textFile(“hdfs://...”)

errors = file.filter(lambda line:“ERROR” in line))

errors.cache()

errors.count()
```

RDDs

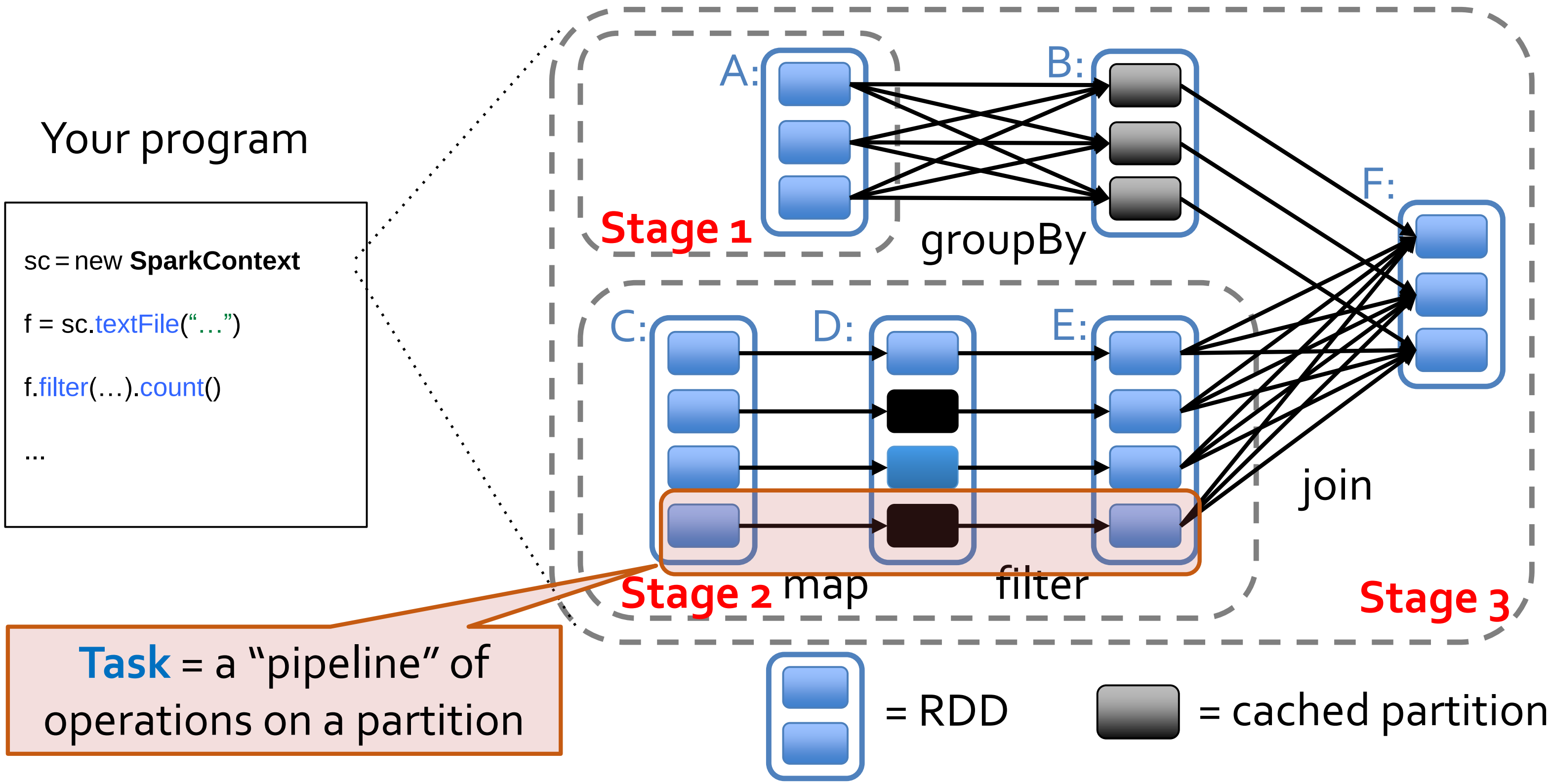
Action

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Operator DAG

The **operator DAG** (Directed Acyclic Graph) captures RDD dependencies

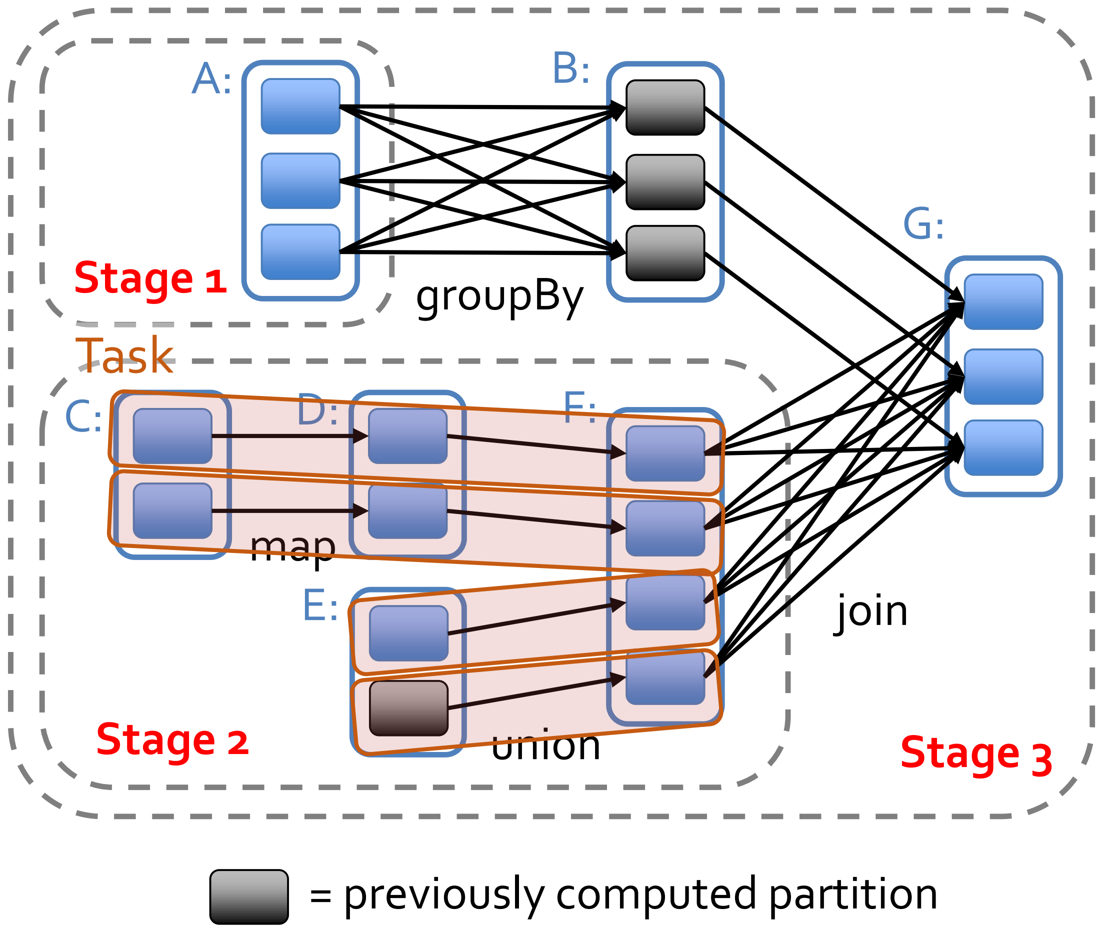
Stage: explained later



From: *Parallel Programming with Spark*, **Matei Zaharia**, AmpCamp 2013

Stages

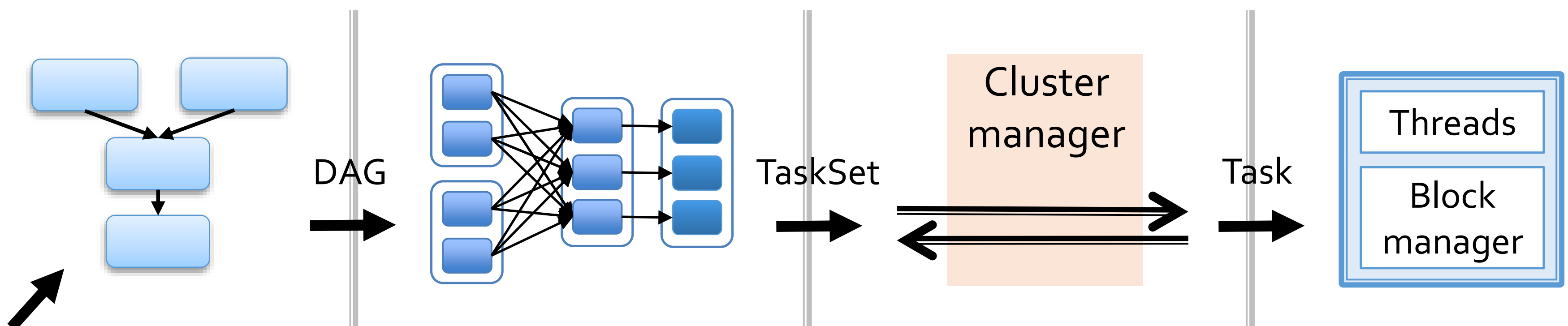
- A set of independent tasks, as large as possible
- Stage boundaries are only at <u>input RDDs</u> or <u>"shuffle" operations</u> (like groupBy*, join, …)



From: *Introduction to Spark Internals*, **Matei Zaharia**, 2012

Scheduling Process

RDD Objects | **DAGScheduler** | **TaskScheduler** | Worker



`rdd1.join(rdd2)`
`.groupBy(…)`
`.filter(…)`

build **operator DAG** (DAG = Directed Acyclic Graph)

split graph into **stages** of tasks (stages ="maximal" sets of independent tasks)

launch tasks via cluster manager

retry failed or straggling tasks

stage failed

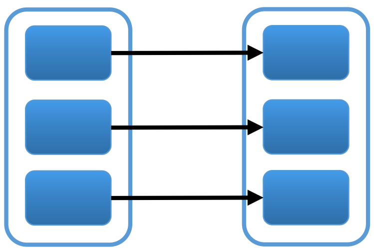
execute tasks

store and serve **blocks**

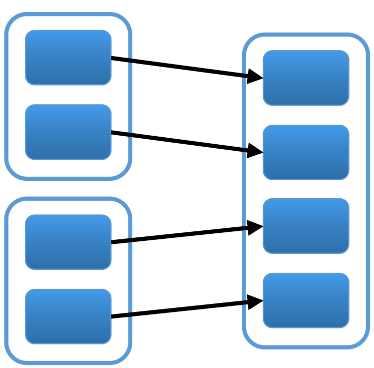
From: *Introduction to Spark Internals*, **Matei Zaharia**, Meetup Dec 2012

Dependency Types in DAG

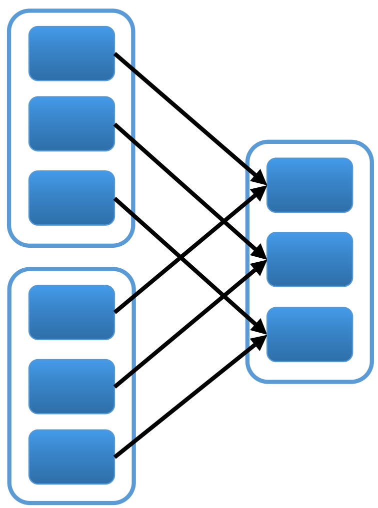
"Narrow" dependencies:



map, filter

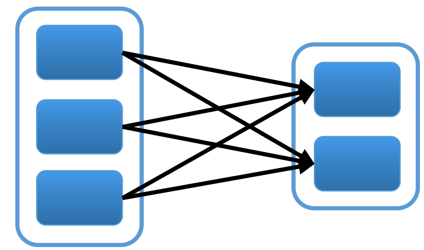


union

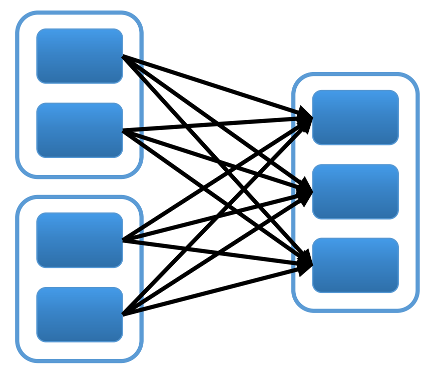


join with inputs co-partitioned

"Wide" (shuffle) deps:



groupByKey

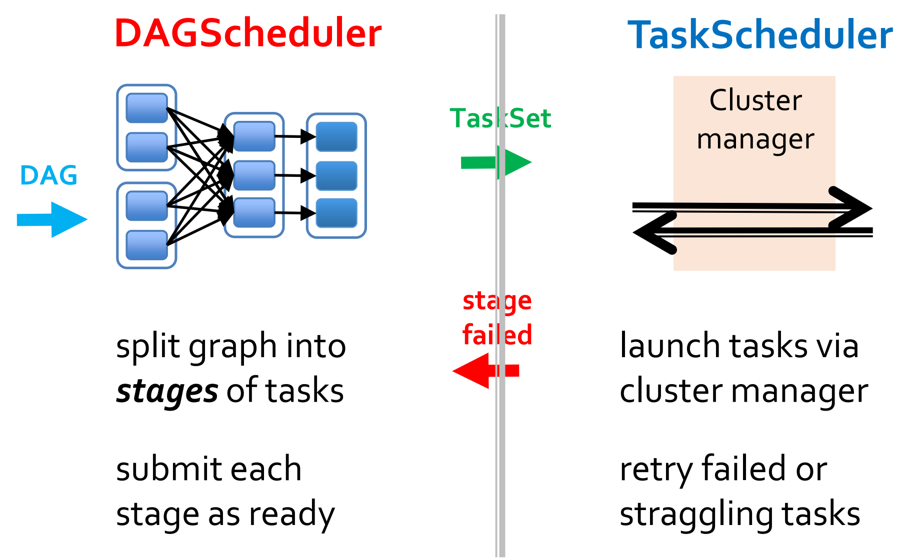


join with inputs not co-partitioned

From: *Introduction to Spark Internals,* **Matei Zaharia**, Meetup Dec 2012

DAG Scheduler vs. Task Scheduler

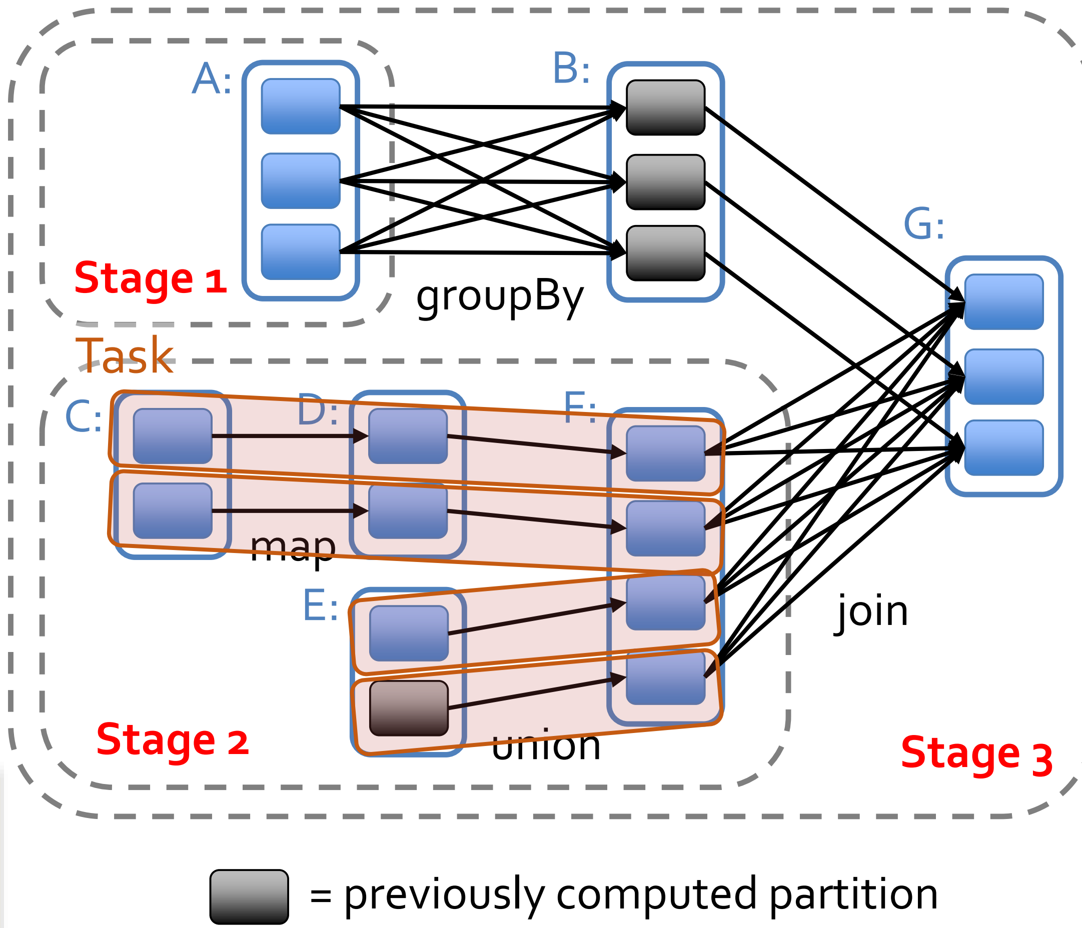
- DAG Scheduler – "higher level"
  - Builds stages of task objects (by code + preferred location)
  - Submits them to TaskScheduler as ready
  - Resubmits failed stages if outputs are lost
- **TaskScheduler**
  - "Lower level" – similar to Hadoop master
  - Given a set of tasks, runs it and reports results
  - Exploits data locality
  - Local / cluster implementation



From: *Introduction to Spark Internals*, **Matei Zaharia**, Meetup Dec 2012

Scheduler Optimizations

- Pipelines narrow ops. within a stage
- Picks join algorithms based on partitioning (minimize shuffles)
- Reuses previously cached data

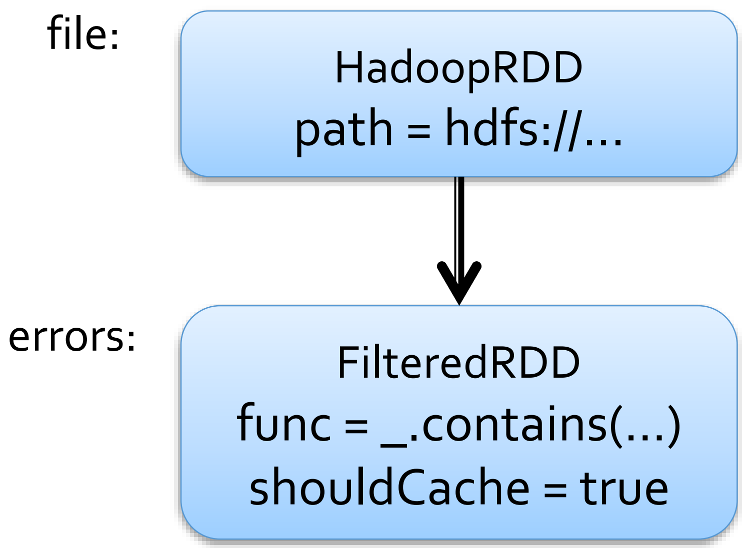


In MapReduce, each M-R phase is "individual"
=> Less optimization!

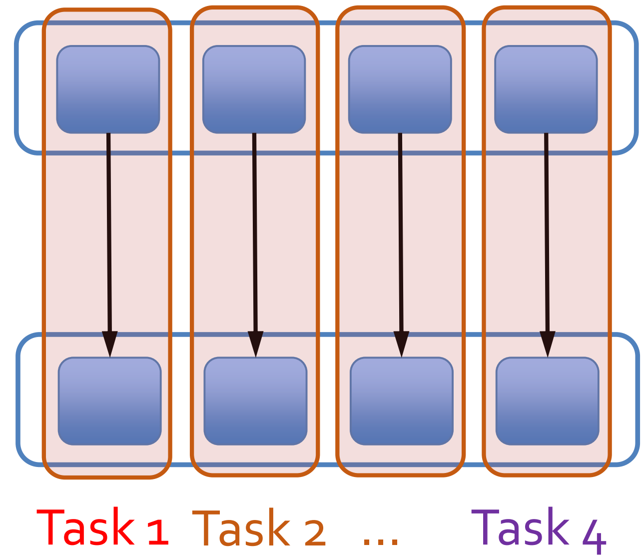
From: *Introduction to Spark Internals*, **Matei Zaharia**, 2012

RDD Graph

Dataset-level view:



Partition-level view:



- **Partition**: a subset of RDD, usually corresponding to a block of HDFS (or other file system)
- **Task**: a "pipeline" of operations on a single partition

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

RDD Interface

- Set of *partitions* ("splits")
- List of *dependencies* on parent RDDs
- Function to *compute* a partition given parents
- Optional *preferred locations*
- Optional *partitioning info* (Partitioner)

Captures all current Spark operations!

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Example: HadoopRDD

- partitions = one per HDFS block
- dependencies = none
- compute*(partition)* = read corresponding block
- preferredLocations*(part)* = HDFS block location
- partitioner = none

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Example: JoinedRDD

- partitions = one per reduce task
- dependencies = “shuffle” on each parent
- compute*(partition)* = read and join shuffled data

- preferredLocations*(part)* = none
- partitioner = HashPartitioner(numTasks)

Spark will now know this data is hashed!

From: *Parallel Programming with Spark,* **Matei Zaharia**, AmpCamp 2013

Thank you.

Questions?