CMSC424: Database Design

Module: NoSQL; Big Data Systems

Overview

Instructor: Amol Deshpande
amol@umd.edu

NoSQL and Big Data Systems: Motivation

- Book Chapters
  - 10.1, 10.2 **(7TH EDITION)**
- Key topics:
  - Big data motivating scenarios
  - Why systems so far (relational databases, data warehouses, parallel databases) don't work

RDBMS: Application Scenarios

- Online Transaction Processing (OLTP)
  - E-commerce, Airline Reservations, Class registrations, etc.
  - Simple queries (get all orders for a customer)
  - Many updates (inserts, updates, deletes)
  - Need ACID properties (consistency, etc.)
- Online Analytical Processing (OLAP)
  - Decision-support, data mining, ML (today), etc.
  - Huge volumes of data, but not updated
  - Complex, but read-only queries (many joins, group-by's)

RDBMS Evolution

- Original database systems aimed to support both use cases
- Slowly, specialized systems were built, starting late 80's-early 90's, especially for decision support (Data Warehouses)

- Today, different RDBMSs systems for different use cases, e.g.,:
  - VoltDB for **OLTP** – fully in-memory, very fast transactions, but no complex queries
  - **(OLAP)** Teradata, Aster Data, Snowflake, AWS Redshift – handle PBs of data, but batch updates only – many indexes and summary structures (cubes) for queries – typically "parallel" (i.e., use many machines)
- Fundamental and wide differences in the technology

- But both still support SQL as the primary interface (with visualizations, exploration, and other tools on top)

NoSQL + Big Data Systems: Motivation

- Very large volumes of data being collected
  - Driven by growth of web, social media, and more recently internet-of-things
  - Web logs were an early source of data
    - Analytics on web logs has great value for advertisements, web site structuring, what posts to show to a user, etc

- Big Data: differentiated from data handled by earlier generation databases
  - **Volume**: much larger amounts of data stored
  - **Velocity**: much higher rates of insertions
  - **Variety**: many types of data, beyond relational data

The FOUR V's of Big Data

From traffic patterns and music downloads to web history and medical records, data is recorded, stored, and analyzed to enable the technology and services that the world relies on every day. But what exactly is big data, and how can these massive amounts of data be used?

As a leader in the sector, IBM data scientists break big data into four dimensions: **Volume, Velocity, Variety and Veracity**

Depending on the industry and organization, big data encompasses information from multiple internal and external sources such as transactions, social media, enterprise content, sensors and mobile devices. Companies can leverage data to adapt their products and services to better meet customer needs, optimize operations and infrastructure, and find new sources of revenue.

By 2015
4.4 MILLION IT JOBS
will be created globally to support big data, with 1.9 million in the United States

Volume
SCALE OF DATA

40 ZETTABYTES
[43 TRILLION GIGABYTES]
of data will be created by 2020, an increase of 300 times from 2005

6 BILLION PEOPLE
have cell phones

WORLD POPULATION: 7 BILLION

It's estimated that
2.5 QUINTILLION BYTES
[2.3 TRILLION GIGABYTES]
of data are created each day

Most companies in the U.S. have at least
100 TERABYTES
[100,000 GIGABYTES]
of data stored

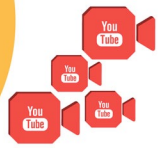
Variety
DIFFERENT FORMS OF DATA

As of 2011, the global size of data in healthcare was estimated to be
150 EXABYTES
[161 BILLION GIGABYTES]

30 BILLION PIECES OF CONTENT
are shared on Facebook every month

By 2014, it's anticipated there will be
420 MILLION WEARABLE, WIRELESS HEALTH MONITORS

4 BILLION+ HOURS OF VIDEO
are watched on YouTube each month



400 MILLION TWEETS
are sent per day by about 200 million monthly active users

Velocity
ANALYSIS OF STREAMING DATA

The New York Stock Exchange captures
1 TB OF TRADE INFORMATION
during each trading session

Modern cars have close to
100 SENSORS
that monitor items such as fuel level and tire pressure

By 2016, it is projected there will be
18.9 BILLION NETWORK CONNECTIONS
– almost 2.5 connections per person on earth

Veracity
UNCERTAINTY OF DATA

1 IN 3 BUSINESS LEADERS
don't trust the information they use to make decisions

in one survey were unsure of how much of their data was inaccurate

Poor data quality costs the US economy around
$3.1 TRILLION A YEAR

Sources: McKinsey Global Institute, Twitter, Cisco, Gartner, EMC, SAS, IBM, MEPTEC, QAS

IBM

Some motivating scenarios

- Deciding what to show a user in a social network, or news aggregator
- Advertising on the Web or Mobile
- Analyzing user behavior on web sites to optimize or increase engagement
- Analyzing large numbers of images and building search indexes on them
- Text analytics for topic modelling, summarization, …
- Internet of things…
- And many many others…

Two Primary Use Cases

- OLTP-like
  - Simple queries, but lots of updates
  - Need to support distributed users
  - Need to support non-relational data (e.g., graphs, JSONs)
  - Need to scale fast (10 users to 10s of Millions of Users)
  - Need to work well in 3-tier Web Apps
  - Need to support fast schema changes

NoSQL Storage Systems: HDFS, Cassandra, MongoDB, Neo4j, AWS DynamoDB, and many many others

- OLAP-like
  - Complex analysis on large volumes of data
  - Often no “real-time” component, and no updates
  - Mostly non-relational data (images, webpages, text, etc)
  - Tasks often procedural in nature (analyse webpages for searching, data cleaning, ML)

Big data frameworks: Hadoop MapReduce, Flink, Spark, and many others

Why (Parallel) Databases Don't Work

- The data is often not relational in nature
  - E.g., images, text, graphs
- The analysis/queries are not relational in nature
  - E.g., Image Analysis, Text Analytics, Natural Language Processing, Web Analytics, Social Network Analysis, Machine Learning, etc.
  - Databases don't really have constructs to support this
    - User-defined functions can help to some extent
  - Need to interleave relational-like operations with non-relational (e.g., data cleaning, etc.)
  - Domain users are more used to procedural languages
- The operations are often one-time
  - Only need to analyse images once in a while to create a "deep learning" model
  - Databases are really better suited for repeated analysis of the data
- Much of the analysis not time-sensitive
- Parallel databases too expensive given the data volumes
  - Were designed for large enterprises, with typically big budgets

Examples of Systems

- Too much variety in the systems out there today
  - different types of data models supported
    - Files/Objects (HDFS, AWS S3), Document (MongoDB), Graph (Neo4j), Wide-table (Cassandra, DynamoDB), Multi-Model (Azure CosmosDB)
  - different types of query languages or frameworks or workloads
    - SQL (Snowflake, Redshift, …), MongoQL, Cassandra QL, DataFrames (Spark), MapReduce (Hadoop), TensorFlow for ML, …
  - different environmental assumptions
    - Distributed vs parallel, disks or in-memory only, single-machine or not, streaming or static, etc.
  - different performance focus and/or guarantees
    - e.g., consistency guarantees in a distributed setting differ quite a bit
- Many of these systems work with each
  - e.g., Spark can read data from most of the storage systems
  - Interoperability increasing a requirement

Questions

- Optimizing a news aggregator platform -- what's most crucial for analyzing user behavior to increase engagement?
  1. Storing user data in a single, centralized database
  2. Performing complex OLAP queries on relational data
  3. Utilizing text analytics for topic modeling and summarization
  4. Restricting data analysis to real-time updates only

- What is an essential database feature for supporting a rapidly scaling social networking site, growing from 10 to tens of millions of users?
  1. Strict ACID transaction compliance
  2. Ability to perform complex joins on relational data
  3. Support for fast schema changes and distributed users
  4. Sole reliance on OLTP for all database operations

Questions

- For advertising on mobile platforms, why is analyzing large volumes of non-relational data (e.g., user clickstreams, app usage patterns) using OLAP-like systems preferred?
  1. Real-time data updates are crucial
  2. Simple queries with few updates are sufficient
  3. Complex analysis on large datasets offers deeper insights
  4. Only relational data models are supported for mobile advertising

- In the context of the Internet of Things (IoT), which database feature is most critical for handling data from distributed devices?
  1. Support for relational data models
  2. Capability to handle non-relational data such as JSONs and graphs
  3. Focus on OLAP-like systems for complex analysis
  4. No need for supporting distributed users

Questions

- Why might parallel databases be considered ineffective for analyzing social network data?
  1. Social network analysis relies solely on relational data models
  2. The analysis often requires interleaving relational-like operations with non-relational tasks, such as data cleaning
  3. User-defined functions are not helpful in social network analysis
  4. Parallel databases are specifically designed for social network analysis

- Considering the need for interoperability among different systems, which of the following is an increasingly important requirement?
  1. Reducing the variety of data models supported
  2. Limiting systems to either SQL or NoSQL query languages
  3. Ensuring that systems can work with each other, like Spark reading data from various storage systems
  4. Focusing solely on parallel databases for all use cases

Questions

- Why are databases that support fast schema changes and distributed users critical for 3-tier web applications?
  1. They ensure data is stored in a centralized location for easy access
  2. They provide consistency guarantees in a distributed setting
  3. They facilitate the rapid development and scaling of web applications by adapting to changing data models and user bases
  4. They limit the application to relational data models for simplicity

- Why are domain users, especially those working with big data and non-relational databases, more inclined towards systems that allow procedural languages and user-defined functions?
  1. Procedural languages are less powerful and versatile than SQL
  2. These features offer a way to perform customized, complex analyses that go beyond traditional relational operations, including data cleaning and machine learning
  3. Relational databases provide all the necessary constructs for big data analysis
  4. Parallel databases are too cost-effective and efficient to consider alternatives

What We Will Cover

- Apache Spark
  - Current leader in big data (OLAP-style) frameworks
  - Supports many query/analysis models, including a light version of SQL
- MongoDB
  - Perhaps the most popular NoSQL system, uses a ”document” (JSON) data model
  - Focus primarily on OLTP
  - Doesn’t really support joins (some limited ability today) – have to do that in the app
- How to “Parallelize” Operations
  - Useful to understand how Spark and other systems actually work
  - Often times you have to build these in the application layer
  - The original MapReduce framework
    - Led to development of much work on large-scale data analysis (OLAP-style)
    - Basically a way to execute a group-by at scale on non-relational data
- Hadoop Distributed File System (briefly)
  - A key infrastructure piece, with no real alternative
  - Basic file system interface, with replication and redundancy built in for failures
- Quick overview of other NoSQL data models

Plan

- Operations on Datasets
- Parallel Architectures
- Parallelizing Data-centric Operations
- Apache Spark
- Map-Reduce
- MongoDB
- Other NoSQL Systems

CMSC424: Database Design

Module: NoSQL; Big Data Systems

Operations on Datasets

Instructor: Amol Deshpande
amol@umd.edu

Operations on Datasets

- So far we have focused on “relations”/”tables”
  - Basically “multisets” of “tuples”
- Let’s generalize that concept and discuss a few more operations
  - Each operation takes in one or more datasets as input
  - … and outputs one dataset
- But what is a “dataset”?

Inputs and Outputs

- Sets or "multi-sets" or "collections" of ..
  - Objects → key-value pairs → tuples

o1
o2
o3
o4

Set of objects (e.g., images, JSON documents)

o1
o1
o1
o4

Multi-set of objects

k1	o1
k1	o2
k2	o3
k3	o4

Set of key-value pairs (e.g., pairs of (userid, profile))

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

Set of tuples aka Relation/Table

Operations on Datasets

- So far we have focused on “relations”/”tables”
  - Basically “multisets” of “tuples”
- Let’s generalize that concept and discuss a few more operations
  - Each operation takes in one or more datasets as input
  - … and outputs one dataset
- An operation that works on a collection of “objects” also works on a collection of ”key-value pairs” or “tuples”

Select/Find/Match/Filter

- Input: Object Collection, Output: Object Collection
  - So also works on Relations
- Select only those objects that match the condition
  - Need a predicate or a “function” that returns a boolean
- SQL: “where”

Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10

Select/Find/Match/Filter

- Input: Object Collection, Output: Object Collection
  - So also works on Relations
- Select only those objects that match the condition
  - Need a predicate or a “function” that returns a Boolean
- More generally:
  - D.filter(f): where “f” is a Boolean function

Input

o1
o2
o3
o4

f(o1) = T
f(o2) = F
f(o3) = F
f(o4) = T

o1
o4

Project/Map

- Input: Object Collection, Output: Object Collection
- For each object, output a transformed object
- Need a way to specify the transformation
- SQL can handle this through UDFs and “select” clause
  - select f(r.a, r.b) from R;

Input t

o1
o2
o3
o4

t.map(f)

f(o1)
f(o2)
f(o3)
f(o4)

Map Example

Input: a collection of log records

199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245
unicomp6.unicomp.net - - [01/Jul/1995:00:00:06 -0400] "GET /shuttle/countdown/ HTTP/1.0" 200 3985
…
…

map: use regex to extract IP address

Output: IP addresses

199.72.81.55
unicomp6.unicomp.net
….
…

Project/Map

- Input: Object Collection, Output: Object Collection
- For each object, output a transformed object
- Need a way to specify the transformation
- SQL Project is a simplification of this

Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\pi_{A,D}(r)$

A	D
α	7
α	7
β	3
β	10

A	D
α	7
β	3
β	10

mapValues

- Input: Collection of (key, value) pairs; Output: Collection of (key, value) pairs
- Only apply the “map” operation to the value

Input t

k1	o1
k1	o2
k2	o3
k3	o4

t.mapValues(f)

k1	f(o1)
k1	f(o2)
k2	f(o3)
k3	f(o4)

flatMap

- Has other names (e.g., unwind in MongoDB)
- Input: Object collection, Output: Object collection
- For each row, apply a function that may generate >= 0 objects
- SQL: No easy way to do this
  - LATERAL JOIN comes close (as does CROSS APPLY in SQL Server)

Input t

o1
o2
o3
o4

t.map(f)

f(o1) = [o1', o2', o3']

f(o2) = []

f(o3) = [o4']

f(o4) = [o5'

o1'
o2'
o3'
o4'
o5'

flatMap: Example

Input: a collection of sentences

“ACT I"
“SCENE I. Before LEONATO'S house."
“But few of any sort, and none of name.”
…

flatMap(split)

Output: a collection of words

“ACT”
“I”
“SCENE”
“I.”
“Before”
“LEONATO’S"
“house”
….

Group(By)

- Input: Collection of (k, v) pairs, Output: Collection of (k, [v]) pairs
  - k = key, v = value
  - So also works on Relations as long as you specify the key (i.e., groupby columns)
- Group the input rows by the "key"
- Relational Algebra: No support (can't have sets as values)
- SQL: Most implementations support it
  - e.g., postgresql has array aggregates or string aggregates
- Other Names: "nest"

Input t

k1	o1
k1	o2
k2	o3
k3	o4

t.groupByKey()

k1	[o1, o2]
k2	[o3]
k3	[o4]

GroupBy Example

Input: a collection of user ratings

user1 product1 5.0
user1 product2 1.0
user1 product3 5.0
user2 product26 5.0

map(f), where:

```
def f (s):
    splits = s.split(" ")
    return (splits[0],
            (splits[1], splits[2]))
```

(“user1”, (“product1”, 5.0))
(“user1”, (“product2”, 1.0))
(“user1”, (“product3”, 5.0))
(“user2”, (“product26”, 5.0))
….

groupByKey

(“user1”, [(“product1”, 5.0), (“product2”, 1.0), (“product3”, 5.0)])
(“user2”, [(“product26”, 5.0)])
….

Group(By) Aggregates

- Input: Collection of (k, v) pairs, Output: Collection of (k, v) pairs
- Also called “reduceByKey” or “aggregateByKey”
  - Need to specify a “reduce” or “aggregate” function
- Group values by key, and apply a provided “function” to get a single value
- SQL has a predefined set of functions (SUM, COUNT, MAX, …)

Input t

k1	o1
k1	o2
k2	o3
k3	o4

t.reduceByKey(f)

k1	f(o1, o2)
k2	f(o3)
k3	f(o4)

Group(By) Aggregates

- PostgreSQL (and other systems) support user-defined aggregate functions
  - init(): what's the initial state (e.g., for AVG: (count = 0, sum = 0))
  - update(): modify state given a new value (e.g., for AVG: (count + 1, sum + newval))
  - final(): generate the final aggregate (e.g., for AVG: sum/count)
  - The update operation must be insensitive to the order in which the values are processed
  - i.e., output should be the same if it sees: v1, v2, v3, versus if it sees: v3, v2, v1 in that order
  - Must process tuples sequentially

```
s = init()
s = update(s, v1)
s = update(s, v2)
…
result = final(s)
```

- Another way to do it
  - Provide a binary function that is commutative and distributive
  - Shouldn't matter in which order the objects are processed
  - More "parallelizable"
  - Can generate a (sum, count) pair, but for "average" need another "map"

```
Any of these are fine
result = f(f(f(v1, v2), f(v3, v4)), v5)
result = f(f(f(v1, f(v2, v3)), v4)), v5)
result = f(f(f(f(f(v1, v2), v3), v4), v5)
result = f(f(v1, f(v2, v3)), f(v4, v5))
```

Unnest

- Opposite of “nest” (group by)
- Similar to “flatMap” and “unwind” (in MongoDB)
  - But defined for relational algebra (extended to handle sets as values)
- Useful abstraction to deal with non-1NF data (e.g., JSON which supports arrays)

Table t

k1	[o1, o2]
k2	[o3]
k3	[o4]

unnest

k1	o1
k1	o2
k2	o3
k3	o4

Set Union/Intersection/Difference

- Input: Two collections, Output: One collection
- SQL support: union/except/intersection
  - Requires collections (tables) to have the same schema
  - Removes duplicates by default (most SQL operations don't remove duplicates)
  - Can use "union all" etc., to preserve duplicates

Relation r, s

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

r ∪ s:

A	B
α	1
α	2
β	1
β	3

r − s:

A	B
α	1
β	1

Cartesian Product

- Input: Two tables, Output: One table
- Note: a “set” product will result in nested output
  - First row would be: ((alpha, 1), (alpha, 10, a))
  - Relation algebra flattens it

Relation r, s

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

r × s:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Cartesian Product

- Input: Two tables, Output: One table
- Note: a “set” product will result in nested output
  - First row would be: ((alpha, 1), (alpha, 10, a))
  - Relation algebra flattens it

o1
o2
o3
o4

×

x1
x2
x3

=

(o1, x1)
(o1, x2)
(o1, x3)
(o2, x1)
(o3, x2)
(o4, x3)
…

Joins

- Input: Two tables, Output: One table
- Cartesian product followed by a “select”/“map”
- Many variations of joins used in database literature
  - Note: semi-join and anti-join are technically “select” operations on “r”, not a ”join” operation

Tables: r(A, B), s(B, C)

name	Symbol	SQL Equivalent	RA expression
cross product	$\times$	select * from r, s;	$r \times s$
natural join	$\bowtie$	natural join	$\pi_{r.A,\ r.B,\ s.C}\sigma_{r.B\ =\ s.B}(r\ x\ s)$
theta join	$\bowtie_\theta$	from .. where θ;	$\sigma_\theta(r\ x\ s)$
equi-join	$\bowtie_\theta$ *(theta must be equality)*		
left outer join	$r \mathbin{⟕} s$	left outer join (with “on”)	(see previous slide)
full outer join	$r \mathbin{⟗} s$	full outer join (with “on”)	–
(left) semijoin	$r \ltimes s$	none	$\pi_{r.A,\ r.B}(r \bowtie s)$
(left) antijoin	$r \triangleright s$	none	$r - \pi_{r.A,\ r.B}(r \bowtie s)$

Lookup

- Another way to look at a left outer join
- Input: Table, Output: Table
- Augment the input table with data from another table
- Supported by “Excel”, MongoDB, etc.

Table R

A	B	C
a1	b1	c1
a2	b2	c2
a3	b1	c1

Lookup in S using “C”

A	B	C	D	E
a1	b1	c1	d1	e1
a2	b2	c2	d2	e1
a3	b1	c1	d1	e1

C	D	E
c1	d1	e1
c2	d2	e1
c3	d2	e3

Table S

C must be a “key” for S,
o/w not well-defined

Pivot

- Flip rows and columns
- No SQL equivalent, although supported by many systems (e.g., CROSSTAB in PostgreSQL)
- Usually used in conjunction with aggregates (so that the number of rows is small)

Table R

A	B	C
a1	b1	c1
a2	b2	c2
a3	b1	c1

pivot

a1	a2	a3
b1	b2	b1
c1	c2	c1

Operations on Datasets

- Sorting and ordering
- Ranking (sparse vs dense rank)
- Distinct (duplicate elimination)
- Sample: generate a random sample
- Data Cubes
  - Allows aggregating on multiple attributes simultaneously

Recap

- Many data management systems view data as collection/multiset of tuples or objects or (key, value) pairs
- A common set of operations supported by most
  - Some Unary, Some Binary (or more generally, n-ary)
- Language constructs often map one-to-one to physical operators, but not always
- More declarative a language ➔ More opportunities to optimize
  - e.g., Pandas (Python Library), MongoDB, Apache Spark RDD interface, etc, not declarative even though pretty high-level
  - However, physical operators themselves can be heavily optimized, especially in parallel settings

Questions

- I have a dataset of images – each image as a separate file – I would like to associate each of them with a set of labels using an ML algorithm
- Dataset of Social Media Posts. Want to create “vectors” for each post.
- Dataset of tweets (each tweet is a separate object). Find the most common hashtags.
- Files containing web server logs. Need to find all 404s (i.e., people visiting links that don’t exist).
- Dataset of genomic sequences. Given a list of patterns (e.g., AGCTG), find the number of occurrences in each sequence.
- Dataset of Scientific Articles (i.e., PDFs). Find the average number of citations per article.

CMSC424: Database Design

Module: NoSQL; Big Data Systems

Parallel/Distributed Architectures; Data Replication; Sharding; Failures

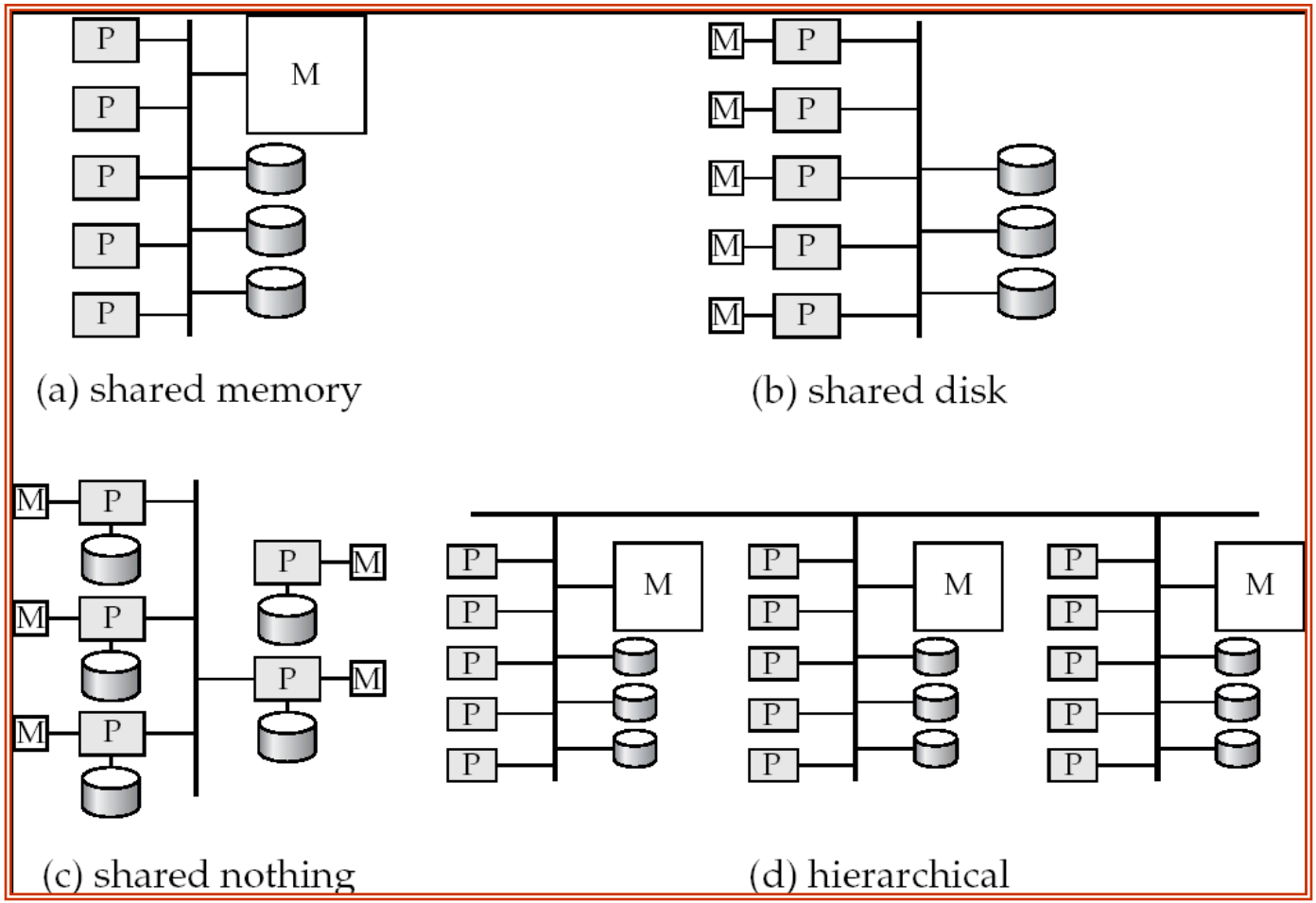
Instructor: Amol Deshpande
amol@umd.edu

Parallel and Distributed Architectures

- Ability to scale “up” a computer is limited ➔ Use many computers together
  - Called cluster or network of computers (and today, just a “data center”)
- Also need to ”meet” where the users are
  - To minimize interactive latencies (e.g., social networks)
- Has made parallel and distributed architectures very common today

Parallel Architectures

- Shared-nothing vs. shared-memory vs. shared-disk



Parallel Architectures

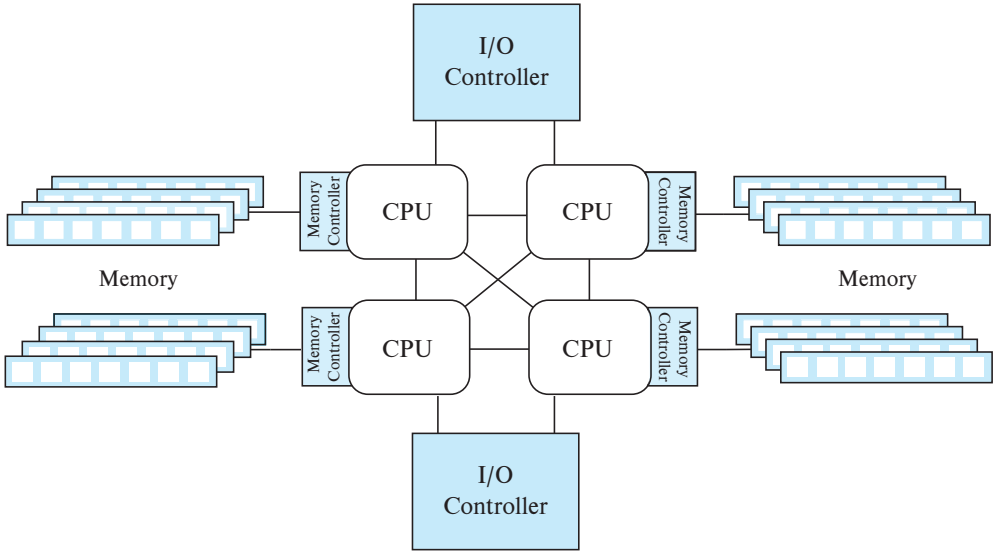


Figure 20.6 Architecture of a modern shared-memory system.

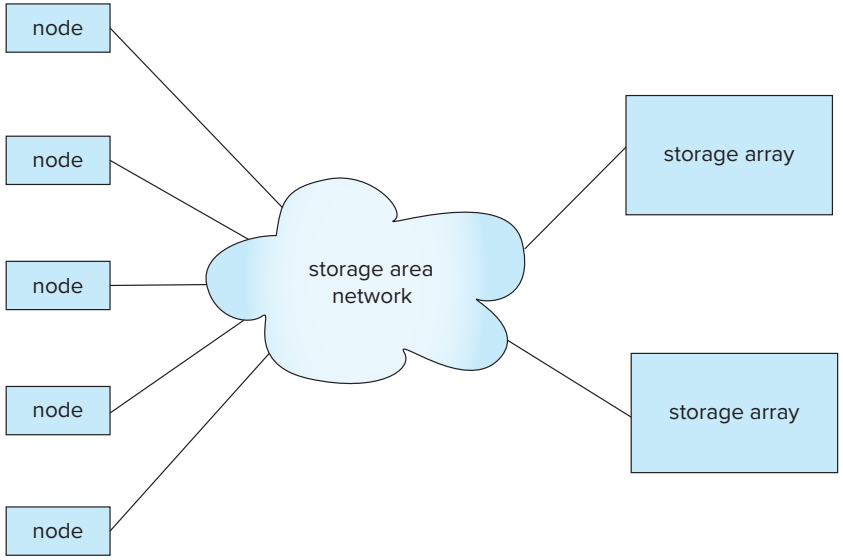


Figure 20.8 Storage-area network.

Parallel Architectures

	Shared Memory	Shared Disk	Shared Nothing
Communication between processors	Extremely fast	Disk interconnect is very fast	Over a LAN, so slowest
Scalability ?	Not beyond 32 or 64 or so (memory bus is the bottleneck)	Not very scalable (disk interconnect is the bottleneck)	Very very scalable
Notes	Cache-coherency an issue	Transactions complicated; natural fault-tolerance.	Distributed transactions are complicated (deadlock detection etc);
Main use	Low degrees of parallelism	Not used very often	Everywhere

Parallel Systems

- A **coarse-grain parallel** machine → a small number of powerful processors
- A **massively parallel** or **fine grain parallel** machine → thousands of smaller processors.
- We see a variety of mixes of these today, especially with the rise of multi-core machines

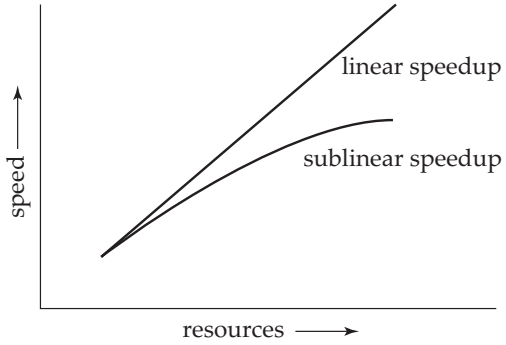
- Two main performance measures:
  - **throughput** --- the number of tasks that can be completed in a given time interval
  - **response time** --- the amount of time it takes to complete a single task from the time it is submitted

Speed-Up and Scale-Up

- **Speedup**: a fixed-sized problem executing on a small system is given to a system which is *N*-times larger.
  - Measured by:

    $$speedup = \frac{small\ system\ elapsed\ time}{large\ system\ elapsed\ time}$$

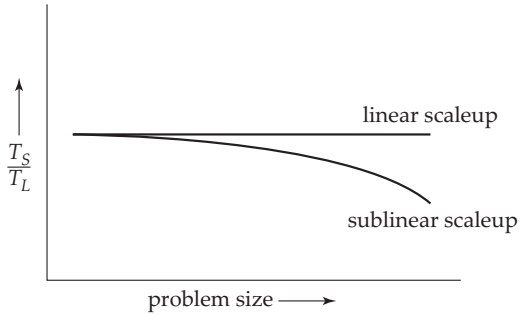
  - Speedup is **linear** if equation equals N.



- **Scaleup**: increase the size of both the problem and the system
  - *N*-times larger system used to perform *N*-times larger job
  - Measured by:

    $$scaleup = \frac{small\ system\ small\ problem\ elapsed\ time}{big\ system\ big\ problem\ elapsed\ time}$$

  - Scale up is **linear** if equation equals 1.

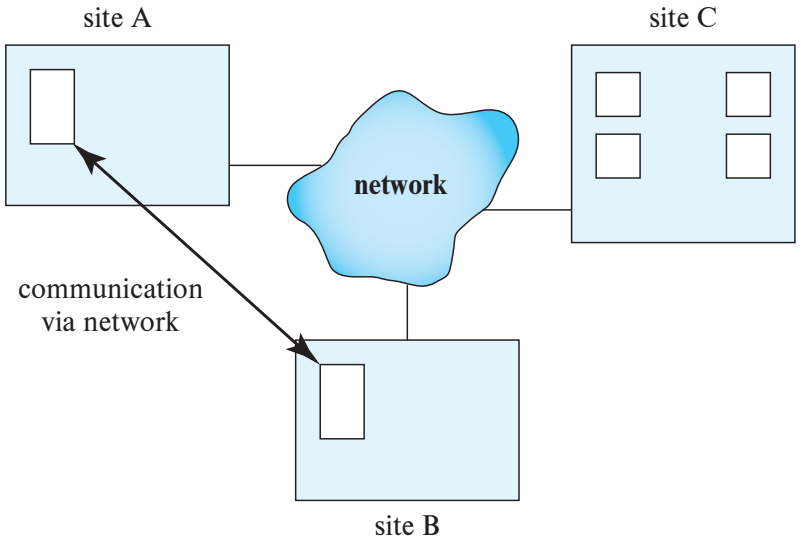


Factors Limiting Speedup and Scaleup

- **Sequential computation:** Some parts may not be parallalelizable
  - **Amdahl's Law**: If "p" is the fraction of the task that can be parallelized, then the best speedup you can get is: $\frac{1}{(1-p)+(p/n)}$.
  - If "p" is 0.9, the best speedup is 10
- **Startup costs**: Cost of starting up multiple processes may dominate computation time, if the degree of parallelism is high.
- **Interference**: Processes accessing shared resources (e.g., system bus, disks, or locks) compete with each other, thus spending time waiting on other processes, rather than performing useful work.
- **Skew**: Increasing the degree of parallelism increases the variance in service times of parallely executing tasks. Overall execution time determined by **slowest** of parallely executing tasks.

What about "Distributed" Systems?

- Over a wide area network
- Typically not done for *performance reasons*
  - For that, use a parallel system
- Done because of necessity
  - Imagine a large corporation with offices all over the world
  - Or users distributed across the globe
  - Also, for redundancy and for disaster recovery reasons



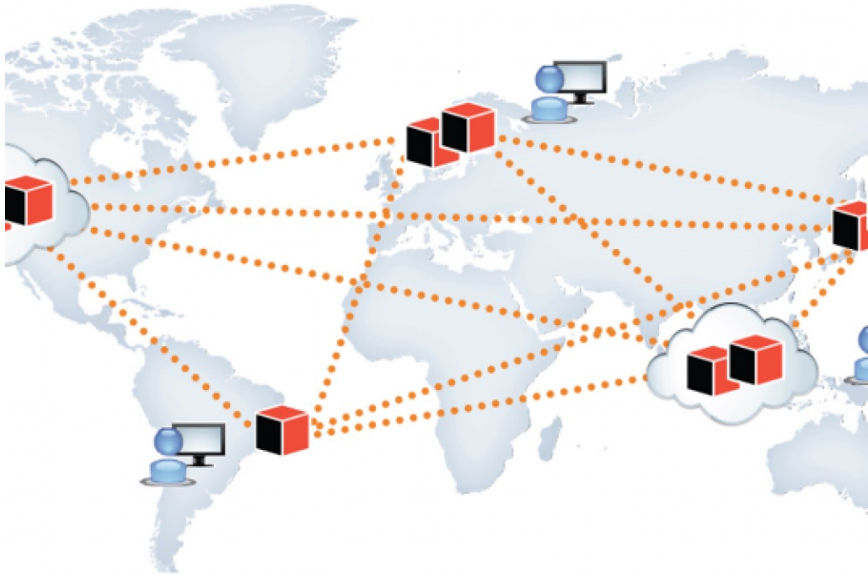


Figure 20.9 A distributed system.

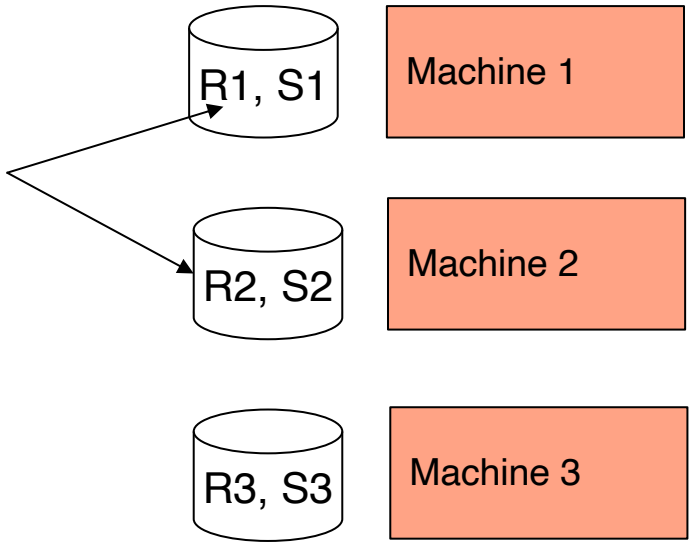
Parallel or Distributed Systems

- Key Questions from Data Management Perspective:
  - How to partition (or “shard”) data across a collection of storage devices/machines
  - How to execute an “operation” across a group of computers
    - In different configurations (shared-memory vs shared-disk vs shared-nothing vs NUMA)
    - Trade-offs and bottlenecks can be vastly different
  - How to execute an “update” across a group of computers
    - Need to ensure consistency
  - How to deal with “failures”

Data Partitioning

- Partition a relation or a dataset across machines
  - Typically through “hashing”
- Advantages:
  - **In-memory computation**: data fits in memory across machines
  - **Parallelism**: simple read/write queries can be distributed across machines
- Disadvantages:
  - **Complex queries:** require combining data across all partitions, especially “joins” are tricky

Partitions of R (Not different relations)

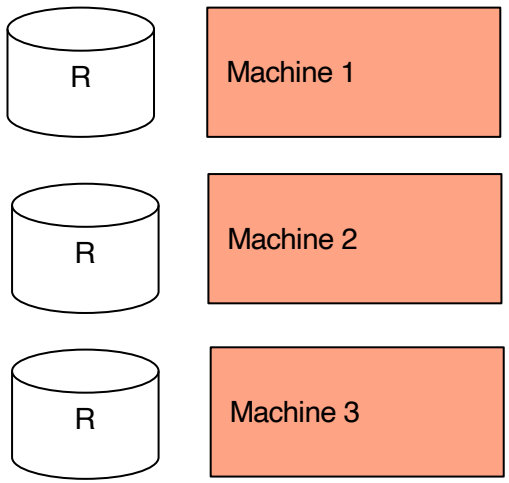


Machine 1 can directly read R1, S1

If it wants R2, Machine 2 must read it and send it to Machine 1

Data Replication

- A data item (file, relation, relation fragment, object, tuple) is **replicated** if it is stored redundantly in two or more sites
- Advantages:
  - **Availability**: failures can be handled through replicas
  - **Parallelism**: queries can be run on any replica
  - **Reduced data transfer**: queries can go to the "closest" replica
- Disadvantages:
  - **Increased cost of updates:** both computation as well as latency
  - **Increased complexity of concurrency control:** need to update all copies of a data item/tuple



Read queries can go to any machine

Write queries must go to "all" machines (if we want consistency)

e.g., what if Application 1 writes to Machine 1, and Application 2 sends its write to Machine 3
-- May result in an inconsistent state

Data Sharding + Replication

- Many data management systems today combine both
  - Partition a dataset/file/relation into smaller pieces and distributed it across machines
  - Replicate each of the pieces multiple times
- This may be done:
  - In a data center with very fast networks, or
  - In a wide-area setting with slower networks and higher latencies
- So need to worry about:
  - Efficient execution of complex queries
  - Consistency for updates
  - Recovery from failures

Failures

- Need to consider:
  - Disk failures: one of the disks (hard drives or SSDs) fails
    - Not uncommon with 10's of thousands of disks
  - Network failures: machines may not be able to talk to each other
  - Machine failure: a machine crashes during the execution of a query or a transaction
- Required guarantees:
  - Shouldn't lose any data if a disk fails
  - Consistency (when making updates) shouldn't be affected if one of the involved machines fails
    - Or if machines are not able to talk to each other
  - Shouldn't have to restart a complex analytics task entirely if one of the involved machines fails

CMSC424: Database Design

Module: NoSQL; Big Data Systems

Parallelizing Operations

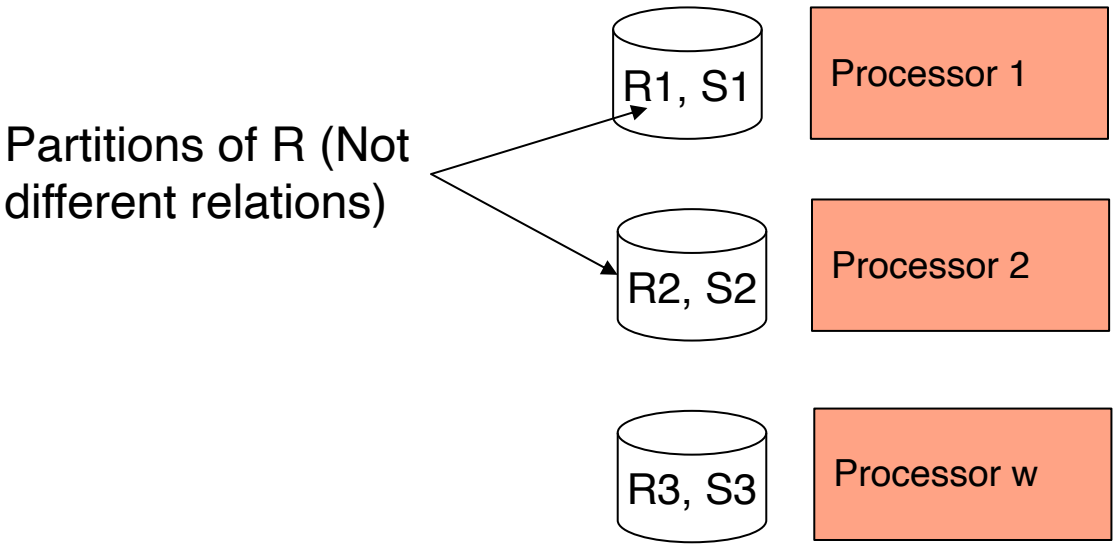
Instructor: Amol Deshpande
amol@umd.edu

Parallelizing Operations

- Book Chapters
  - 18.5, 18.6
- Key topics:
  - Parallelizing a Sort Operation
  - Parallelizing a Join Operation
  - Parallelizing a Group By Operation

Setup

- Assume Shared-Nothing Model
- Relations are already partitioned across a set of machines (will talk about how next video)
- How to execute different operations?



Processor 1 can directly read R1, S1

If it wants R2, Processor 2 must read it and send it to Processor 1

Parallel Sort

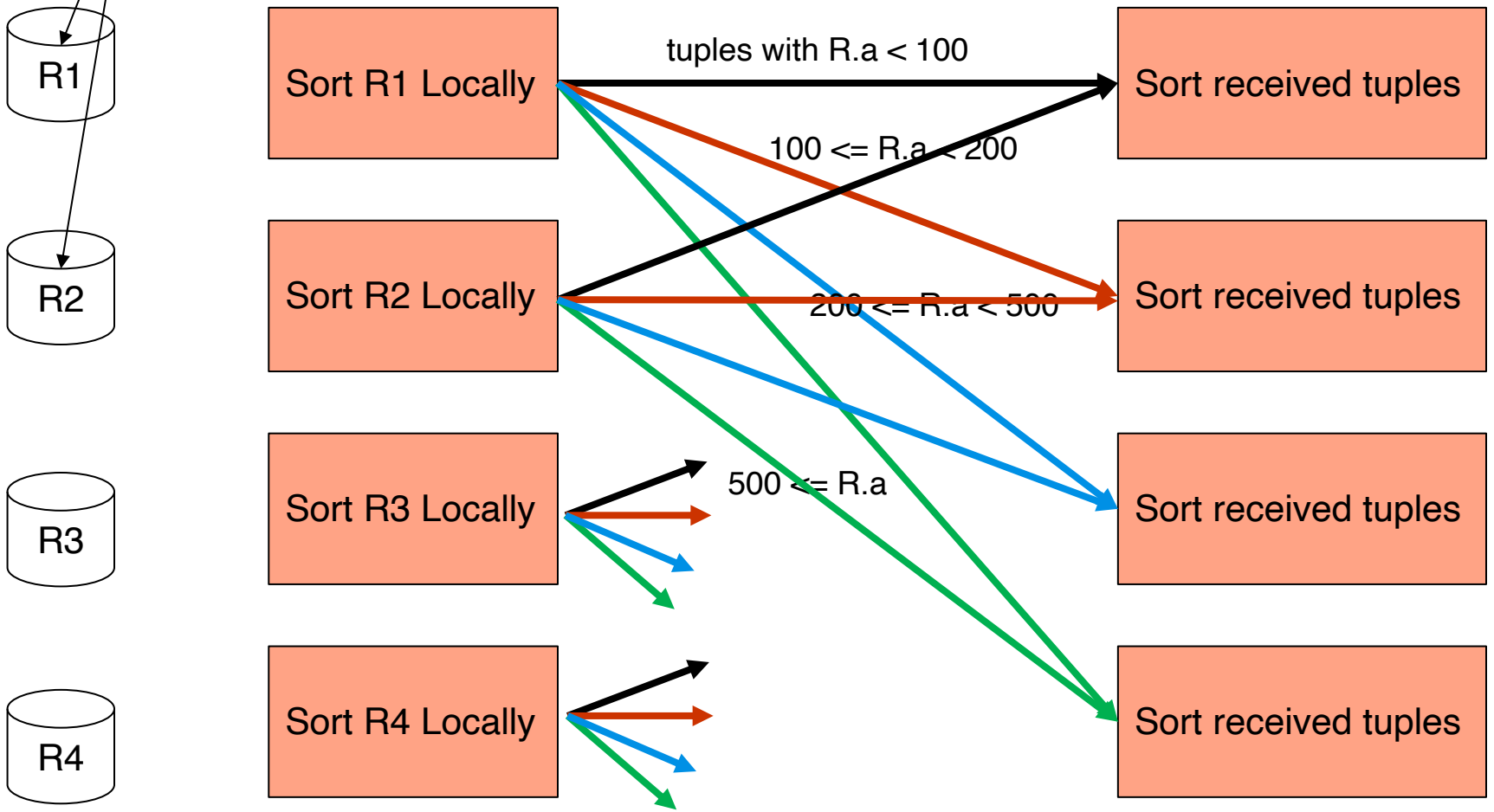
1. Each processor sorts a portion of the data (e.g., the data on their local disk)
2. If the data is small enough, all the processors can send it to a single machine to do a “merge”
3. If the data is large, then ”merge” itself done in parallel through range partitioning
    1. Each processor in the merge phase gets assigned a range of the data
    2. All other processors send the appropriate data based on that range partitioning

- In either phase, the processors work by themselves (“data parallelism”) but data must be “shuffled” in between
- Other approaches exist, but basically same steps

Parallel Sort

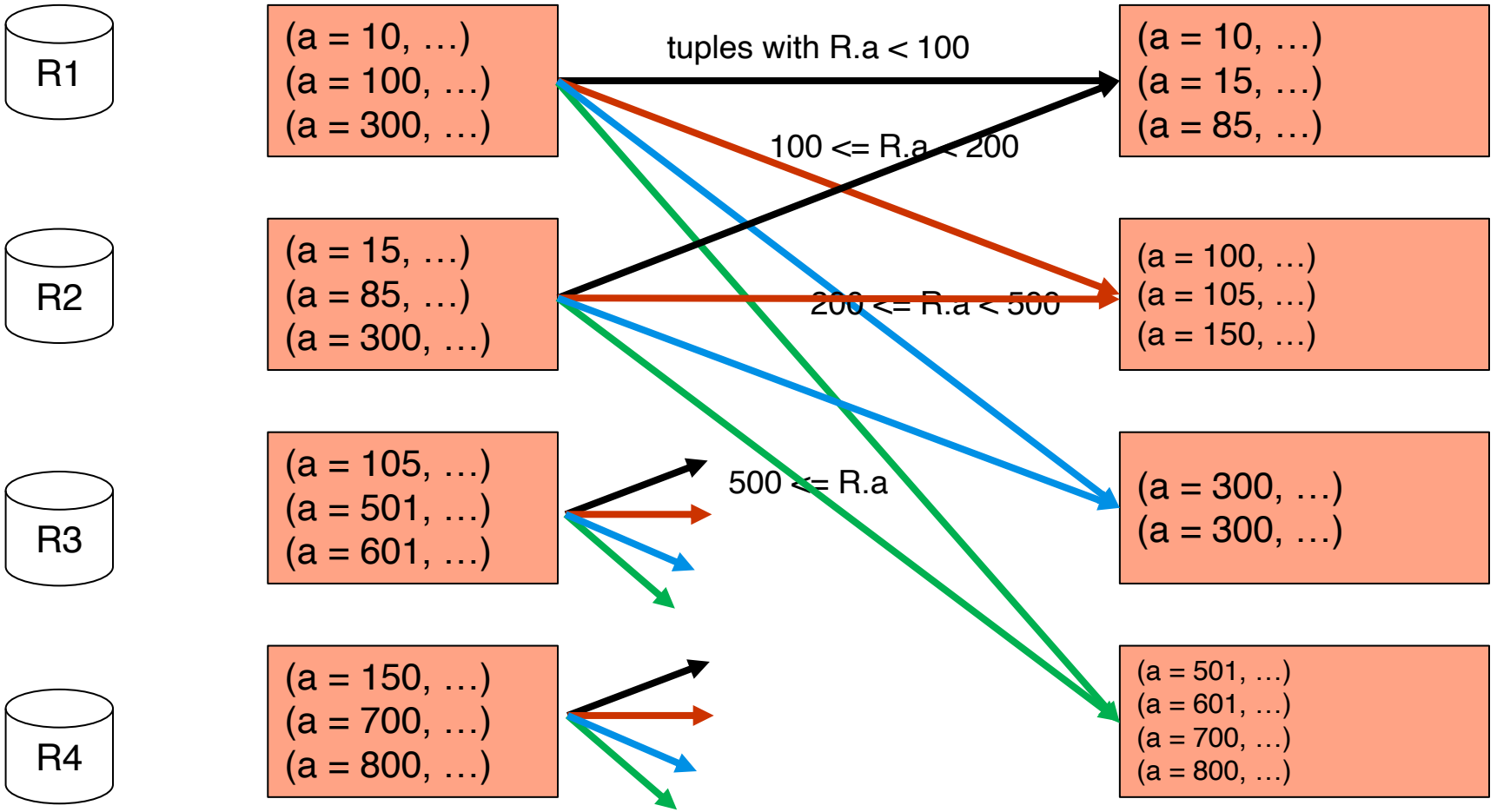
Partitions of R (Not different relations)

Can be same machines or different



Shuffle – typically expensive

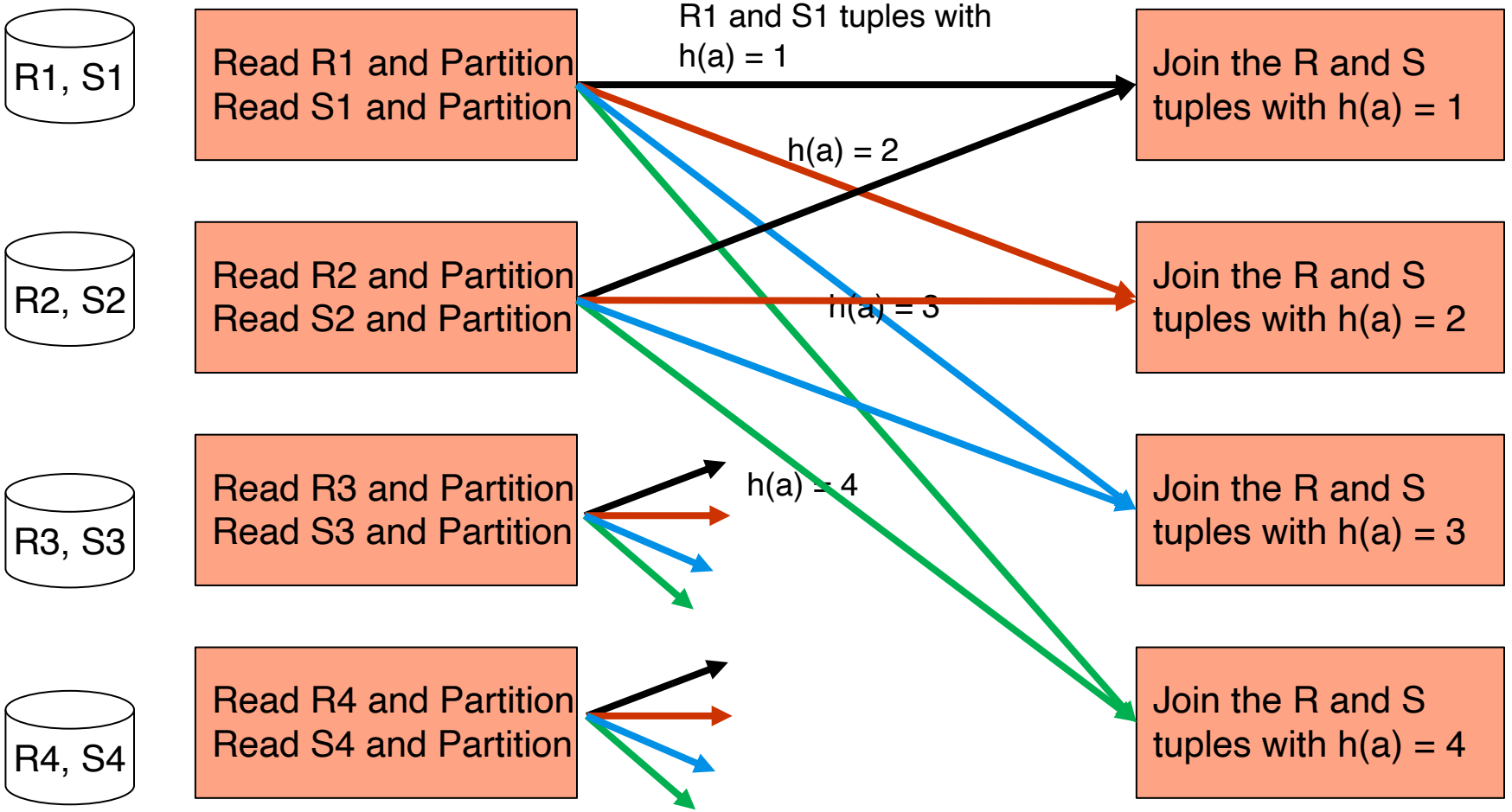
Parallel Sort



Parallel Join

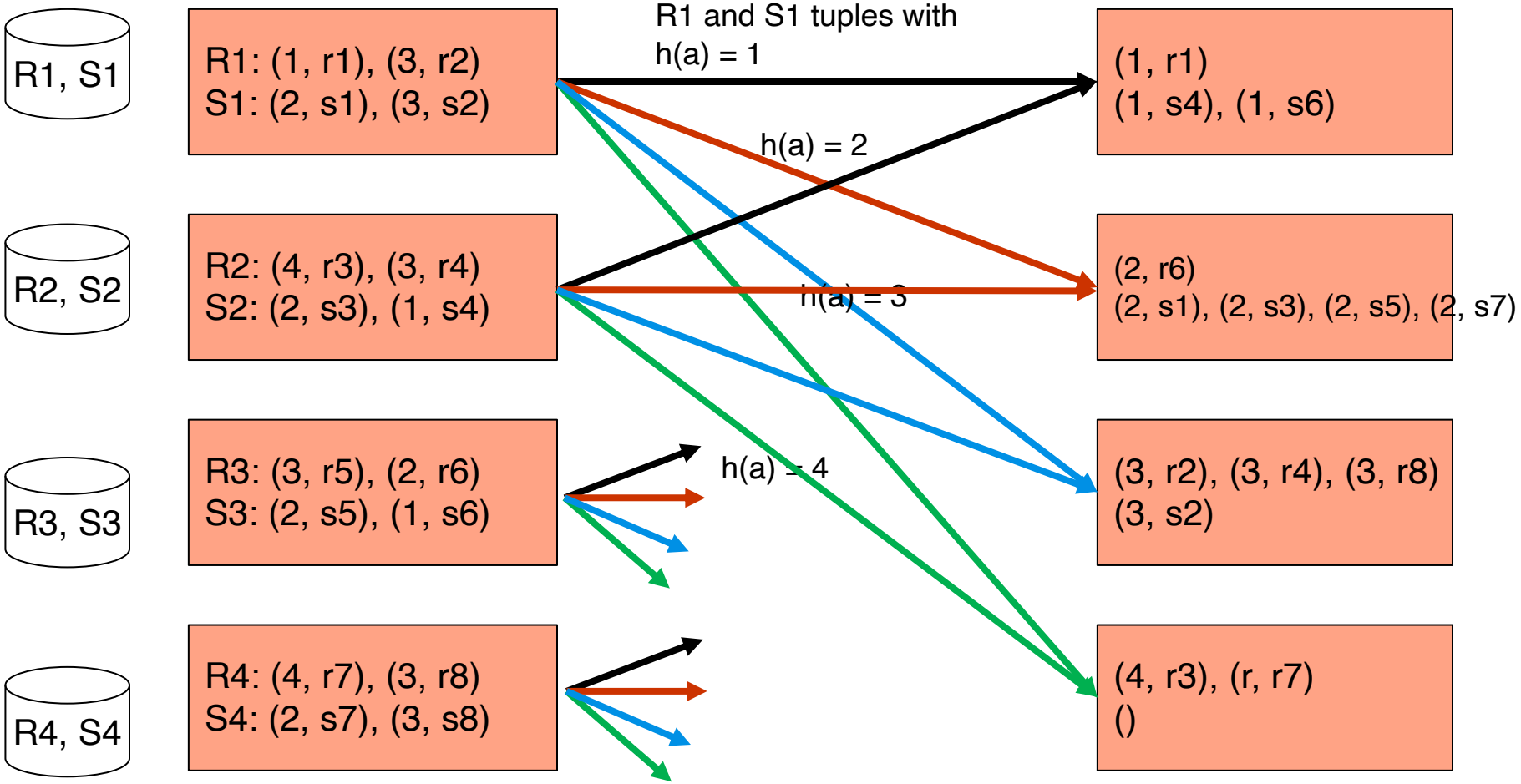
- Hash-based approach
  - Very similar to how partitioning hash join works (i.e., the variant we saw for the case when the relations don't fit in memory)
  - Most common for equi-joins where hashing can be used
  - Easier to guarantee balanced work
- Sort-based approach
  - Similar to the parallel sort approach
  - Both relations sorted using the same key
  - Same processor used for merging in the second phase for both relations

Hash-based Parallel Join



Shuffle – typically expensive

Hash-based Parallel Join

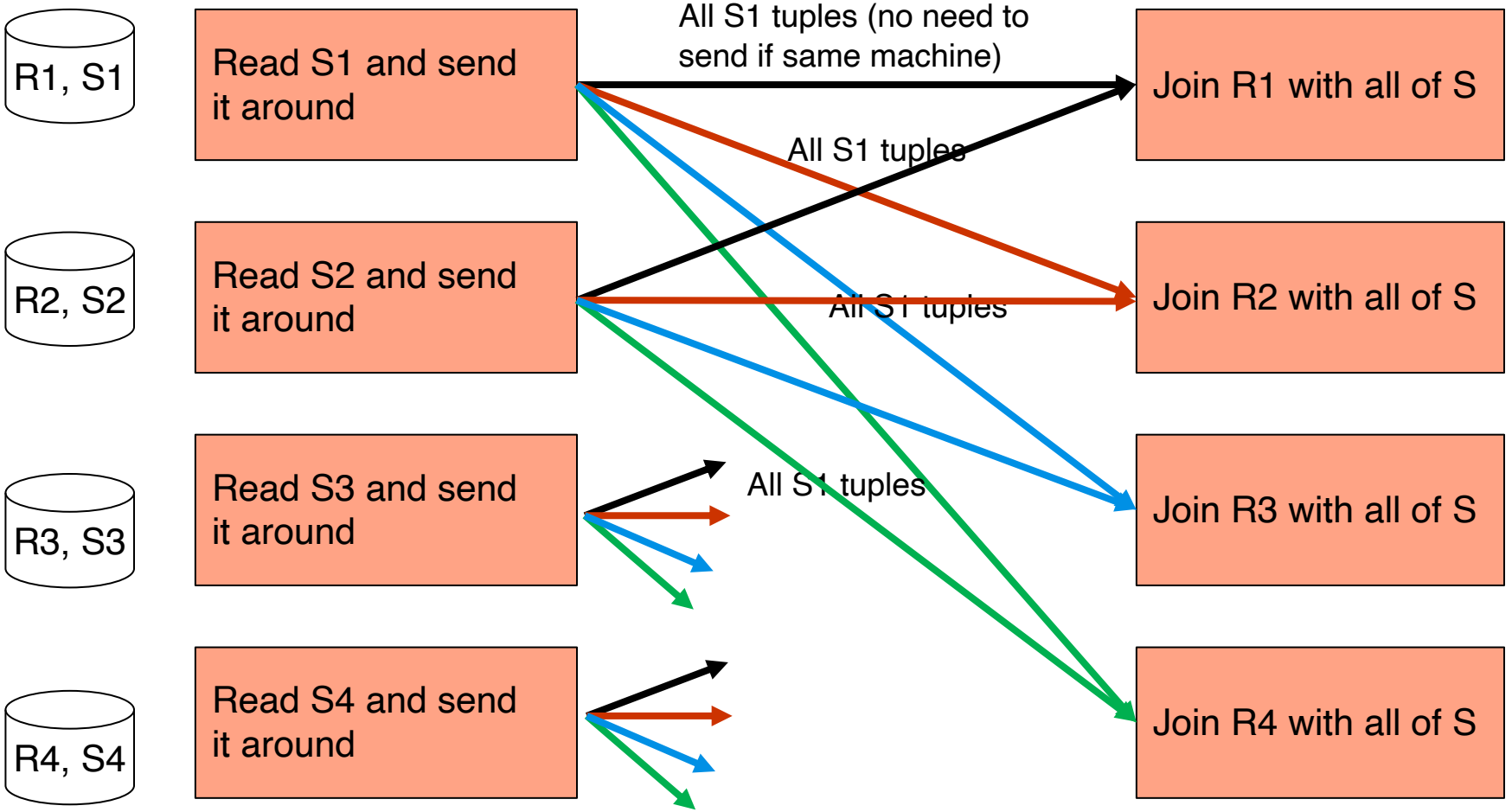


Shuffle – typically expensive

Fragment-and-Replicate Join

- Partitioning not possible for some join conditions
  - E.g., non-equijoin conditions, such as r.A > s.B.
- For joins were partitioning is not applicable, parallelization can be accomplished by **fragment and replicate** technique
- Special case – **asymmetric fragment-and-replicate**:
  - One of the relations, say *r*, is partitioned; any partitioning technique can be used.
  - The other relation, *s*, is replicated across all the processors.
  - Processor P_i then locally computes the join of r_i with all of s using any join technique.

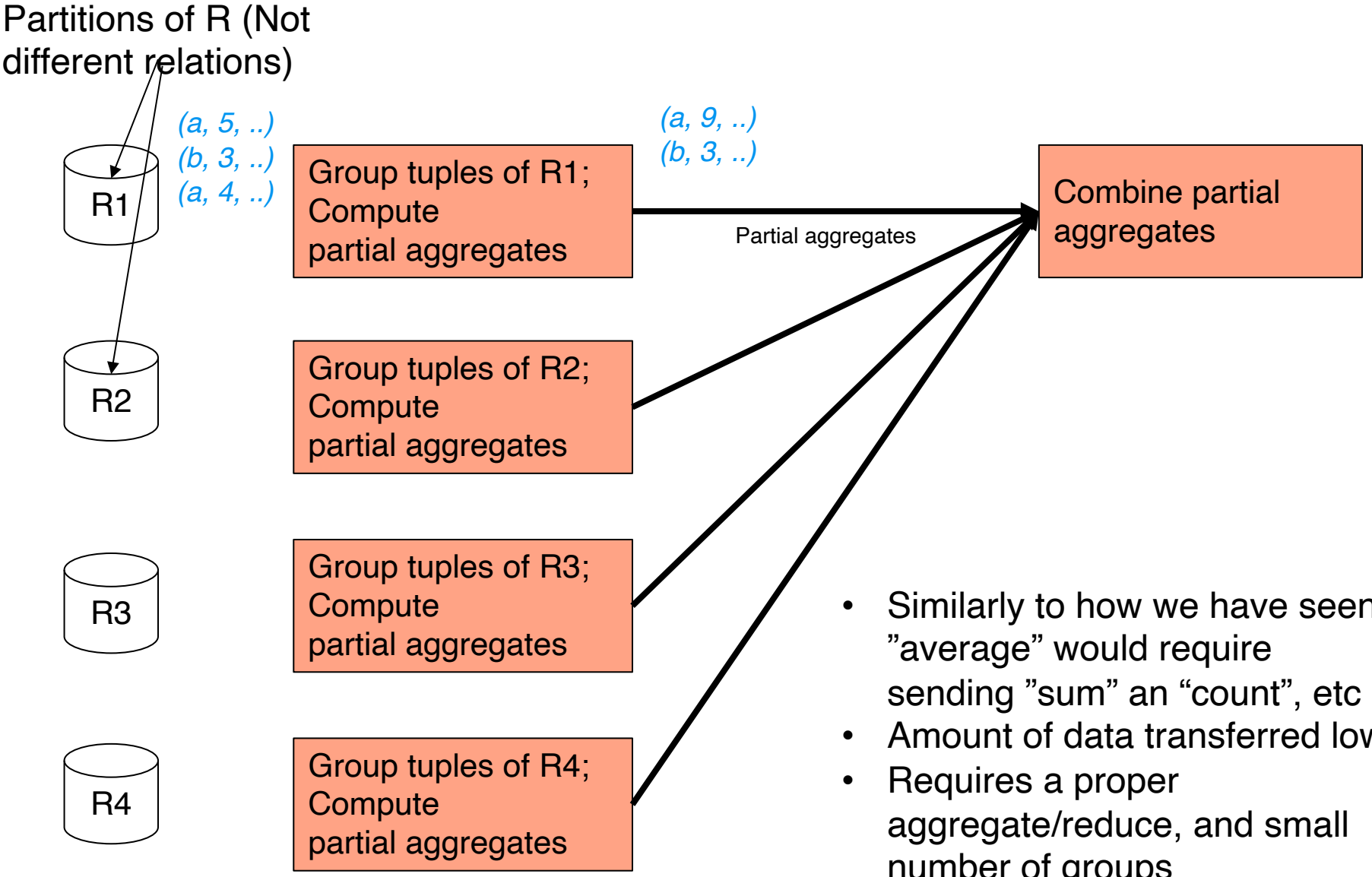
Asymmetric Fragment and Replicate



Grouping/Aggregation

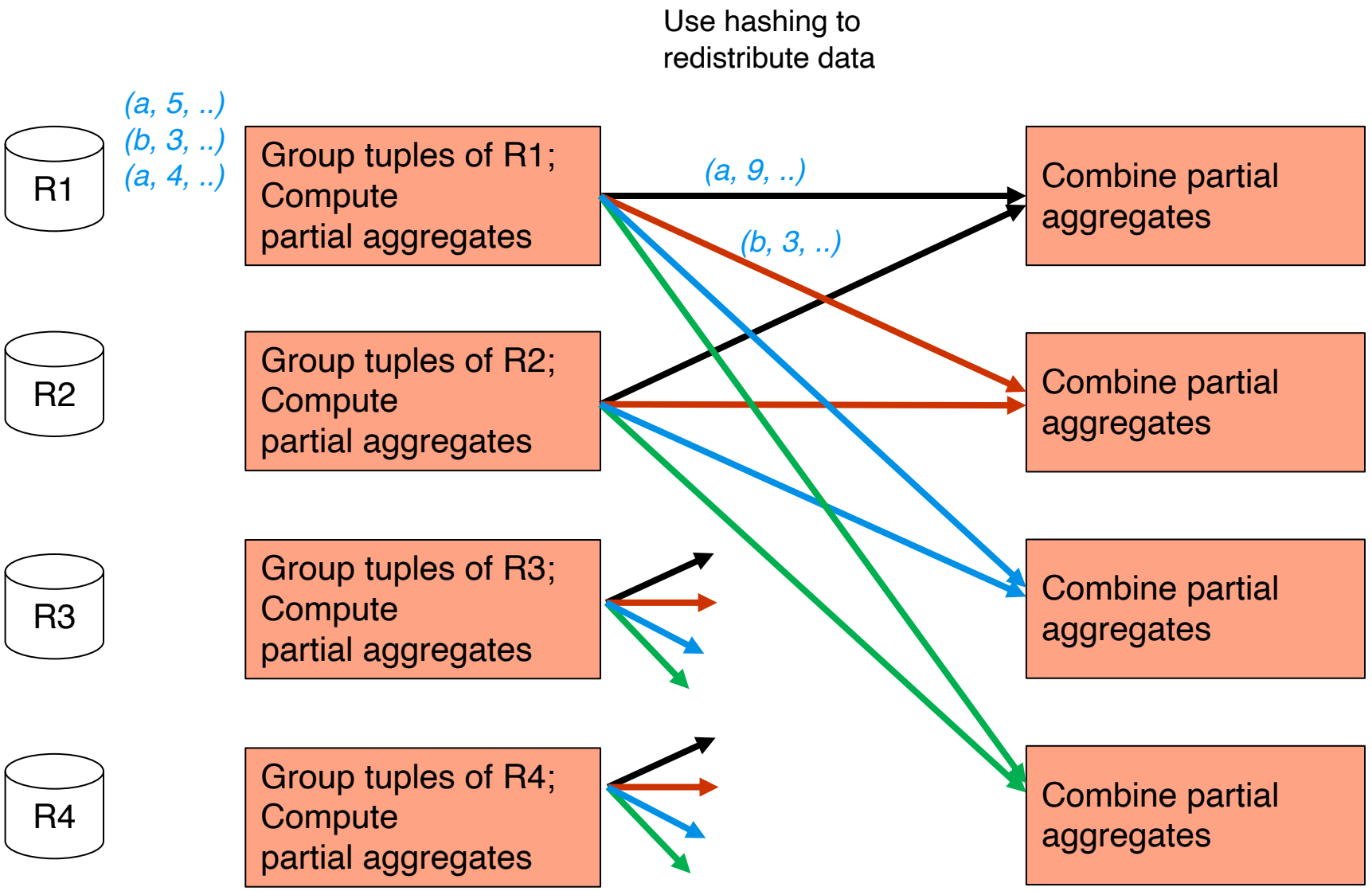
- Very common operation, especially in Map-Reduce applications
  - E.g., grouping by “hostnames” or “words” (as in project 5) or “labels” (in ML context), etc.
  - The idea of distributing data, doing some computations, and collecting results is quite powerful
  - Even ”joins” can be seen as a “group by” operation (you can group the tuples of the two relations on the join attribute, and then compute join)
- Need to differentiate between “groupby” and “aggregate” (”reduce”)
  - **Groupby:** For every value of ”group by attribute” (i.e., “key”), collect all tuples/records with that key on a single machine
  - **Aggregate/Reduce:** Perform some computation on them, typically reducing the size of the data
  - Spark has more granular operations than SQL
- Challenges:
  - Number of keys might be very large
  - Should try to do as much pre-aggregation as possible

Scenario 1: Small # of Groups + Reduce



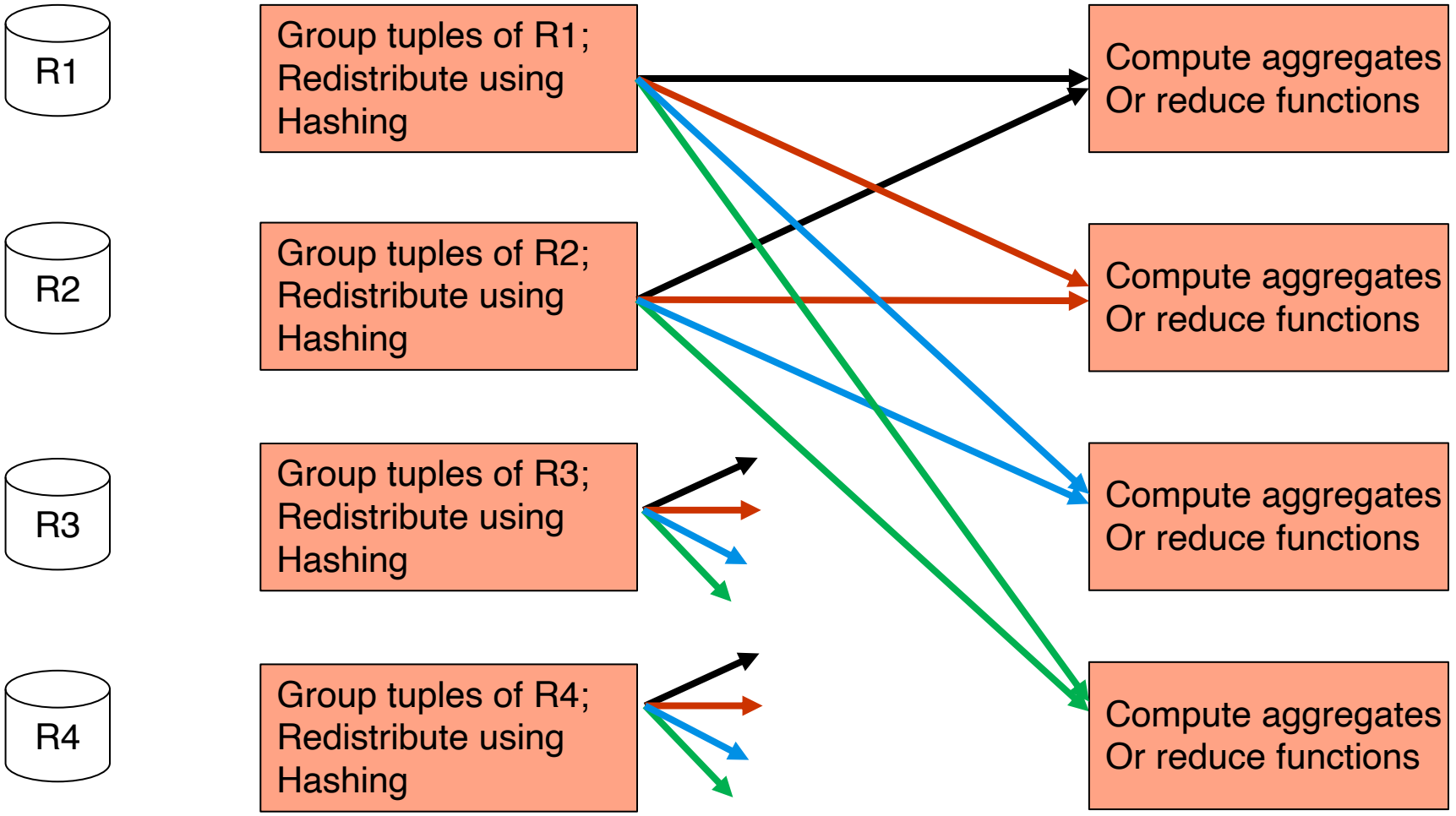
- Similarly to how we have seen, ”average” would require sending ”sum” an “count”, etc
- Amount of data transferred low
- Requires a proper aggregate/reduce, and small number of groups

Scenario 2: Large # of Groups + Reduce



Scenario 3: Large # of Groups + No Reduce

e.g., if we want to compute "median" or some other complex statistics – no "partial aggregation" possible



Other Relational Operations

Selection $\sigma_\theta(r)$

- If θ is of the form $a_i = v$, where a_i is an attribute and v a value.
  - If r is partitioned on a_i the selection is performed at a single processor.
- If θ is of the form $l <= a_i <= u$ (i.e., θ is a range selection) and the relation has been range-partitioned on a_i
  - Selection is performed at each processor whose partition overlaps with the specified range of values.
- In all other cases: the selection is performed in parallel at all the processors.

Other Relational Operations (Cont.)

- **Duplicate elimination**
  - Perform by using either of the parallel sort techniques
    - eliminate duplicates as soon as they are found during sorting.
  - Can also partition the tuples (using either range- or hash-partitioning) and perform duplicate elimination locally at each processor.
- **Projection**
  - Projection without duplicate elimination can be performed as tuples are read in from disk in parallel.
  - If duplicate elimination is required, any of the above duplicate elimination techniques can be used.

CMSC424: Database Design

Module: NoSQL; Big Data Systems

Apache Spark

Instructor: Amol Deshpande
amol@umd.edu

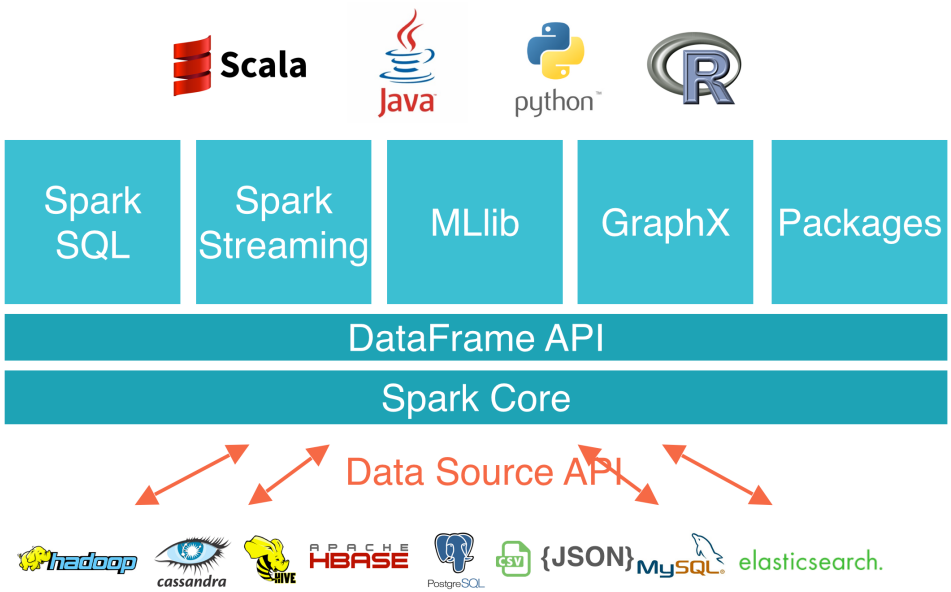
Apache Spark

- Book Chapters
  - 10.4 **(7TH EDITION)** covers this topic, but Spark programming guide is a better resource
  - Assignments will refer to the programming guide

- Key topics:
  - A Resilient Distributed Dataset (RDD)
  - Operations on RDDs

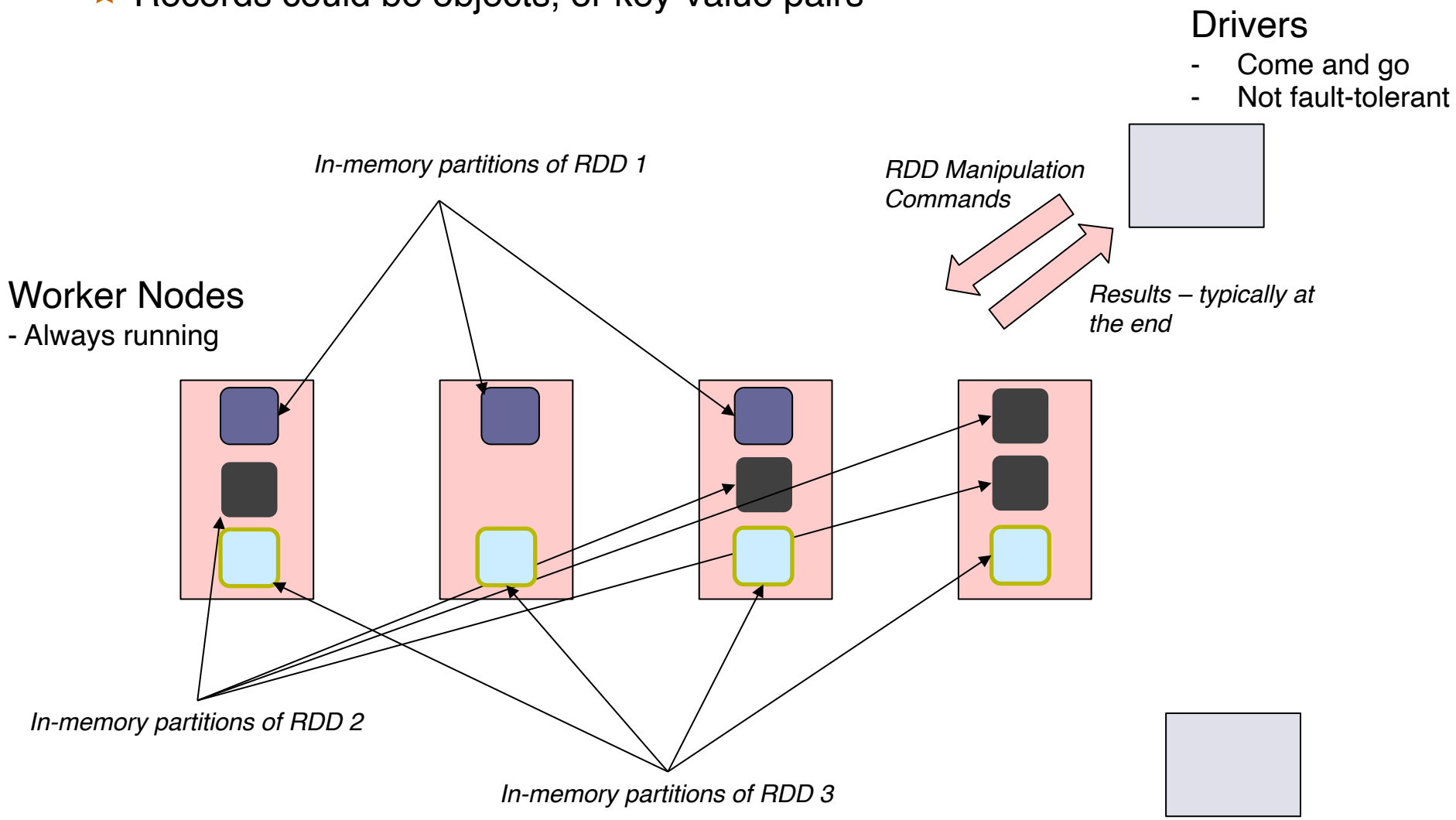
Spark

- Open-source, distributed cluster computing framework
- Much better performance than Hadoop MapReduce through in-memory caching and pipelining
- Originally provided a low-level RDD-centric API, but today, most of the use is through the “Dataframes” (i.e., relations) API
  - ★ Dataframes support relational operations like Joins, Aggregates, etc.



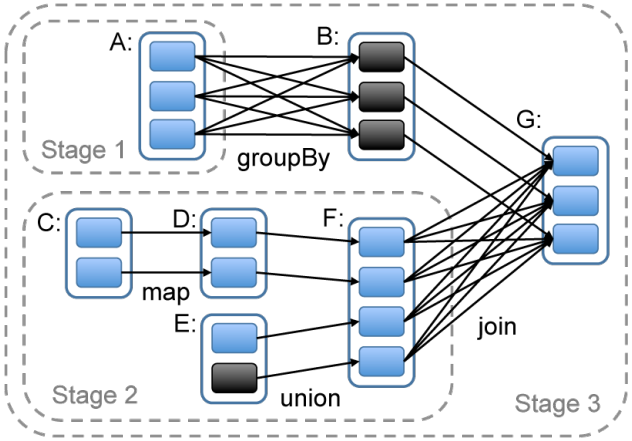
Resilient Distributed Dataset (RDD)

- **RDD =** Collection of records stored across multiple machines in-memory
  - ★ Records could be objects, or key-value pairs



Spark

- Why “Resilient”?
  - Can survive the failure of a worker node
  - Spark maintains a “lineage graph” of how each RDD partition was created
  - If a worker node fails, the partitions are recreated from its inputs
  - Only a small set of well-defined operations are permitted on the RDDs
    - But the operations usually take in arbitrary ”map” and “reduce” functions



- Fault tolerance for the “driver” is trickier
  - Drivers have arbitrary logic (cf., the programs you are writing)
  - In some cases (e.g., Spark Streaming), you can do fault tolerance
  - But in general, driver failure requires a restart

Example Spark Program

Initialize RDD by reading the textFile and partitioning
If textFile stored on HDFS, it is already partitioned – just read each partition as a separate RDD partition

Split each line into words, creating an RDD of words

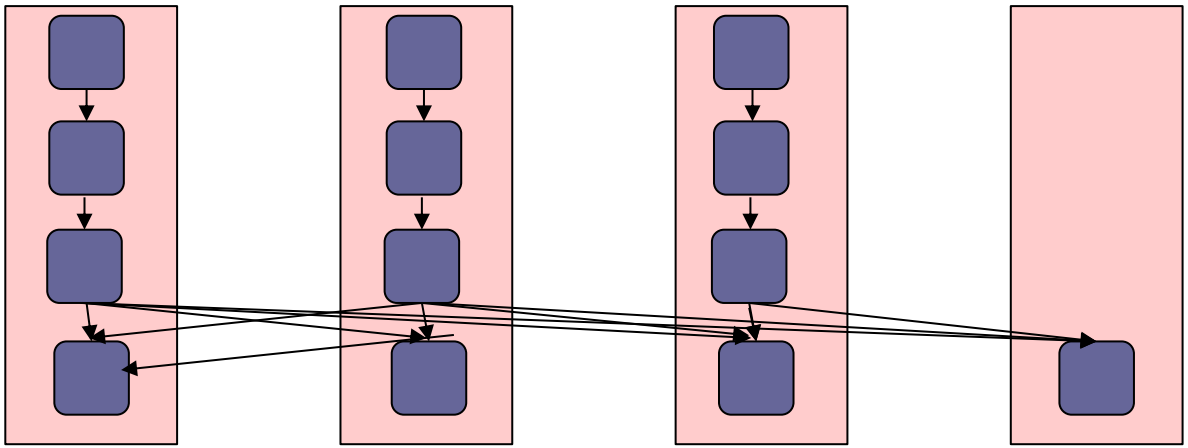
For each word, output (word, 1), creating a new RDD

Do a group-by SUM aggregate to count the number of times each word appears

Driver

```
from pyspark import SparkContext


sc = SparkContext("local", "Simple App")


textFile = sc.textFile("README.md")


counts = textFile
                    .flatMap(lambda line: line.split(" "))
                    .map(lambda word: (word, 1))
                    .reduceByKey(lambda a, b: a + b)


print(counts.take(100))
```

Retrieve 100 of the values in the final RDD



Spark

- Operations often take in a ”function” as input
- Using the inline “lambda” functionality

```
flatMap(lambda line: line.split(" "))
```

- Or a more explicit function declaration

```
def split(line):
      return line.split(" ")

flatMap(split)
```

- Similarly ”reduce” functions essentially tell Spark how to do pairwise aggregation

```
reduceByKey(lambda a, b: a + b)
```

  - Spark will apply this to the dataset pair of values at a time
  - Difficult to do something like “median”

Spark: Map

InputRDD: [x1, x2, …., x_n]

map(lambda x: x + 1)

def fn(x):
    return x+1
map(fn)

outputRDD: [x1+1, x2+1, …., x_n+1]

x1, x2, … can be anything, including documents, images, text files, tuples, dicts, etc.

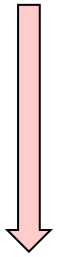
InputRDD: [x1, x2, …, x_n]

map(fn)

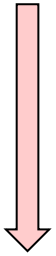
outputRDD: [fn(x1), fn(x2), …, fn(x_n)]

Spark: flatMap

InputRDD: [(a1, b1), (a2, b2), ….]

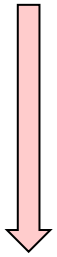


flatMap(lambda x: [x[0], x[1]])

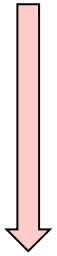


outputRDD: [a1, b1, a2, b2, …]

InputRDD: [‘the little brown fox…’, …]



flatMap(lambda x: x.split())

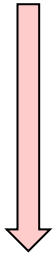


outputRDD: [‘the’, ‘little’, ‘brown’, …]

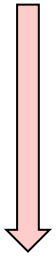
Spark: groupByKey

InputRDD: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]

InputRDD must be a collection of 2-tuples
Usually called (Key, Value) pairs
Value can be anything (e.g., dicts, tuples, bytes)



groupByKey()

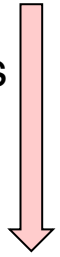


outputRDD: [(a1, [b1, b3, b4, …]), (a2, [b3, b5,…]), …]

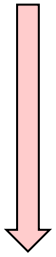
Spark: reduceByKey

InputRDD: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]

InputRDD must be a collection of 2-tuples
Usually called (Key, Value) pairs



reduceByKey(func)



def func(V1, V2):
return V3

All of V1, V2, and V3
be of the same type

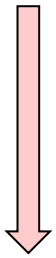
outputRDD: [(a1, …func(func(b1, b3), b4)…),
(a2, …func(func(b2, b5), …)…),]

”func” executed in parallel in a pairwise fashion

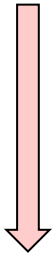
Spark: join

InputRDD1: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]
InputRDD2: [(a1, c1), (a2, c2), (a1, c3), (a1, c4), (a2, c5)…]

InputRDD1 and InputRDD2 both must be a collection of 2-tuples



inputRDD1.join(inputRDD2)



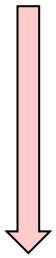
outputRDD: [(a1, (b1, c1)),
(a1, (b1, c3)),
(a1, (b1, c4)),
….]

Spark: cogroup

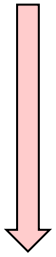
InputRDD1: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]

InputRDD2: [(a1, c1), (a2, c2), (a1, c3), (a1, c4), (a2, c5)…]

InputRDD1 and InputRDD2 both must be a collection of 2-tuples



inputRDD1.cogroup(inputRDD2)



outputRDD: [(a1, ([b1, b3, b4, …], [c1, c3, c4, …]),
(a2, ([b2, b5, …], [c2, c5, …]), …
]

RDD Operations

Transformations

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R) and pair RDD functions doc (Scala, Java) for details.

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*])	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean

Actions

The following table lists some of the common actions supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R)

and pair RDD functions doc (Scala, Java) for details.

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the `foreach()` may result in undefined behavior. See Understanding closures for more details.

Dataframes Example

```
def basic_df_example(spark):
    # $example on:create_df$
    # spark is an existing SparkSession
    df = spark.read.json("examples/src/main/resources/people.json")
    # Displays the content of the DataFrame to stdout
    df.show()
    # +----+-------+
    # | age|   name|
    # +----+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +----+-------+
    # $example off:create_df$


    # $example on:untyped_ops$
    # spark, df are from the previous example
    # Print the schema in a tree format
    df.printSchema()
    # root
    # |-- age: long (nullable = true)
    # |-- name: string (nullable = true)


    # Select only the "name" column
    df.select("name").show()
    # +-------+
    # |   name|
    # +-------+
    # |Michael|
    # |   Andy|
    # | Justin|
    # +-------+


    # Select everybody, but increment the age by 1
    df.select(df['name'], df['age'] + 1).show()
    # +-------+---------+
    # |   name|(age + 1)|
    # +-------+---------+
    # |Michael|     null|
    # |   Andy|       31|
    # | Justin|       20|
    # +-------+---------+


    # Select people older than 21
    df.filter(df['age'] > 21).show()
    # +---+----+
    # |age|name|
    # +---+----+
    # | 30|Andy|
    # +---+----+


    # Count people by age
    df.groupBy("age").count().show()
    # +----+-----+
    # | age|count|
    # +----+-----+
    # |  19|    1|
    # |null|    1|
    # |  30|    1|
    # +----+-----+
    # $example off:untyped_ops$

    sqlDF = spark.sql("SELECT * FROM people")
    sqlDF.show()
    # +----+-------+
    # | age|   name|
    # +----+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +----+-------+
    # $example off:run_sql$


    # $example on:global_temp_view$
    # Register the DataFrame as a global temporary view
    df.createGlobalTempView("people")


    # Global temporary view is tied to a system preserved database
`global_temp`
    spark.sql("SELECT * FROM global_temp.people").show()
    # +----+-------+
    # | age|   name|
    # +----+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +----+-------+
```

Summary

- Spark is a popular and widely used framework for large-scale computing
- Simple programming interface
  - You don't need to typically worry about the parallelization
  - That's handled by Spark transparently
  - In practice, may need to fiddle with number of partitions etc.
- Managed services supported by several vendors including Databricks (started by the authors of Spark), Cloudera, etc.
- Many other concepts that we did not discuss
  - Shared accumulator and broadcast variables
  - Support for Machine Learning, Graph Analytics, Streaming, and other use cases
- Alternatives include: Apache Tez, Flink, and several others

CMSC424: Database Design

Module: NoSQL; Big Data Systems

MapReduce Overview

Instructor: Amol Deshpande
amol@umd.edu

Big Data; Storage Systems

- Book Chapters
  - 10.3 **(7TH EDITION)**

- Key topics:
  - Why MapReduce and History
  - Word Count using MapReduce

The MapReduce Paradigm

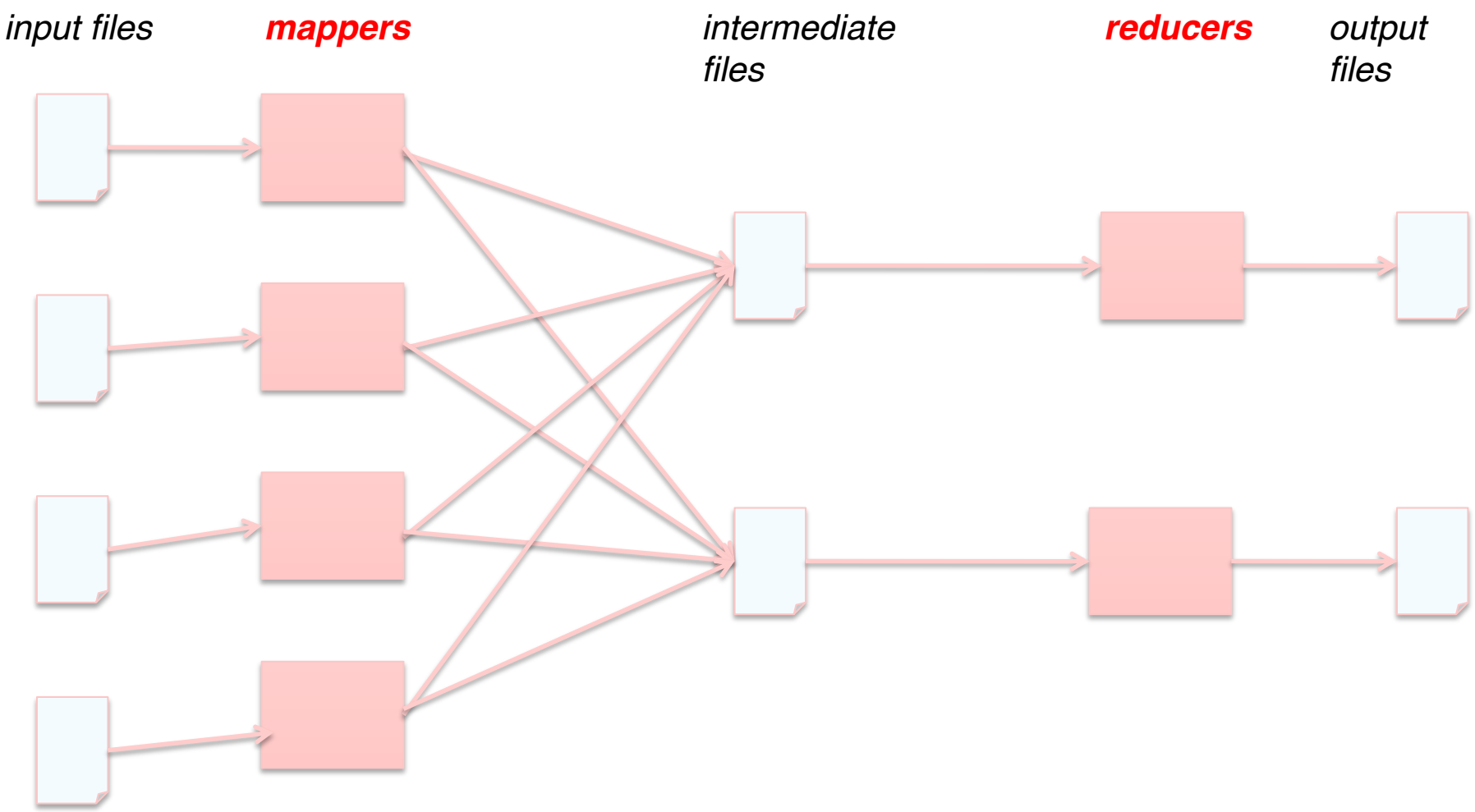
- Platform for reliable, scalable parallel computing
- Abstracts issues of distributed and parallel environment from programmer
  - Programmer provides core logic (via map() and reduce() functions)
  - System takes care of parallelization of computation, coordination, etc.
- Paradigm dates back many decades
  - But very large scale implementations running on clusters with 10^3 to 10^4 machines are more recent
  - Google Map Reduce, Hadoop, ..
- Data storage/access typically done using distributed file systems or key-value stores

MapReduce Framework

- Provides a fairly restricted, but still powerful abstraction for programming

- Programmers write a pipeline of functions, called *map* or *reduce*
  - **map programs**
    - inputs: a list of “records” (record defined arbitrarily – could be images, genomes etc…)
    - output: for each record, produce a set of “(key, value)” pairs
  - **reduce programs**
    - input: a list of “(key, {values})” grouped together from the mapper
    - output: whatever
  - Both can do arbitrary computations on the input data as long as the basic structure is followed

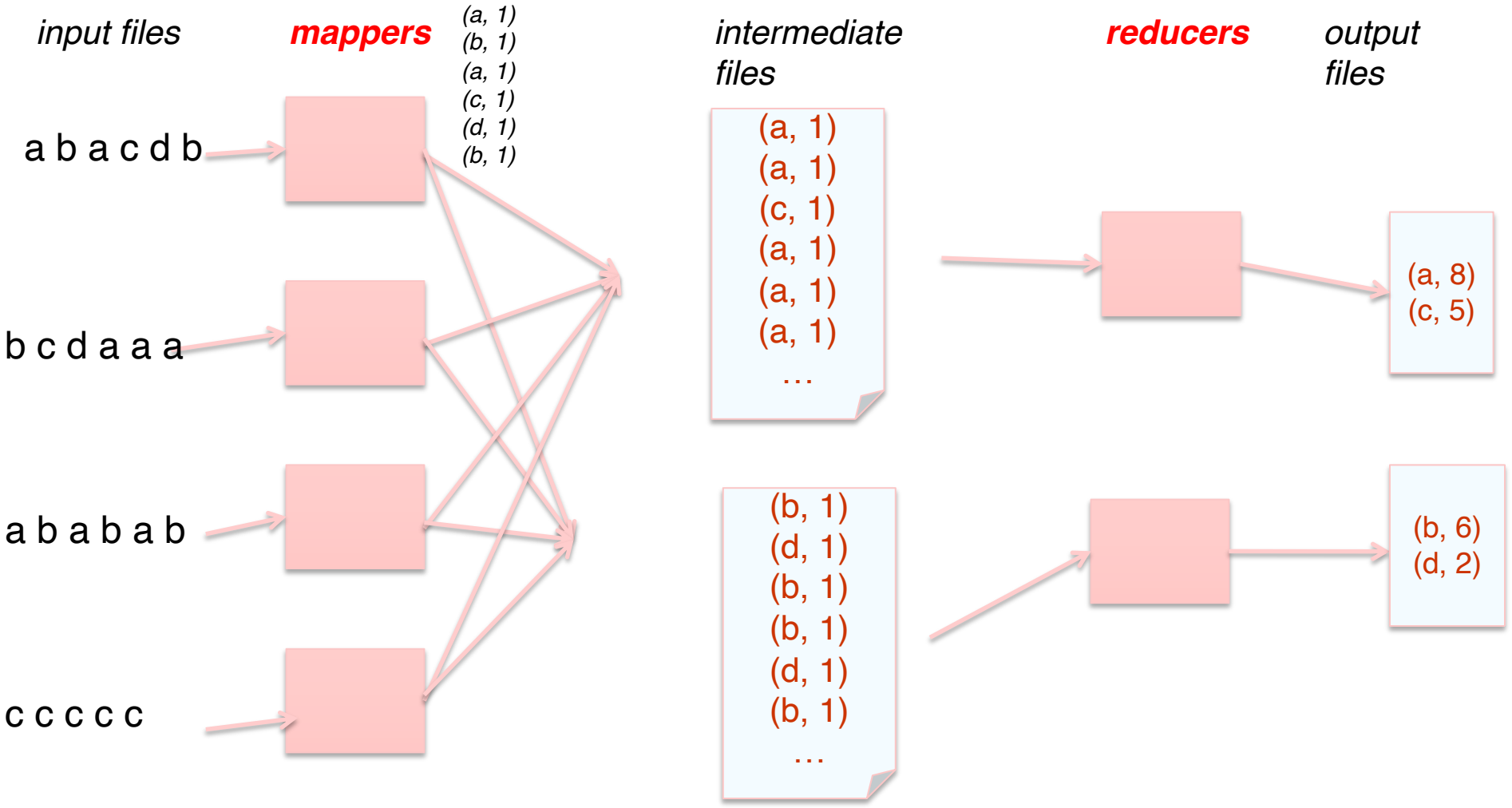
MapReduce Framework



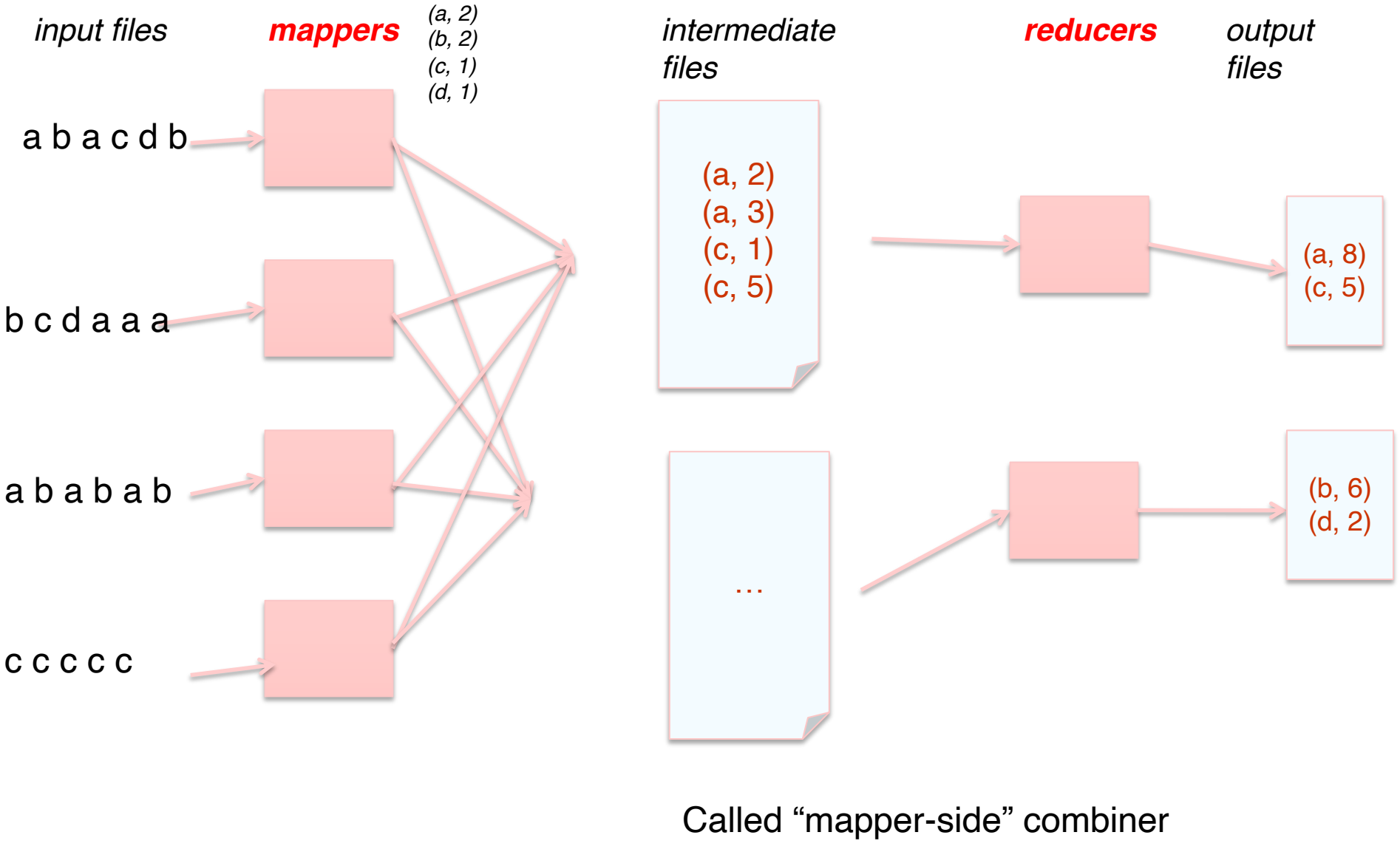
Word Count Example

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

MapReduce Framework: Word Count



More Efficient Word Count



Called “mapper-side” combiner

Hadoop MapReduce

- Google pioneered original map-reduce implementation
  - For building web indexes, text analysis, PageRank, etc.
- Hadoop -- widely used open source implementation in Java
- Huge ecosystem built around Hadoop now, including HDFS, consistency mechanisms, connectors to different systems (e.g., key-value stores, databases), etc.
- Apache Spark a newer implementation of Map-Reduce
  - More user-friendly syntax
  - Significantly faster because of in-memory processing
  - SQL-like in many ways (“DataFrames”)

CMSC424: Database Design

Module: NoSQL; Big Data Systems

MongoDB

Instructor: Amol Deshpande
amol@umd.edu

MongoDB: History 1

- A prototypical NoSQL database
- Short for hu**mongo**us
- First version in 2009!
- Still very popular
  - IPO in 2017
  - Now worth >7B in market capital (as of 2020)

Slides adapted from CS186 Slides by:
Alvin Cheung
Aditya Parameswaran

MongoDB: History 2

- Internet & social media boom led to a demand for
  - Rapid data model evolution: "a move fast and break things" mentality to system dev
    - E.g., adding a new attrib to a Facebook profile
    - Contrary to DBMS wisdom of declaring schema upfront and changing rarely (costly!)
  - Rapid txn support, even at the cost of losing some updates or non-atomicity
    - Contrary to DBMS wisdom of ACID, esp. with distribution/2PC (costly!)

- Early version centered around storing and querying json documents quickly
- Made several tradeoffs for speed
  - No joins ➔ now support left outer joins
  - Limited query opt ➔ still limited, but many improvements
  - No txn support apart from atomic writes to json docs ➔ limited support for multi-doc txns
  - No checks/schema validation ➔ now support json schema validation (rarely used!)

MongoDB: History 3

https://www.mongodb.com/blog/post/what-about-durability

- Most egregious: no durability or write ahead logging!

Excuse 1: Durability is overrated

Excuse 2: It's hard to implement

We get lots of questions about why MongoDB doesn't have full single server durability, and there are many people that think this is a major problem. We wanted to shed some light on why we haven't done single server durability, what our suggestions are, and our future plans.

To start, there are some very practical reasons why we think single server durability is overvalued. First, there are many scenarios in which that server loses all its data no matter what. If there is water damage, fire, some hardware problems, etc… no matter how durable the software is, data can be lost. Yes - there are ways to mitigate some of these, but those add another layer of complexity, that has to be tested, proofed, and adds more variables which can fail.

In the real world, traditional durability often isn't even done correctly. If you are using a DBMS that uses a transaction log for durability, you either have to turn off hardware buffering or have a battery backed RAID controller. Without hardware buffering, transaction logs are very slow. Battery backed raid controllers will work well, but you have to really have one. With the move towards the cloud and outsourced hosting, custom hardware is not always an option.

Sure enough, this was fixed later (four years after the first version!)

MongoDB: History 4

Bottomline:

MongoDB has now evolved into a mature "DBMS" with some different design decisions, and relearning many of the canonical DBMS lessons

We'll focus on two primary design decisions:

- The data model
- The query language

Will discuss these two to start with, then some of the architectural issues

MongoDB Data Model

MongoDB	DBMS
Database	Database
Collection	Relation
Document	Row/Record
Field	Column

Document = {…, field: value, …}

Where value can be:

- Atomic
- A document
- An array of atomic values
- An array of documents

{ qty : 1, status : "D", size : {h : 14, w : 21}, tags : ["a", "b"] },

Can also mix and match, e.g., array of atomics and documents, or array of arrays
[Same as the JSON data model]

Internally stored as BSON = Binary JSON

- Client libraries can directly operate on this natively

MongoDB Data Model 2

MongoDB	DBMS
Database	Database
Collection	Relation
Document	Row/Record
Field	Column

Document = {…, field: value, …}

Can use JSON schema validation

- Some integrity checks, field typing and ensuring the presence of certain fields
- Rarely used, and we'll skip for our discussion

Special field in each document: _id

- Primary key
- Will also be indexed by default
- If it is not present during ingest, it will be added
- Will be first attribute of each doc.
- This field requires special treatment during projections as we will see later

MongoDB Query Language (MQL)

- Input = collections, output = collections
  - Very similar to Spark

- Three main types of queries in the query language
  - Retrieval: Restricted SELECT-WHERE-ORDER BY-LIMIT type queries
  - Aggregation: A bit of a misnomer; a general pipeline of operators
    - Can capture Retrieval as a special case
    - But worth understanding Retrieval queries first…
  - Updates

- All queries are invoked as
  - db.collection.operation1(…).operation2(…)…
    - collection: name of collection
  - Unlike SQL which lists many tables in a FROM clause, MQL is centered around manipulating a single collection (like Spark)

Syntax somewhat different when called from within Python3 (using pymongo)

Some MQL Principles : Dot (.) Notation

- "." is used to drill deeper into nested docs/arrays
- Recall that a value could be atomic, a nested document, an array of atomics, or an array of nested documents

- Examples:
  - "instock.qty" ➔ qty field within the instock field
  - "instock.1" ➔ second element within the instock array
    - Element could be an atomic value or a nested document
  - "instock.1.qty" ➔ qty field within the second document within the instock array

- Note: such dot expressions need to be in quotes

Some MQL Principles : Dollar ($) Notation

- $ indicates that the string is a special keyword
  - E.g., $gt, $lte, $add, $elemMatch, …
- Used as the "field" part of a "field : value" expression
- So if it is a binary operator, it is *usually* done as:
  - {LOperand : { $keyword : ROperand}}
  - e.g., {qty : {$gt : 30}}
- Alternative: arrays
  - {$keyword : [argument list]}
  - e.g., {$add : [1, 2]}

- Exception: $fieldName, used to refer to a previously defined field on the value side
  - Purpose: disambiguation
  - Only relevant for aggregation pipelines
  - Let's not worry about this for now.

Retrieval Queries Template

db.collection.**find**(<*predicate*>, optional <*projection*>)

returns documents that match <*predicate*>
keep fields as specified in <*projection*>
both <*predicate*> and <*projection*> expressed as documents
in fact, most things are documents!

db.inventory.find({ })
returns all documents

```
> db.inventory.find()
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d993"), "item" : "journal", "qty" : 25, "size" : { "h" : 14, "w" : 21, "uom" : "cm" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d994"), "item" : "notebook", "qty" : 50, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d995"), "item" : "paper", "qty" : 100, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d996"), "item" : "planner", "qty" : 75, "size" : { "h" : 22.85, "w" : 30, "uom" : "cm" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d997"), "item" : "postcard", "qty" : 45, "size" : { "h" : 10, "w" : 15.25, "uom" : "cm" }, "status" : "A" }
```

Syntax somewhat different when called from within Python3 (using pymongo)

Retrieval Queries: Basic Queries

```
> db.inventory.find()
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d993"), "item" : "journal", "qty" : 25, "size" : { "h" : 14, "w" : 21, "uom" : "cm" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d994"), "item" : "notebook", "qty" : 50, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d995"), "item" : "paper", "qty" : 100, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d996"), "item" : "planner", "qty" : 75, "size" : { "h" : 22.85, "w" : 30, "uom" : "cm" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d997"), "item" : "postcard", "qty" : 45, "size" : { "h" : 10, "w" : 15.25, "uom" : "cm" }, "status" : "A" }
```

db.collection.find(<predicate>, optional <projection>)

- find({ status : "D"})
  - all documents with status D ➔ paper, planner
- find ({ qty : {$gte : 50} })
  - all documents with qty >= 50 ➔ notebook, paper, planner
- find ({ status : "D", qty : {$gte : 50} })
  - all documents that satisfy both ➔ paper, planner
- find({ $or: [{ status : "D" }, { qty : { $lt : 30 } }] })
  - all documents that satisfy either ➔ journal, paper, planner

```
> db.inventory.find( { $or: [ { status: "D" }, { qty: { $lt: 30 } } ] } )
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d993"), "item" : "journal", "qty" : 25, "size" : { "h" : 14, "w" : 21, "uom" : "cm" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d995"), "item" : "paper", "qty" : 100, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d996"), "item" : "planner", "qty" : 75, "size" : { "h" : 22.85, "w" : 30, "uom" : "cm" }, "status" : "D" }
```

Syntax somewhat different when called from within Python3 (using pymongo)

Retrieval Queries: Nested Documents

```
> db.inventory.find()
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d993"), "item" : "journal", "qty" : 25, "size" : { "h" : 14, "w" : 21, "uom" : "cm" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d994"), "item" : "notebook", "qty" : 50, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "A" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d995"), "item" : "paper", "qty" : 100, "size" : { "h" : 8.5, "w" : 11, "uom" : "in" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d996"), "item" : "planner", "qty" : 75, "size" : { "h" : 22.85, "w" : 30, "uom" : "cm" }, "status" : "D" }
{ "_id" : ObjectId("5fb55fad75b4eb537dc3d997"), "item" : "postcard", "qty" : 45, "size" : { "h" : 10, "w" : 15.25, "uom" : "cm" }, "status" : "A" }
```

db.collection.find(<predicate>, optional <projection>)

- find({ size: { h: 14, w: 21, uom: "cm" } })
  - exact match of nested document, including ordering of fields! ➔ journal
- find ({ "size.uom" : "cm", "size.h" : {$gt : 14 } })
  - querying a nested field ➔ planner
  - Note: when using . notation for sub-fields, expression must be in quotes
  - Also note: binary operator handled via a nested document

Syntax somewhat different when called from within Python3 (using pymongo)

Retrieval Queries: Arrays

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Slightly different example dataset for Arrays and Arrays of Document Examples
db.collection.find(<predicate>, optional <projection>)

- find({ tags: ["red", "blank"] })
  - Exact match of array ➔ notebook
- find({ tags: "red" })
  - If one of the elements matches red ➔ journal, notebook, paper, planner
- find({ tags: "red", tags: "plain" })
  - If one matches red, one matches plain ➔ paper
- find({ dim: { $gt: 15, $lt: 20 } })
  - If one element is >15 and another is <20 ➔ journal, notebook, paper, postcard
- find({ dim: {$elemMatch: { $gt: 15, $lt: 20 } } })
  - If a single element is >15 and <20 ➔ postcard
- find({ "dim.1": { $gt: 25 } })
  - If second item > 25 ➔ planner
  - Notice again that we use quotes to when using . notation

Syntax somewhat different when called from within Python3 (using pymongo)

Retrieval Queries: Arrays of Documents

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

db.collection.find(<predicate>, optional <projection>)

- find({ instock: { loc: "A", qty: 5 } })
  - Exact match of document [like nested doc/atomic array case] ➔ journal
- find({ "instock.qty": { $gte : 20 } })
  - One nested doc has >= 20 ➔ paper, planner, postcard
- find({ "instock.0.qty": { $gte : 20 } })
  - First nested doc has >= 20 ➔ paper, planner
- find({ "instock": { $elemMatch: { qty: { $gt: 10, $lte: 20 } } } })
  - One doc has 20 >= qty >10➔ paper, journal, postcard
- find({ "instock.qty": { $gt: 10, $lte: 20 } })
  - One doc has 20 >= qty, another has qty>10 ➔ paper, journal, postcard, planner

Syntax somewhat different when called from within Python3 (using pymongo)

Retrieval Queries Template: Projection

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

db.collection.find(<predicate>, optional <projection>)

- Use 1s to indicate fields that you want
  - Exception: _id is always present unless explicitly excluded
- OR Use 0s to indicate fields you don't want
- Mixing 0s and 1s is not allowed for non _id fields

- find({ }, {item: 1})

```
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal" }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook" }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper" }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner" }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard" }
```

- find({ }, {item: 1, _id : 0})

```
{ "item" : "journal" }
{ "item" : "notebook" }
{ "item" : "paper" }
{ "item" : "planner" }
{ "item" : "postcard" }
```

- find({},{item : 1, tags: 0, _id : 0})

```
Error: error: {
"ok" : 0,
"errmsg" : "Cannot do exclusion on field tags in inclusion projection",
"code" : 31254,
"codeName" : "Location31254" }
```

- find({},{item : 1, "instock.loc": 1, _id : 0})

```
{ "item" : "journal", "instock" : [ { "loc" : "A" }, { "loc" : "C" } ] }
{ "item" : "notebook", "instock" : [ { "loc" : "C" } ] }
{ "item" : "paper", "instock" : [ { "loc" : "A" }, { "loc" : "B" } ] }
{ "item" : "planner", "instock" : [ { "loc" : "A" }, { "loc" : "B" } ] }
{ "item" : "postcard", "instock" : [ { "loc" : "B" }, { "loc" : "C" } ] }
```

Syntax somewhat different when called from within Python3 (using pymongo)

Retrieval Queries : Addendum

```
db.inventory.find()
"_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
"_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
"_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
"_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
"_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Two additional operations that are useful for retrieval:

- Limit (k) like LIMIT in SQL
  - e.g., db.inventory.find({ }).limit(1)
- Sort ({ }) like ORDER BY in SQL
  - List of fields, -1 indicates decreasing 1 indicates ascending
  - e.g., db.inventory.find({ }, {_id : 0, instock : 0}).sort({ "dim.0": -1, item: 1 })

```
{ "item" : "planner", "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "item" : "journal", "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "item" : "notebook", "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "item" : "paper", "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "item" : "postcard", "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Syntax somewhat different when called from within Python3 (using pymongo)

Retrieval Queries: Summary

find() = SELECT <projection>
FROM Collection
WHERE <predicate>
limit() = LIMIT
sort() = ORDER BY

```
db.inventory.find(
        { tags : red },
        {_id : 0, instock : 0} )
.sort ( { "dim.0": -1, item: 1 } )
.limit (2)
```

FROM
WHERE
SELECT
ORDER BY
LIMIT

Syntax somewhat different when called from within Python3 (using pymongo)

What did we not cover?

- The use of regexes for matching
- $all : all entries in an array satisfy a condition
- $in : checking if a value is present in an array of atomic values
- The presence or absence of fields
  - Can use special “null” values
  - {field : null} checks if a field is null or missing
  - $exists : checking the presence/absence of a field

Syntax somewhat different when called from within Python3 (using pymongo)

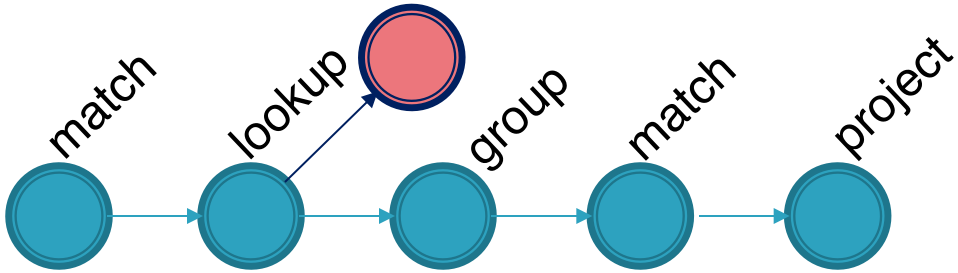
MongoDB Query Language (MQL)

- Input = collections, output = collections
  - Very similar to Spark
- Three main types of queries in the query language
  - Retrieval: Restricted SELECT-WHERE-ORDER BY-LIMIT type queries
  - Aggregation: A bit of a misnomer; a general pipeline of operators
    - Can capture Retrieval as a special case
    - But worth understanding Retrieval queries first…
  - Updates
- All queries are invoked as
  - db.collection.operation1(…).operation2(…)…
    - collection: name of collection
  - Unlike SQL which lists many tables in a FROM clause, MQL is centered around manipulating a single collection (like Spark)

Syntax somewhat different when called from within Python3 (using pymongo)

Aggregation Pipelines

- Composed of a linear *pipeline* of *stages*
- Each stage corresponds to one of:
  - match // first arg of find ( )
  - project // second arg of find ( ) but more expressiveness
  - sort/limit // same as retrieval
  - group
  - unwind
  - lookup
  - … lots more!!
- Each stage manipulates the existing collection in some way



- Syntax:

```
db.collection.aggregate ( [
 { $stage1Op: { } },
 { $stage2Op: { } },
  …
 { $stageNOp: { } }
] )
```

Next Set of Examples

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
{ "_id" : "01020", "city" : "CHICOPEE", "loc" : [ -72.576142, 42.176443 ], "pop" : 31495, "state" : "MA" }
{ "_id" : "01028", "city" : "EAST LONGMEADOW", "loc" : [ -72.505565, 42.067203 ], "pop" : 13367, "state" : "MA" }
{ "_id" : "01030", "city" : "FEEDING HILLS", "loc" : [ -72.675077, 42.07182 ], "pop" : 11985, "state" : "MA" }
{ "_id" : "01032", "city" : "GOSHEN", "loc" : [ -72.844092, 42.466234 ], "pop" : 122, "state" : "MA" }
{ "_id" : "01012", "city" : "CHESTERFIELD", "loc" : [ -72.833309, 42.38167 ], "pop" : 177, "state" : "MA" }
{ "_id" : "01010", "city" : "BRIMFIELD", "loc" : [ -72.188455, 42.116543 ], "pop" : 3706, "state" : "MA" }
{ "_id" : "01034", "city" : "TOLLAND", "loc" : [ -72.908793, 42.070234 ], "pop" : 1652, "state" : "MA" }
{ "_id" : "01035", "city" : "HADLEY", "loc" : [ -72.571499, 42.36062 ], "pop" : 4231, "state" : "MA" }
{ "_id" : "01001", "city" : "AGAWAM", "loc" : [ -72.622739, 42.070206 ], "pop" : 15338, "state" : "MA" }
{ "_id" : "01040", "city" : "HOLYOKE", "loc" : [ -72.626193, 42.202007 ], "pop" : 43704, "state" : "MA" }
{ "_id" : "01008", "city" : "BLANDFORD", "loc" : [ -72.936114, 42.182949 ], "pop" : 1240, "state" : "MA" }
{ "_id" : "01050", "city" : "HUNTINGTON", "loc" : [ -72.873341, 42.265301 ], "pop" : 2084, "state" : "MA" }
{ "_id" : "01054", "city" : "LEVERETT", "loc" : [ -72.499334, 42.46823 ], "pop" : 1748, "state" : "MA" }
```

One document per zipcode: 29353 zipcodes

Syntax somewhat different when called from within Python3 (using pymongo)

Grouping (with match/sort) Simple Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

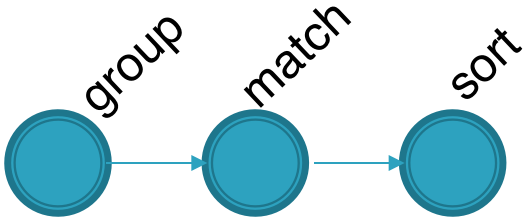
Find states with population > 15M, sort by decending order

```
db.zips.aggregate( [
{ $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
{ $match: { totalPop: { $gte: 15000000 } } },
{ $sort : { totalPop : -1 } }
 ] )
```

GROUP BY

AGGS.

match after group = HAVING




```
{ "_id" : "CA", "totalPop" : 29754890 }
{ "_id" : "NY", "totalPop" : 17990402 }
{ "_id" : "TX", "totalPop" : 16984601 }
…
```

Q: what would the SQL query for this be?

SELECT state AS id, **SUM**(pop) **AS** totalPop
FROM zips
GROUP BY state
HAVING totalPop >= 15000000
ORDER BY totalPop **DESCENDING**

Syntax somewhat different when called from within Python3 (using pymongo)

Grouping Syntax

```
$group : {
        _id: <expression>, // Group By Expression
        <field1>: { <aggfunc1> : <expression1> },
        ... }
```

Returns one document per unique group, indexed by _id

Agg.func. can be standard ops like $sum, $avg, $max

Also MQL specific ones:

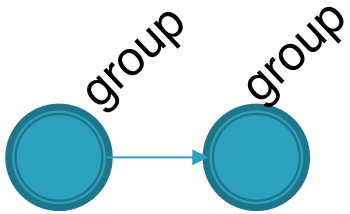
- **$first** : return the first expression value per group
  - makes sense only if docs are in a specific order [usually done after sort]
- **$push** : create an array of expression values per group
  - didn't make sense in a relational context because values are atomic
- **$addToSet** : like $push, but eliminates duplicates

Multiple Attrib. Grouping Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

Group by 2 attribs, giving nested id

```
aggregate( [
{ $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
{ $group: { _id: "$_id.state", avgCityPop: { $avg: "$pop" } } }
] )
```



Group by previously def. id.state

Notice use of $ to refer to previously defined fields

Q: Guesses on what this might be doing?
A: Find average city population per state

```
{ "_id" : "GA", "avgCityPop" : 11547.62210338681 }
{ "_id" : "WI", "avgCityPop" : 7323.00748502994 }
{ "_id" : "FL", "avgCityPop" : 27400.958963282937 }
{ "_id" : "OR", "avgCityPop" : 8262.561046511628 }
{ "_id" : "SD", "avgCityPop" : 1839.6746031746031 }
{ "_id" : "NM", "avgCityPop" : 5872.360465116279 }
{ "_id" : "MD", "avgCityPop" : 12615.775725593667 }
...
```

Syntax somewhat different when called from within Python3 (using pymongo)

Multiple Agg. Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

Find, for every state, the biggest city and its population

aggregate([

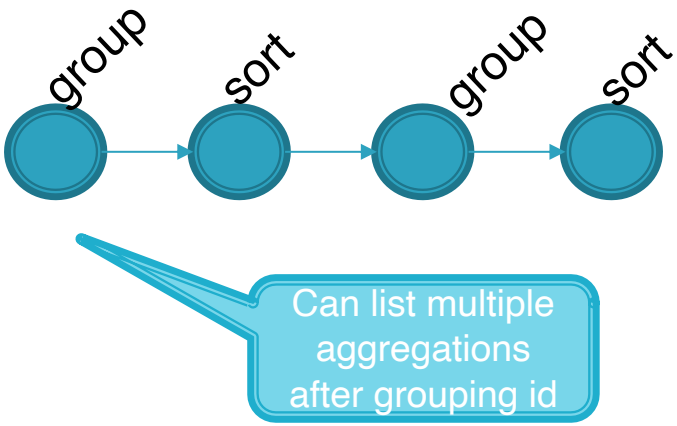
{ $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },

{ $sort: { pop: -1 } },

{ $group: { _id : "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } } },

{ $sort : {bigPop : -1} }

])



Approach:

- Group by pair of city and state, and compute population per city
- Order by population descending
- Group by state, and find first city and population per group (i.e., the highest population city)
- Order by population descending

{ ”_id" : "IL", "bigCity" : "CHICAGO", "bigPop" : 2452177 }

{ "_id" : "NY", "bigCity" : "BROOKLYN", "bigPop" : 2300504 }

{ "_id" : "CA", "bigCity" : "LOS ANGELES", "bigPop" : 2102295 }

{ "_id" : "TX", "bigCity" : "HOUSTON", "bigPop" : 2095918 }

{ "_id" : "PA", "bigCity" : "PHILADELPHIA", "bigPop" : 1610956 }

{ "_id" : "MI", "bigCity" : "DETROIT", "bigPop" : 963243 }

...

Syntax somewhat different when called from within Python3 (using pymongo)

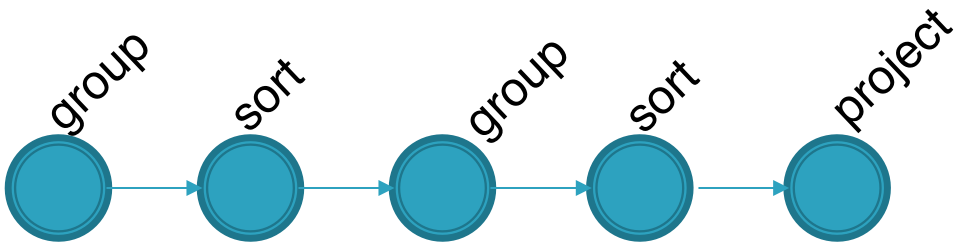
Multiple Agg. with Vanilla Projection Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

If we only want to keep the state and city …

```
aggregate( [
{ $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
{ $sort: { pop: -1 } },
{ $group: { _id : "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } } },
{ $sort : {bigPop : -1} }
{ $project : {bigPop : 0} }
] )
```

```
{ "_id" : "IL", "bigCity" : "CHICAGO" }
{ "_id" : "NY", "bigCity" : "BROOKLYN" }
{ "_id" : "CA", "bigCity" : "LOS ANGELES" }
{ "_id" : "TX", "bigCity" : "HOUSTON" }
{ "_id" : "PA", "bigCity" : "PHILADELPHIA" }
…
```



Syntax somewhat different when called from within Python3 (using pymongo)

Multiple Agg. with Adv. Projection Example

```
> db.zips.find()
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "pop" : 1688, "state" : "MA" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA" }
```

If we wanted to nest the name of the city and population into a nested doc

Can construct new nested documents in output, unlike vanilla projection

```
aggregate( [
{ $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
{ $sort: { pop: -1 } },
{ $group: { _id : "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } } },
{ $sort : {bigPop : -1} },
{ $project : { _id : 0, state : "$_id", bigCityDeets: { name: "$bigCity", pop: "$bigPop" } } }
] )
```

```
{ "state" : "IL", "bigCityDeets" : { "name" : "CHICAGO", "pop" : 2452177 } }
{ "state" : "NY", "bigCityDeets" : { "name" : "BROOKLYN", "pop" : 2300504 } }
{ "state" : "CA", "bigCityDeets" : { "name" : "LOS ANGELES", "pop" : 2102295 } }
{ "state" : "TX", "bigCityDeets" : { "name" : "HOUSTON", "pop" : 2095918 } }
{ "state" : "PA", "bigCityDeets" : { "name" : "PHILADELPHIA", "pop" : 1610956 } }
…
```

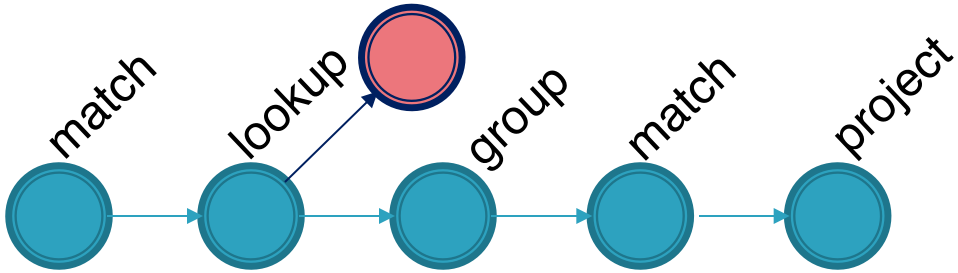
Syntax somewhat different when called from within Python3 (using pymongo)

Advanced Projection vs. Vanilla Projection

- In addition to excluding/including fields like in projection during retrieval (find), projection in the aggregation pipeline allows you to:
  - Rename fields
  - Redefine new fields using complex expressions on old fields
  - Reorganize fields into nestings or unnestings
  - Reorganize fields into arrays or break down arrays
- Try them at home!

Aggregation Pipelines

- Composed of a linear *pipeline* of *stages*
- Each stage corresponds to one of:
  - match // first arg of find ( )
  - project // second arg of find ( ) but more expressiveness
  - sort/limit // same
  - group
  - **unwind**
  - **lookup**
  - … lots more!!
- Each stage manipulates the existing collection in some way



- Syntax:

```
db.collection.aggregate ( [
 { $stage1Op: { } },
 { $stage2Op: { } },
  …
 { $stageNOp: { } }
] )
```

Syntax somewhat different when called from within Python3 (using pymongo)

Unwinding Arrays

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Unwind expands an array by constructing documents one per element of the array

Somewhat like flatMap in Spark

Going back to our old example with an array of tags

Notice no relational analog here: no arrays so no unwinding

[in fact, some RDBMSs do support arrays, but not in the rel. model]

```
aggregate( [
        { $unwind : "$tags" },
        { $project : {_id : 0, instock: 0}}
] )
```

```
{ "item" : "journal", "tags" : "blank", "dim" : [ 14, 21 ] }
{ "item" : "journal", "tags" : "red", "dim" : [ 14, 21 ] }
{ "item" : "notebook", "tags" : "red", "dim" : [ 14, 21 ] }
{ "item" : "notebook", "tags" : "blank", "dim" : [ 14, 21 ] }
{ "item" : "paper", "tags" : "red", "dim" : [ 14, 21 ] }
{ "item" : "paper", "tags" : "blank", "dim" : [ 14, 21 ] }
{ "item" : "paper", "tags" : "plain", "dim" : [ 14, 21 ] }
{ "item" : "planner", "tags" : "blank", "dim" : [ 22.85, 30 ] }
{ "item" : "planner", "tags" : "red", "dim" : [ 22.85, 30 ] }
{ "item" : "postcard", "tags" : "blue", "dim" : [ 10, 15.25 ] }
```

Syntax somewhat different when called from within Python3 (using pymongo)

Unwind: A Common Template

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Q: Imagine if we want to find sum of qtys across items. How would we do this?

A common recipe in MQL queries is to *unwind* and then *group by*

```
aggregate( [
        { $unwind : "$instock" },
        { $group : {_id : "$item", totalqty : {$sum : "$instock.qty"}}}
] )
```

```
{ "_id" : "notebook", "totalqty" : 5 }
{ "_id" : "postcard", "totalqty" : 50 }
{ "_id" : "journal", "totalqty" : 20 }
{ "_id" : "planner", "totalqty" : 45 }
{ "_id" : "paper", "totalqty" : 75 }
```

Syntax somewhat different when called from within Python3 (using pymongo)

Looking Up Other Collections

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

```
{ $lookup: {
        from: <collection to join>,
        localField: <referencing field>,
        foreignField: <referenced field>,
        as: <output array field>
    } }
```

Conceptually, for each document

- find documents in other coll that join (equijoin)
  - local field must match foreign field
- place each of them in an array

Thus, a left outer equi-join, with the join results stored in an array

Straightforward, but kinda gross. Let's see…

Say, for each item, I want to find other items located in the same location = self-join

```
db.inventory.aggregate( [
    { $lookup : {from : "inventory", localField: "instock.loc",
    foreignField: "instock.loc", as:"otheritems"}},
    { $project : {_id : 0, tags : 0, dim : 0}}
] )
```

```
{ "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty"
: 15 } ], "otheritems" : [
    { "_id" : ObjectId("5fb6f9605f0594e0227d3c24"), "item" : "journal",
    "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ],
    "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] },
    { "_id" : ObjectId("5fb6f9605f0594e0227d3c25"), "item" :
    "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red",
    "blank" ], "dim" : [ 14, 21 ] },
    { "_id" : ObjectId("5fb6f9605f0594e0227d3c26"), "item" : "paper",
    "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ],
    "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] },
    …
    ] }
```

And many other records!

Syntax somewhat different when called from within Python3 (using pymongo)

Lookup… after some more projection

```
> db.inventory.find()
{ "_id" : ObjectId("5fb59ab9f50b800678c0e196"), "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e197"), "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e198"), "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e199"), "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "_id" : ObjectId("5fb59ab9f50b800678c0e19a"), "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

db.inventory.aggregate([

{ $lookup : {from:"inventory", localField:"instock.loc", foreignField:"instock.loc", as:"otheritems"}},

{$project : {_id : 0, tags :0, dim :0, **"otheritems._id":0, "otheritems.tags":0, "otheritems.dim":0, "otheritems.instock.qty":0**}}])

{ "item" : "journal", "instock" : [{ "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 }], "otheritems" : [

{ "item" : "journal", "instock" : [{ "loc" : "A" }, { "loc" : "C" }] },

{ "item" : "notebook", "instock" : [{ "loc" : "C" }] },

{ "item" : "paper", "instock" : [{ "loc" : "A" }, { "loc" : "B" }] },

{ "item" : "planner", "instock" : [{ "loc" : "A" }, { "loc" : "B" }] },

{ "item" : "postcard", "instock" : [{ "loc" : "B" }, { "loc" : "C" }] }] }

{ "item" : "notebook", "instock" : [{ "loc" : "C", "qty" : 5 }], "otheritems" : [

{ "item" : "journal", "instock" : [{ "loc" : "A" }, { "loc" : "C" }] },

{ "item" : "notebook", "instock" : [{ "loc" : "C" }] },

{ "item" : "postcard", "instock" : [{ "loc" : "B" }, { "loc" : "C" }] }] }

…

Syntax somewhat different when called from within Python3 (using pymongo)

Some Rules of Thumb when Writing Queries

- $project is helpful if you want to construct or deconstruct nestings (in addition to removing fields or creating new ones)
- $group is helpful to construct arrays (using $push or $addToSet)
- $unwind is helpful for unwinding arrays
- $lookup is your only hope for joins. Be prepared for a mess. Lots of $project needed

MongoDB Query Language (MQL)

- Input = collections, output = collections
  - Very similar to Spark
- Three main types of queries in the query language
  - Retrieval: Restricted SELECT-WHERE-ORDER BY-LIMIT type queries
  - Aggregation: A bit of a misnomer; a general pipeline of operators
    - Can capture Retrieval as a special case
    - But worth understanding Retrieval queries first…
  - Updates
- All queries are invoked as
  - db.collection.operation1(…).operation2(…)…
    - collection: name of collection
  - Unlike SQL which lists many tables in a FROM clause, MQL is centered around manipulating a single collection (like Spark)

Syntax somewhat different when called from within Python3 (using pymongo)

Update Queries: InsertMany

[Insert/Delete/Update] [One/Many]

- Many is more general, so we'll discuss that instead

```
db.inventory.insertMany( [
{ item: "journal", instock: [ { loc: "A", qty: 5 }, { loc: "C", qty: 15 } ], tags: ["blank", "red"], dim: [ 14, 21 ] },
{ item: "notebook", instock: [ { loc: "C", qty: 5 } ], tags: ["red", "blank"] , dim: [ 14, 21 ]},
{ item: "paper", instock: [ { loc: "A", qty: 60 }, { loc: "B", qty: 15 } ], tags: ["red", "blank", "plain"] , dim: [ 14, 21 ]},
{ item: "planner", instock: [ { loc: "A", qty: 40 }, { loc: "B", qty: 5 } ], tags: ["blank", "red"], dim: [ 22.85, 30 ] },
{ item: "postcard", instock: [ {loc: "B", qty: 15 }, { loc: "C", qty: 35 } ], tags: ["blue"] , dim: [ 10, 15.25 ] }
] );
```

Several actions will be taken as part of this insert:

- Will create inventory collection if absent [No schema specification/DDL needed!]
- Will add the _id attrib to each document added (since it isn't there)
- _id will be the first field for each document by default

Update Queries: UpdateMany

```
> db.inventory.find({}, {_id:0})
{ "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Syntax: updateMany ({<condition>}, {<change>})

```
db.inventory.updateMany (
      {"dim.0": { $lt: 15 } },
      { $set: { "dim.0": 15, status: "InvalidWidth"} }
) // if any width <15, set it to 15 and set status to InvalidWidth.
```

```
> db.inventory.find({}, {_id:0})
{ "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 15, 21 ], "status" : "InvalidWidth" }
{ "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 15, 21 ], "status" : "InvalidWidth" }
{ "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 15, 21 ], "status" : "InvalidWidth" }
{ "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 15, 15.25 ], "status" : "InvalidWidth" }
```

Analogous to: UPDATE R SET <change> WHERE <condition>

Update Queries: UpdateMany 2

```
> db.inventory.find({}, {_id:0})
{ "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 14, 21 ] }
{ "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 14, 21 ] }
{ "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 14, 21 ] }
{ "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 10, 15.25 ] }
```

Syntax: updateMany ({<condition>}, {<change>})

```
db.inventory.updateMany (
{"dim.0": { $lt: 15 } },
{ $inc: { "dim.0":  5},
  $set: {status: "InvalidWidth"} } )
```

// if any width <15, increment by 5 and set status to InvalidWidth.

```
> db.inventory.find({}, {_id:0})
{ "item" : "journal", "instock" : [ { "loc" : "A", "qty" : 5 }, { "loc" : "C", "qty" : 15 } ], "tags" : [ "blank", "red" ], "dim" : [ 19, 21 ], "status" : "InvalidWidth" }
{ "item" : "notebook", "instock" : [ { "loc" : "C", "qty" : 5 } ], "tags" : [ "red", "blank" ], "dim" : [ 19, 21 ], "status" : "InvalidWidth" }
{ "item" : "paper", "instock" : [ { "loc" : "A", "qty" : 60 }, { "loc" : "B", "qty" : 15 } ], "tags" : [ "red", "blank", "plain" ], "dim" : [ 19, 21 ], "status" : "InvalidWidth" }
{ "item" : "planner", "instock" : [ { "loc" : "A", "qty" : 40 }, { "loc" : "B", "qty" : 5 } ], "tags" : [ "blank", "red" ], "dim" : [ 22.85, 30 ] }
{ "item" : "postcard", "instock" : [ { "loc" : "B", "qty" : 15 }, { "loc" : "C", "qty" : 35 } ], "tags" : [ "blue" ], "dim" : [ 15, 15.25 ], "status" : "InvalidWidth" }
```

Analogous to: UPDATE R SET <change> WHERE <condition

MongoDB Internals

- MongoDB is a distributed NoSQL database
- Collections are partitioned/sharded based on a field [range-based]
  - Each partition stores a subset of documents
- Each partition is replicated to help with failures
  - The replication is done asynchronously
  - Failures of the main partition that haven't been propagated will be lost
- Limited heuristic-based query optimization (will discuss later)
- Atomic writes to documents within collections by default. Multi-document txns are discouraged (but now supported).

MongoDB Internals

- Weird constraint: intermediate results of aggregations must not be too large (100MB)
  - Else will end up spilling to disk
  - Not clear if they perform any pipelining across aggregation operators

- Optimization heuristics
  - Will use indexes for $match if early in the pipeline [user can explicitly declare]
  - $match will be merged with other $match if possible
    - Selection fusion
  - $match will be moved early in the pipeline sometimes
    - Selection pushdown
    - But: not done always (e.g., not pushed before $lookup)
  - No cost-based optimization as far as one can tell

MongoDB: Summary

Bottomline:

MongoDB has now evolved into a mature "DBMS" with some different design decisions, and relearning many of the canonical DBMS lessons

MongoDB has a flexible data model and a powerful (if confusing) query language.

Many of the internal design decisions as well as the query & data model can be understood when compared with DBMSs

- DBMSs provide a "gold standard" to compare against.
- In the "wild" you'll encounter many more NoSQL systems, and you'll need to do the same thing that we did here!

CMSC424: Database Design

Module: NoSQL; Big Data Systems

Other Storage Systems; Wrapup

Instructor: Amol Deshpande
amol@umd.edu

Big Data Storage Options

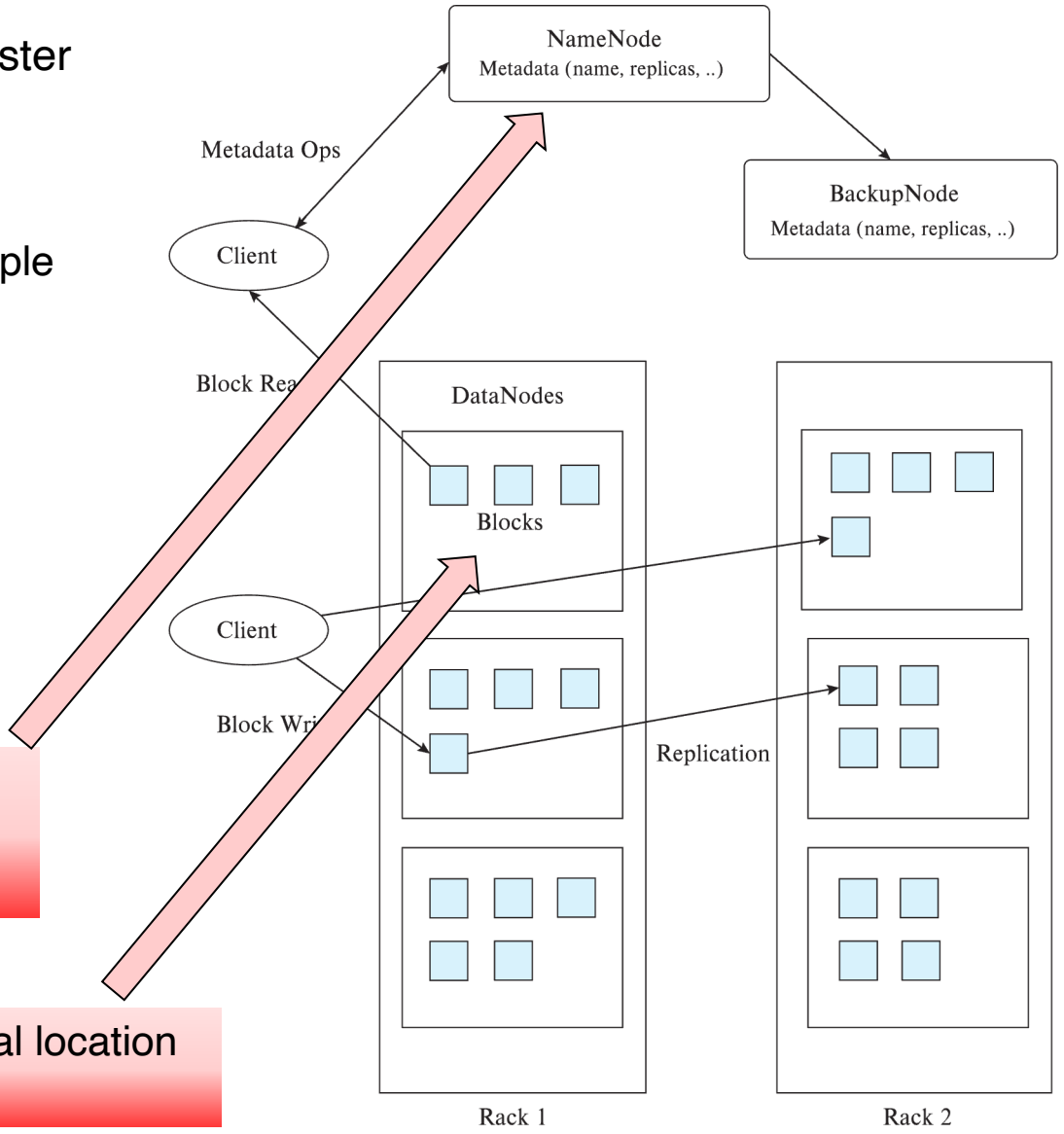
- Parallel or distributed databases
  - Suffer from the issues discussed earlier
- Distributed File Systems
  - Also called object stores
  - A “data lake” is basically a collection of files in a dfs
  - Structured data (relational-like) stored in files (more sophisticated “csv” files)
- Key-value Storage Systems
  - Document stores (MongoDB, etc)
  - Wide column stores (HBase, Cassandra)
  - Graph Stores (Neo4j)
  - And many others…

Distributed File Systems

- A distributed file system stores data across a large collection of machines, but provides single file-system view
- Highly scalable distributed file system for large data-intensive applications.
  - E.g., 10K nodes, 100 million files, 10 PB
- Provides redundant storage of massive amounts of data on cheap and unreliable computers
  - Files are replicated to handle hardware failure
  - Detect failures and recovers from them
- Examples:
  - Google File System (GFS)
  - Hadoop File System (HDFS)

Hadoop File System Architecture

- Single Namespace for entire cluster
- Files are broken up into blocks
  - Typically 64 MB block size
  - Each block replicated on multiple DataNodes
- Client
  - Finds location of blocks from NameNode
  - Accesses data directly from DataNode



- Maps a filename to list of Block IDs
- Maps each Block ID to DataNodes containing a replica of the block

Maps a Block ID to a physical location on disk

Key-Value Storage Systems

- Unlike HDFS, focus here on storing large numbers (billions or even more) of small (KB-MB) sized records
  - **uninterpreted bytes**, with an associated key
    - E.g., Amazon S3, Amazon Dynamo
  - **Wide-table** (can have arbitrarily many attribute names) with associated key
    - Google BigTable, Apache Cassandra, Apache Hbase, Amazon DynamoDB
    - Allows some operations (e.g., filtering) to execute on storage node
  - JSON
    - MongoDB, CouchDB (document model)
- Records **partitioned** across multiple machines
  - Queries are routed by the system to appropriate machine
- Records **replicated** across multiple machines for fault tolerance as well as efficient querying
  - Need to guarantee "consistency" when data is updated
  - **"Distributed Transactions"**

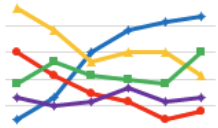
Key-Value Storage Systems

- Key-value stores support
  - ***put***(key, value): used to store values with an associated key,
  - ***get***(key): which retrieves the stored value associated with the specified key
  - ***delete***(key) -- Remove the key and its associated value
- Some support ***range queries*** on key values
- Document stores support richer queries (e.g., MongoDB)
  - Slowly evolving towards the richness of SQL
- Not full database systems (increasingly changing)
  - Have no/limited support for transactional updates
  - Applications must manage query processing on their own
- Not supporting above features makes it easier to build scalable data storage systems, i.e., NoSQL systems

DB-Engines Ranking

The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly.

Read more about the method of calculating the scores.



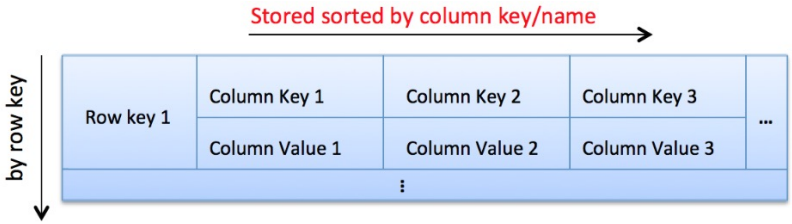
trend chart

360 systems in ranking, November 2020

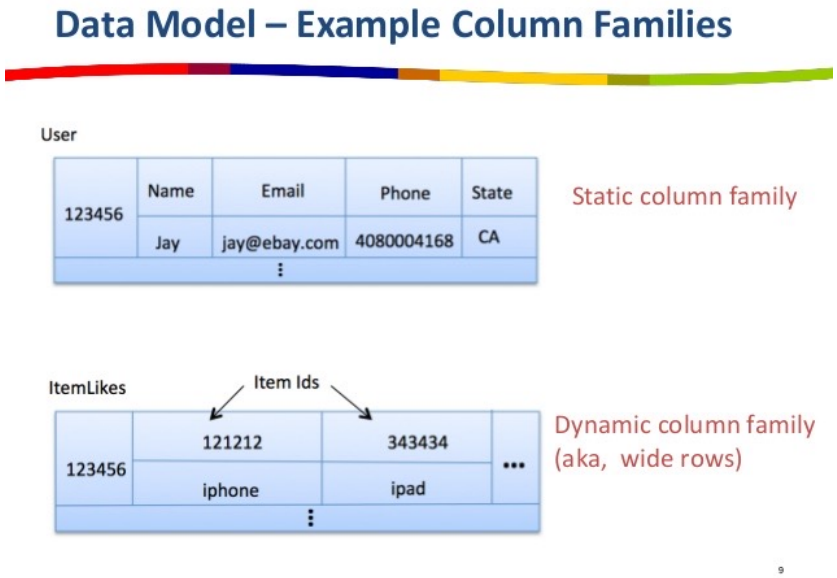
Rank Nov 2020	Rank Oct 2020	Rank Nov 2019	DBMS	Database Model	Score Nov 2020	Score Oct 2020	Score Nov 2019
1.	1.	1.	Oracle	Relational, Multi-model	1345.00	-23.77	+8.93
2.	2.	2.	MySQL	Relational, Multi-model	1241.64	-14.74	-24.64
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1037.64	-5.48	-44.27
4.	4.	4.	PostgreSQL	Relational, Multi-model	555.06	+12.66	+63.99
5.	5.	5.	MongoDB	Document, Multi-model	453.83	+5.81	+40.64
6.	6.	6.	IBM Db2	Relational, Multi-model	161.62	-0.28	-10.98
7.	↑ 8.	↑ 8.	Redis	Key-value, Multi-model	155.42	+2.14	+10.18
8.	↓ 7.	↓ 7.	Elasticsearch	Search engine, Multi-model	151.55	-2.29	+3.15
9.	9.	↑ 11.	SQLite	Relational	123.31	-2.11	+2.29
10.	10.	10.	Cassandra	Wide column	118.75	-0.35	-4.47
11.	11.	↓ 9.	Microsoft Access	Relational	117.23	-1.02	-12.84
12.	12.	↑ 13.	MariaDB	Relational, Multi-model	92.29	+0.52	+6.72
13.	13.	↓ 12.	Splunk	Search engine	89.71	+0.30	+0.64
14.	14.	↑ 15.	Teradata	Relational, Multi-model	75.60	-0.19	-4.75
15.	15.	↓ 14.	Hive	Relational	70.26	+0.71	-13.96
16.	16.	16.	Amazon DynamoDB	Multi-model	68.89	+0.48	+7.52
17.	17.	↑ 25.	Microsoft Azure SQL Database	Relational, Multi-model	66.99	+2.59	+39.37
18.	18.	↑ 19.	SAP Adaptive Server	Relational	55.39	+0.23	+0.10
19.	19.	↑ 20.	SAP HANA	Relational, Multi-model	53.58	-0.66	-1.53
20.	↑ 21.	↑ 22.	Neo4j	Graph	53.53	+2.20	+3.00
21.	↓ 20.	↓ 17.	Solr	Search engine	51.82	-0.66	-5.96
22.	22.	↓ 21.	HBase	Wide column	47.11	-1.25	-6.73
23.	23.	↓ 18.	FileMaker	Relational	46.66	-0.73	-9.07
24.	24.	↑ 27.	Google BigQuery	Relational	35.08	+0.67	+9.64
25.	25.	↓ 24.	Microsoft Azure Cosmos DB	Multi-model	32.50	+0.49	+0.52
26.	26.	↓ 23.	Couchbase	Document, Multi-model	30.55	+0.22	-1.44

Apache Cassandra

- Wide-table key value store



Relational Model	Cassandra Model
Database	Keyspace
Table	Column Family (CF)
Primary key	Row key
Column name	Column name/key
Column value	Column value

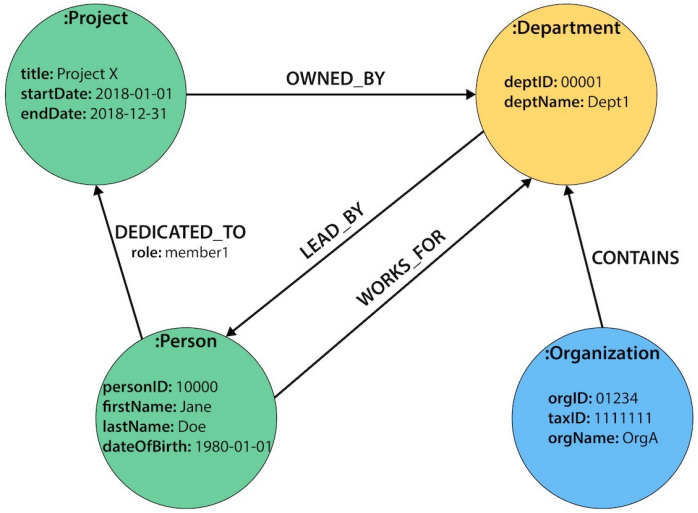



```
1  CREATE TABLE users_by_username (
2      username text PRIMARY KEY,
3      email text,
4      age int
5  )
6
7  CREATE TABLE users_by_email (
8      email text PRIMARY KEY,
9      username text,
10     age int
11 )
```

```
cqlsh> select * from University.Student;

 rollno | dept | name   | semester
--------+------+--------+----------
      1 |   CS | Jeegar |        5
      2 |   CS | Guru99 |        7

(2 rows)
cqlsh>
```

Neo4j

- Graph Database using a Property Graph Model



Cypher

```
MATCH (p:Product)
RETURN p.productName, p.unitPrice
ORDER BY p.unitPrice DESC
LIMIT 10;
```

Cypher Copy to Clipboard Run in Neo4j Browser

```
MATCH (p:Product {productName:"Chocolade"})<-[:PRODUCT]-(:Order)<-[:PURCHASED]-(c:Customer)
RETURN distinct c.companyName;
```

Summary

- Traditional databases don't provide the right abstractions for many newer data processing/analytics tasks
- Led to development of NoSQL systems and Map-Reduce (or similar) frameworks
  - Easier to get started
  - Easier to handle ad hoc and arbitrary tasks
  - Not as efficient
- Over the last 10 years, seen increasing convergence
  - NoSQL stores increasingly support SQL constructs like joins and aggregations
  - Map-reduce frameworks also evolved to support joins and SQL more explicitly
  - Databases evolved to support more data types, richer functionality for ad hoc processing
- Think of Map-Reduce systems as another option
  - Appropriate in some cases, not a good fit in other cases