

Distributed RDF Datastores

Dimitrios Tsoumakos

Some slides taken from :

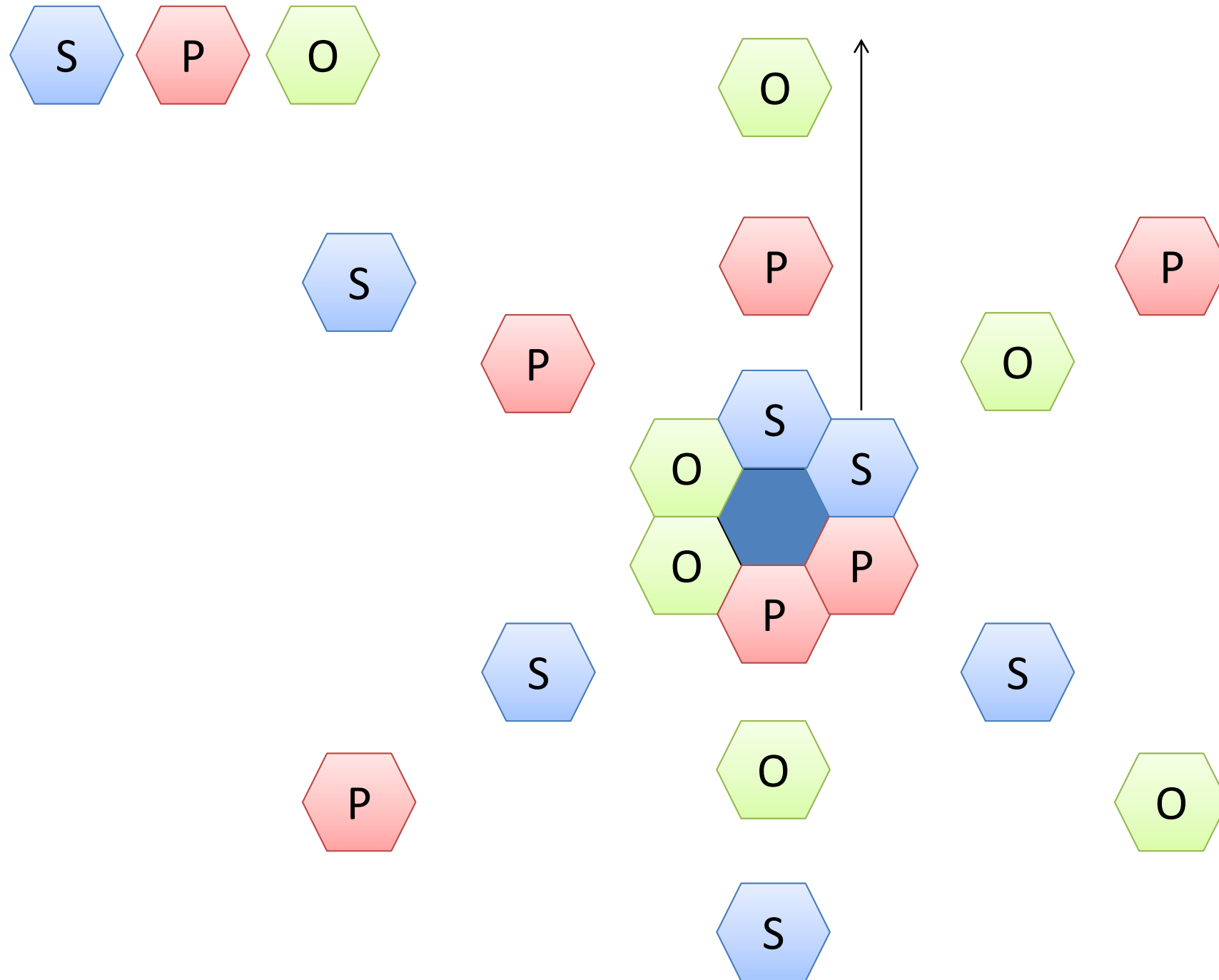
- Distributed Big Graph Management Methods and Systems, N. Papailiou
- Hexastore: Sextuple Indexing for Semantic Web Data Management, C. Weiss, P. Karras, et al
- Matrix “Bit” loaded: A Scalable Lightweight Join Query Processor for RDF Data, Medha Atre, et al

**REMINDER: IMPORTANT NATIVE-INDEXING
DATASTORES**

Native RDF indexing - Hexastore

- ▶ Hexastore [Weiss et al. 2008]
 - Κρατάμε όλα τα sorted permutations των triples
 - SPO, SOP, PSO, POS, OSP, OPS
 - Όλα τα query patterns μπορούν να απαντηθούν με ένα Index scan
 - Όλα τα αρχικά join μπορούν να γίνουν με αποδοτικά merge-joins
 - Πιο ακριβά hash ή sort-merge joins χρειάζονται μόνο όταν κάνουμε join non-ordered intermediate results

Hexastore: Sextuple Indexing



Five-fold Increase in Index Space

- Sharing The Same Terminal Lists
 - SPO-PSO, SOP-OSP, POS-OPS

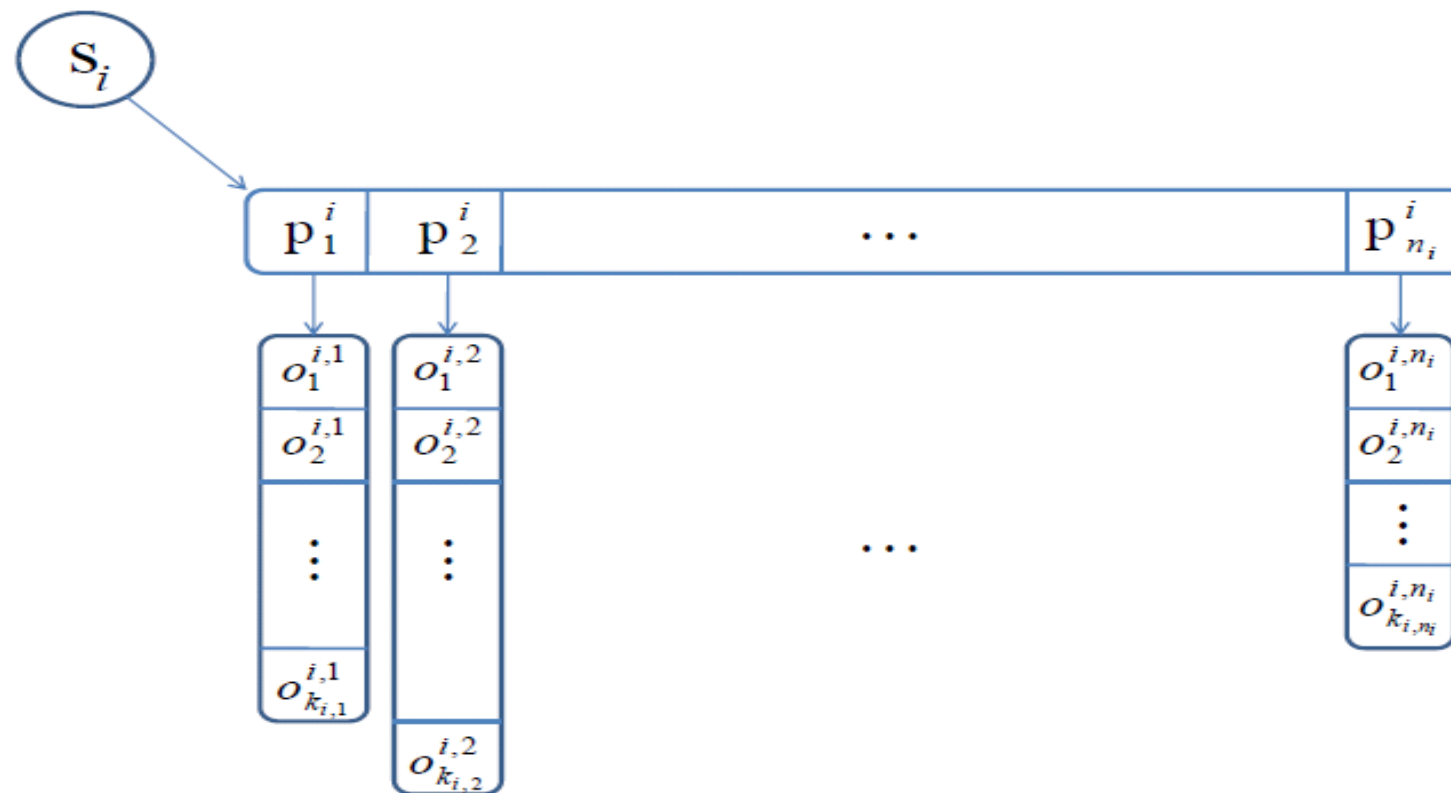
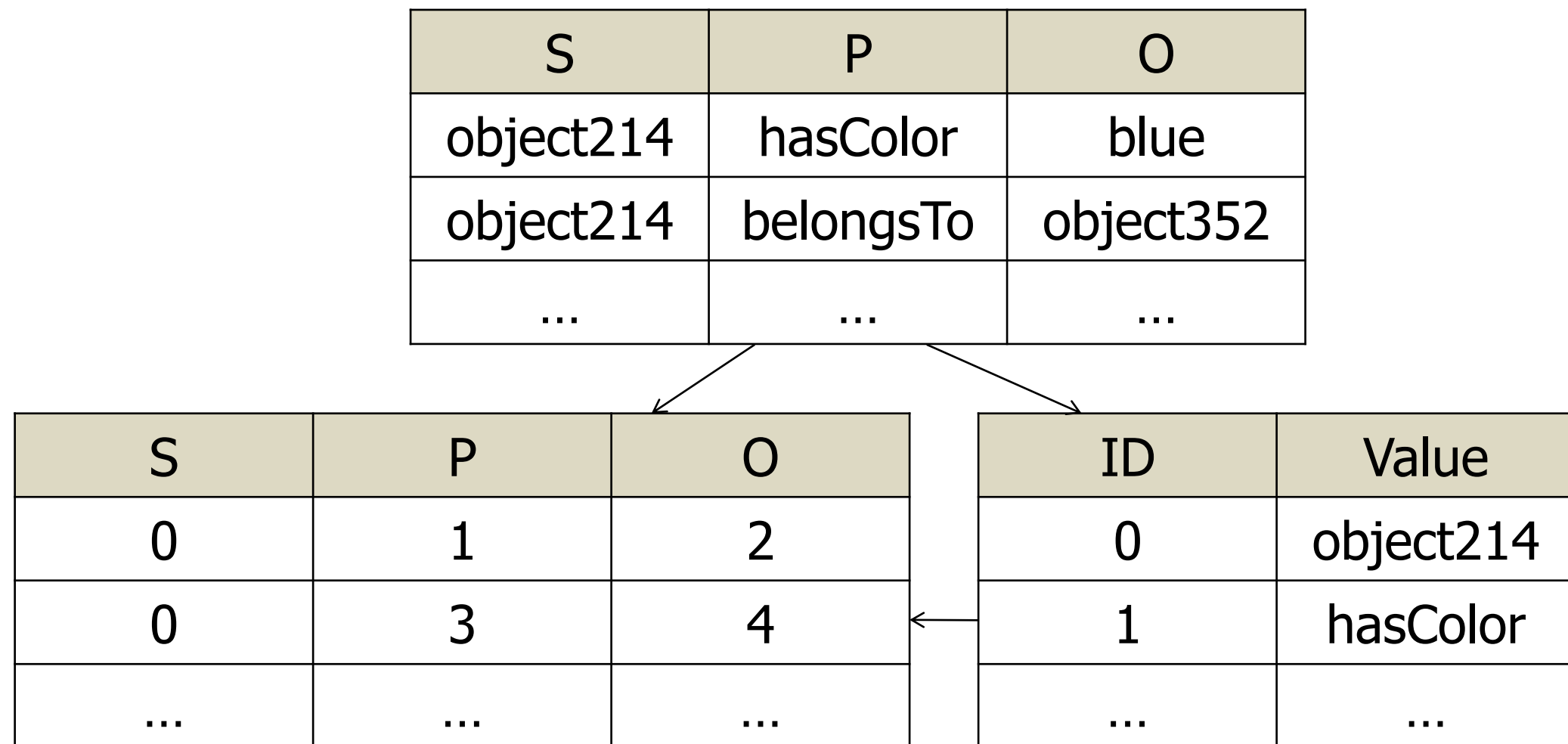


Figure 2: spo indexing in a Hexastore

- The key of each of the three resources in a triple appears in two headers and two vectors, but only in one list

Mapping Dictionary

- Replacing all literals by unique IDs using a mapping dictionary



- Mapping dictionary compresses the triple store
 - Reduced redundancy, Saving a lot of physical space
- We can concentrate on a logical index structure rather than the physical storage design

Clustered B⁺-Tree (RDF-3X, VLDB 2008)

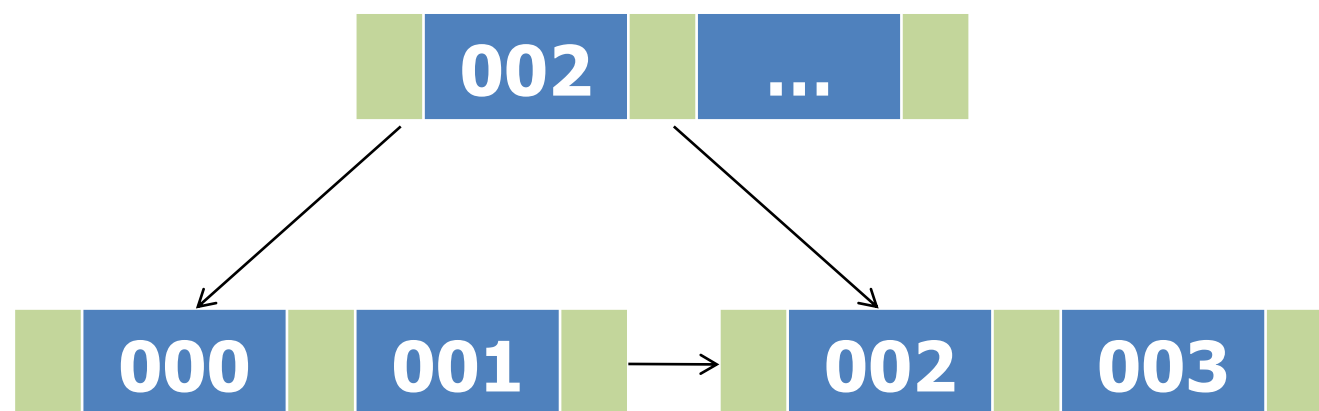
S	P	O
0	1	2
0	3	4
...

Actually, we don't need this table!

- Store everything in a clustered B⁺-Tree
 - Triples are sorted in lexicographical order
 - Allowing the conversion of SPARQL patterns into range scan
 - We don't have to do entire table scan

<Mapping Dictionary>

ID	Value
0	object214
1	hasColor
...	...



Native RDF indexing – RDF3X

Query pattern			Index
Subject	Predicate	Object	
—	—	—	όλα
?	—	—	pos, ops
—	?	—	osp, sop
—	—	?	spo, pso
?	?	—	osp, ops
—	?	?	spo, sop
?	—	?	pos, pso
?	?	?	όλα

- ▶ RDF-3X [Neumann et al. 2008]
 - 6 indexes, aggregated indexes για στατιστικά και εύρεση του βέλτιστου πλάνου εκτέλεσης
 - Aggressive compression

BIG DATA, MODERN DISTRIBUTED COMPUTE ENGINES AND NOSQL DATABASES OVERVIEW

Google™

Processes 20 PB a day (2008)

Crawls 20B web pages a day (2012)

Search index is 100+ PB (5/2014)

Bigtable serves 2+ EB, 600M QPS
(5/2014)

YAHOO!

Hadoop: 365 PB, 330K
nodes (6/2014)

ebay

Hadoop: 10K nodes,
150K cores, 150 PB
(4/2014)

300 PB data in Hive +
600 TB/day (4/2014)

facebook

amazon web services™

S3: 2T objects, 1.1M
request/second (4/2013)

640K ought to be
enough for
anybody.



400B pages,
10+ PB
(2/2014)

JPMorganChase

150 PB on 50k+ servers
running 15k apps (6/2011)

LHC: ~15 PB a year



LSST: 6-10 PB a year
(~2020)

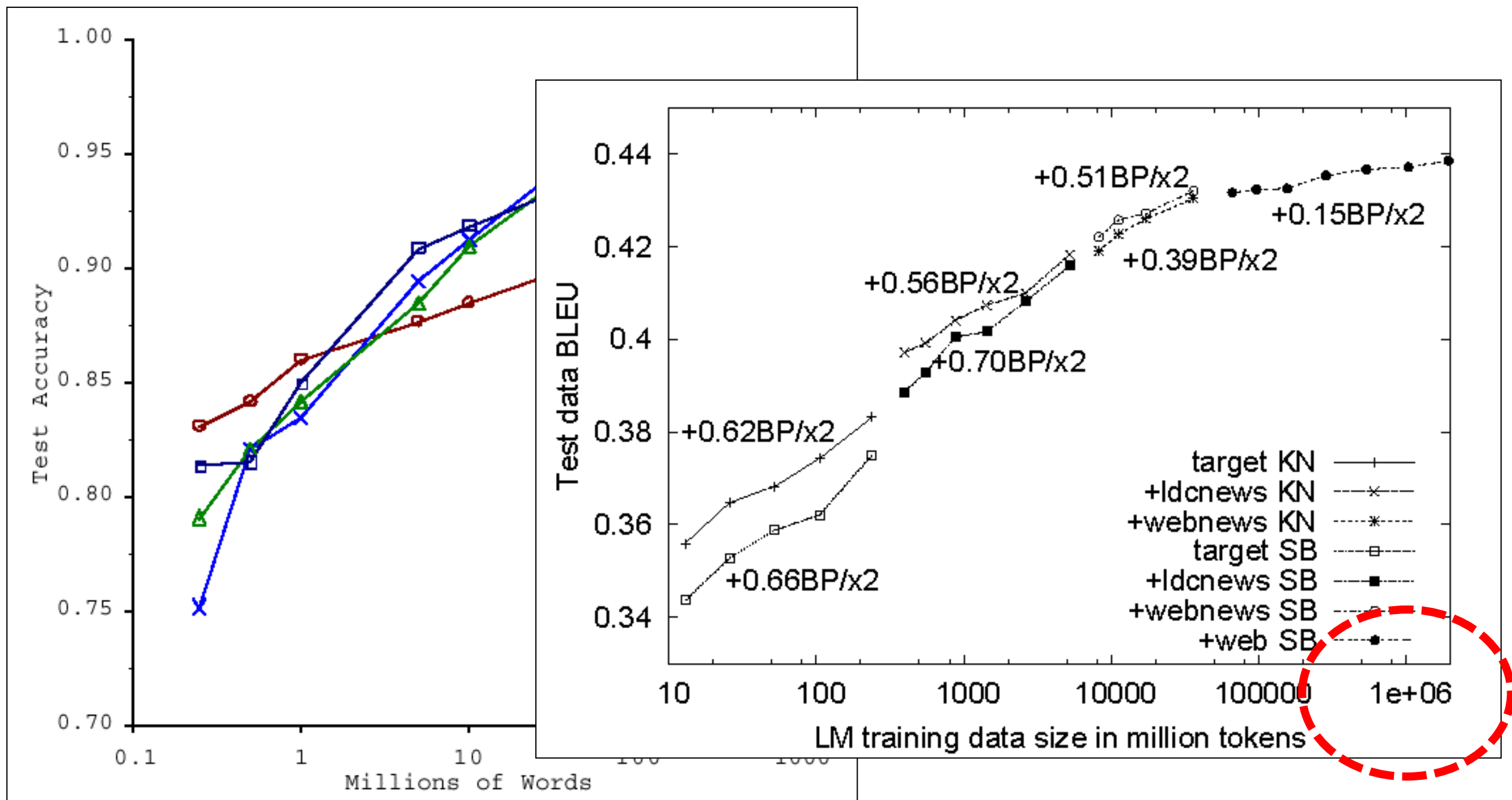
SKA: 0.3 – 1.5 EB
per year (~2020)



How much data?

No data like more data!

s/knowledge/data/g;



How do we get here if we're not Google?

What is cloud computing?

Just a buzzword?

- Before clouds...
 - P2P computing
 - Grids
 - HPC
 - ...
- Cloud computing means many different things:
 - Large-data processing
 - Rebranding of web 2.0
 - Utility computing
 - Everything as a service

Rebranding of web 2.0

- Rich, interactive web applications
 - Clouds refer to the servers that run them
 - AJAX as the de facto standard (for better or worse)
 - Examples: Facebook, YouTube, Gmail, ...
- “The network is the computer”: take two
 - User data is stored “in the clouds”
 - Rise of the netbook, smartphones, etc.
 - Browser *is* the OS

Utility Computing

- What?

- Computing resources as a metered service (“pay as you go”)
- Ability to dynamically provision virtual machines

- Why?

- Cost: capital vs. operating expenses
- Scalability: “infinite” capacity
- Elasticity: scale up or down on demand

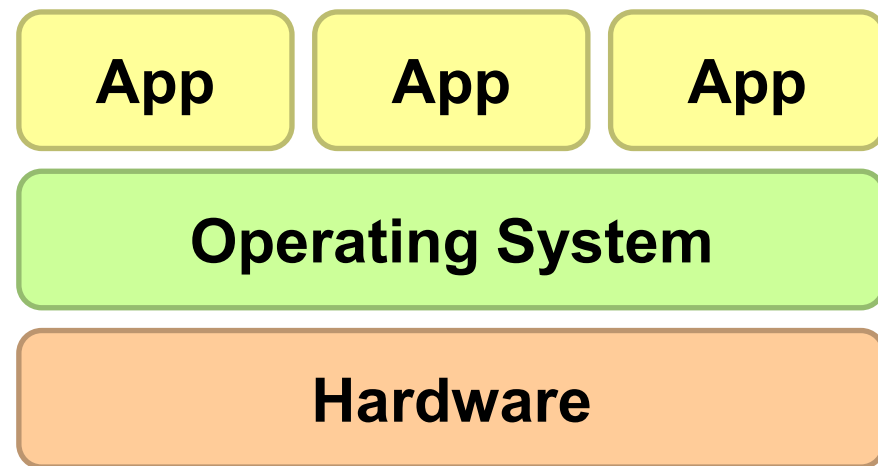
- Does it make sense?

- Benefits to cloud users
- Business case for cloud providers

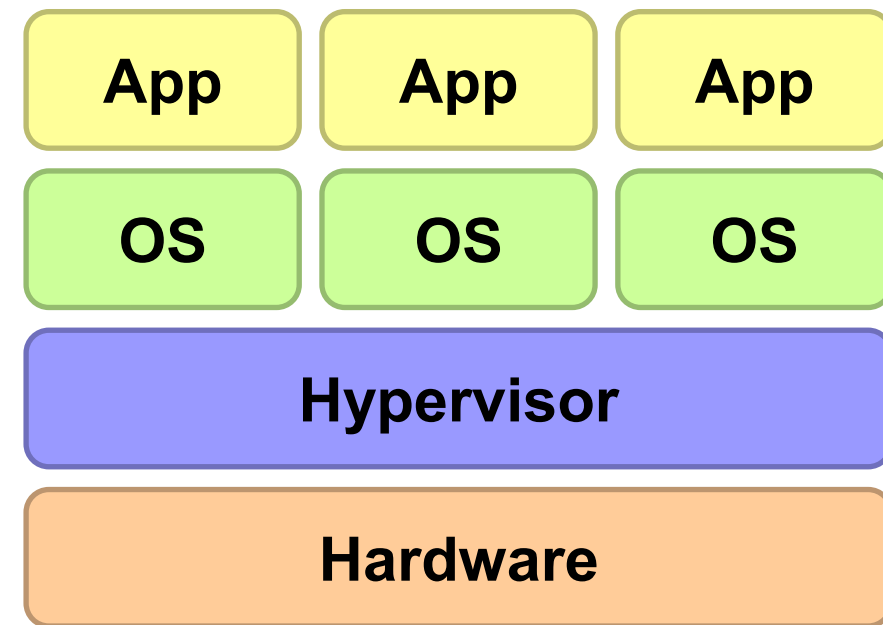
I think there is a world market for about five computers.



Enabling Technology: Virtualization



Traditional Stack



Virtualized Stack

Cloud computing market

Software as a service

Everything is a service

Platform as a service

Infrastructure as a service

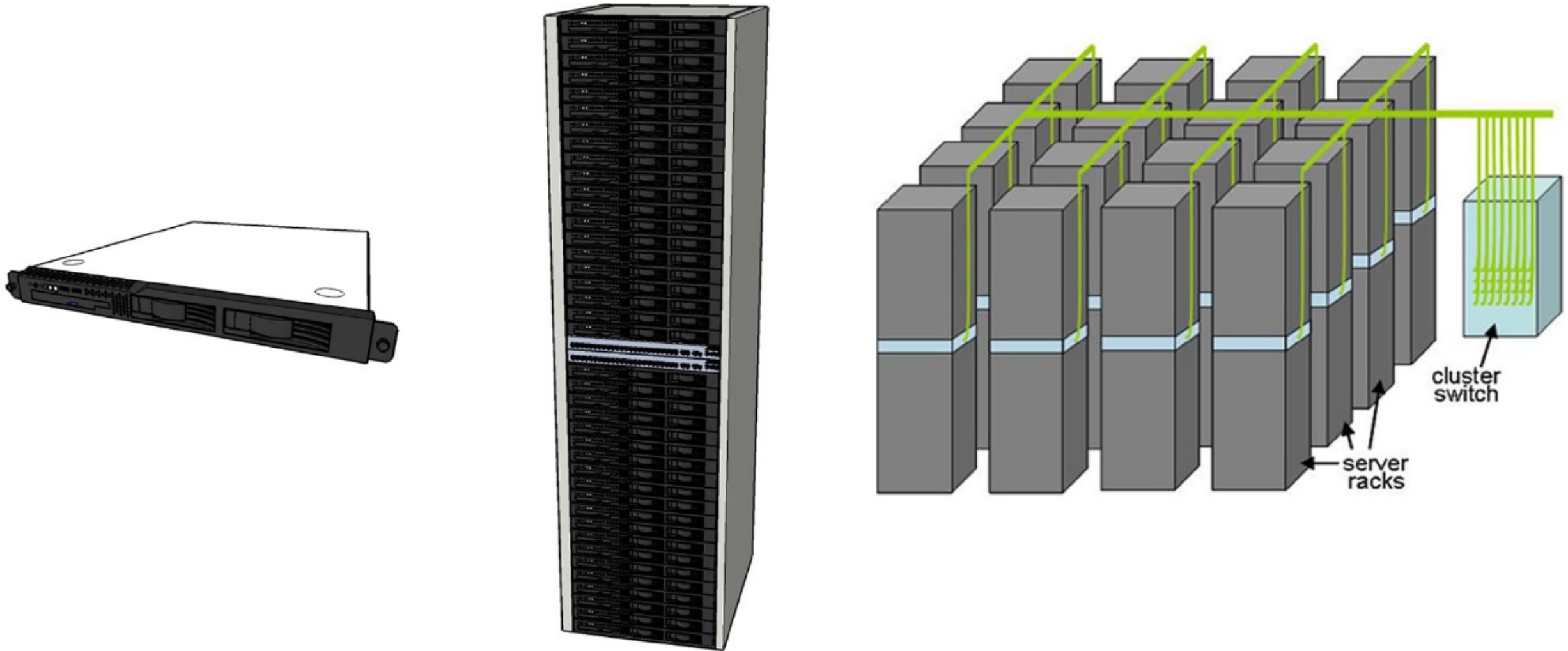
Cloud technology enabler

Hardware provider

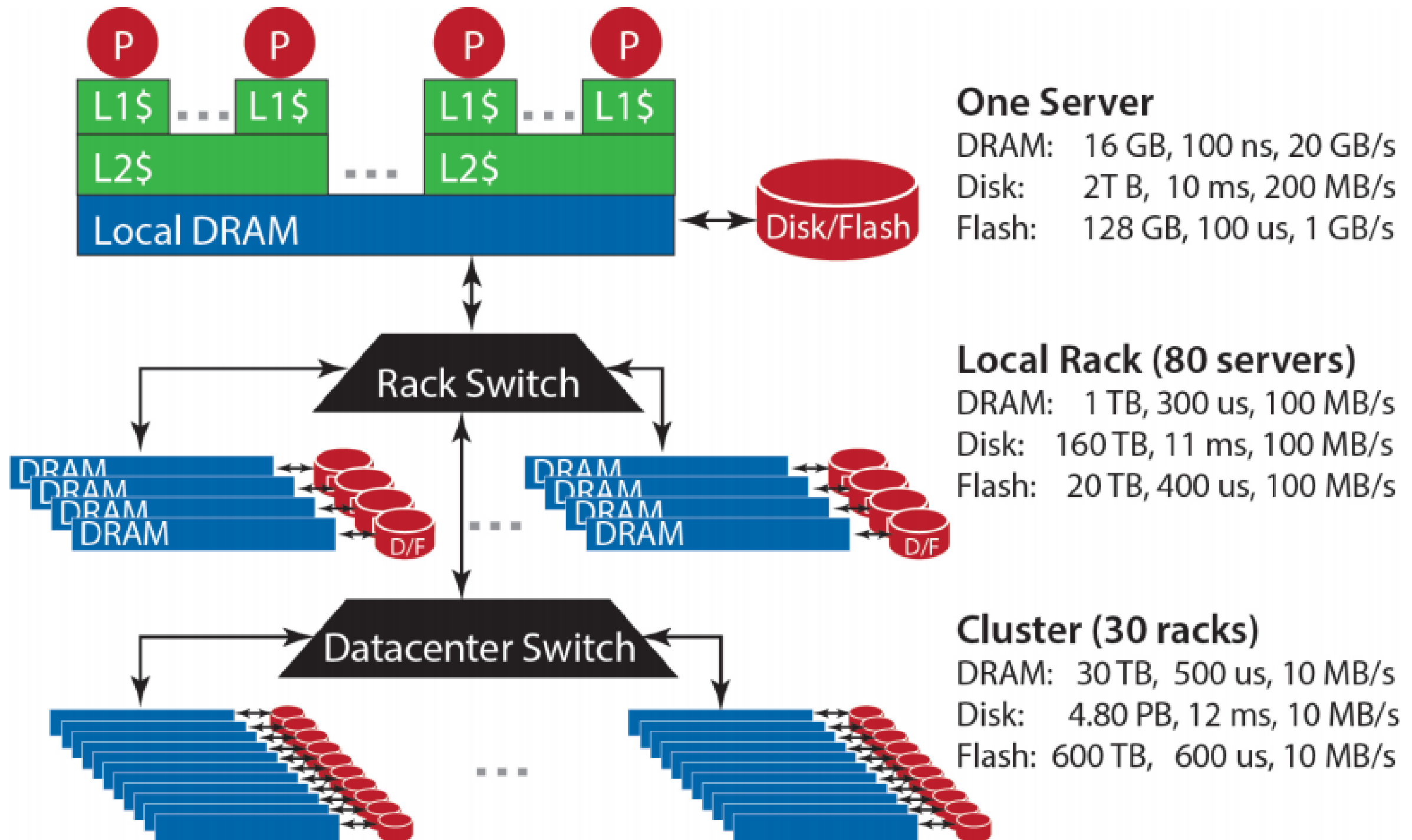
Everything as a Service

- Utility computing = Infrastructure as a Service (IaaS)
 - Why buy machines when you can rent cycles?
 - Examples: Amazon's EC2, Rackspace
- Platform as a Service (PaaS)
 - Give me nice API and take care of the maintenance, upgrades, ...
 - Example: Google App Engine
- Software as a Service (SaaS)
 - Just run it for me!
 - Example: Gmail, Salesforce

Building Blocks

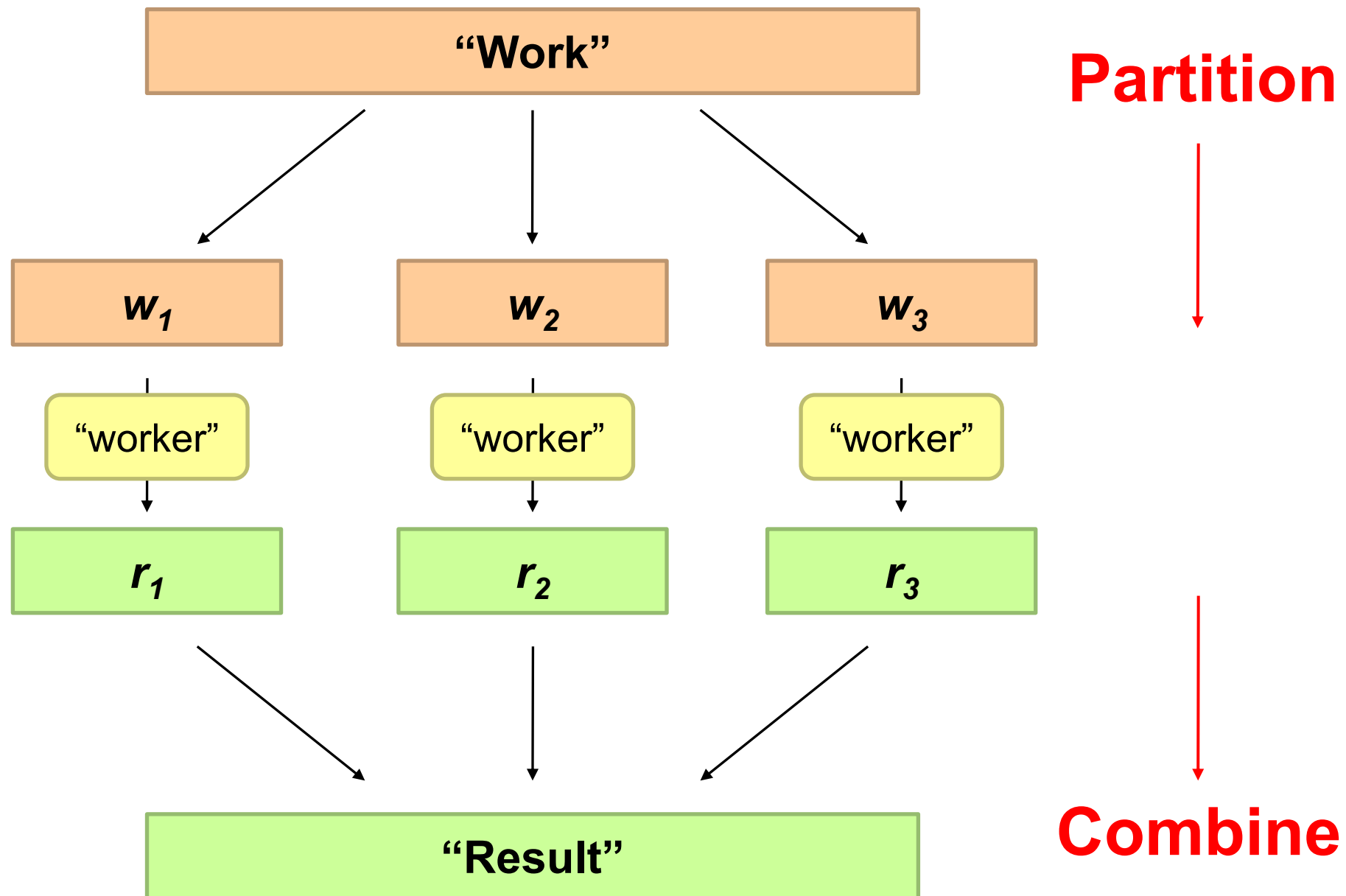


Storage Hierarchy



How do we scale up?

Divide and Conquer



Parallelization Challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

What is the common theme of all of these problems?

Synchronization!

- Parallelization problems arise from:
 - Communication between workers (e.g., to exchange state)
 - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

Managing Multiple Workers

- Difficult because
 - We don't know the order in which workers run
 - We don't know when workers interrupt each other
 - We don't know the order in which workers access shared data
- Thus, we need:
 - Semaphores (lock, unlock)
 - Conditional variables (wait, notify, broadcast)
 - Barriers
- Still, lots of problems:
 - Deadlock, livelock, race conditions...
 - Dining philosophers, sleeping barbers, cigarette smokers...
- Moral of the story: be careful!

“Big Ideas”

- Scale “out”, not “up”
 - Limits of SMP and large shared-memory machines
- Move processing to the data
 - Cluster have limited bandwidth
- Process data sequentially, avoid random access
 - Seeks are expensive, disk throughput is reasonable
- Seamless scalability
 - From the mythical man-month to the tradable machine-hour

MapReduce

What is MapReduce?

- Programming model for expressing distributed computations at a massive scale
- Execution framework for organizing and performing such computations
- Open-source implementation called Hadoop



Typical Large-Data Problem

- Iterate over a large number of records
- Map** ○ Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results **Reduce**
- Generate final output

Key idea: provide a functional abstraction for these two operations

Challenges

1. Cheap nodes fail, especially if you have many

- Mean time between failures for 1 node = 3 years
- Mean time between failures for 1000 nodes = 1 day
- Solution: Build fault-tolerance into system

2. Commodity network = low bandwidth

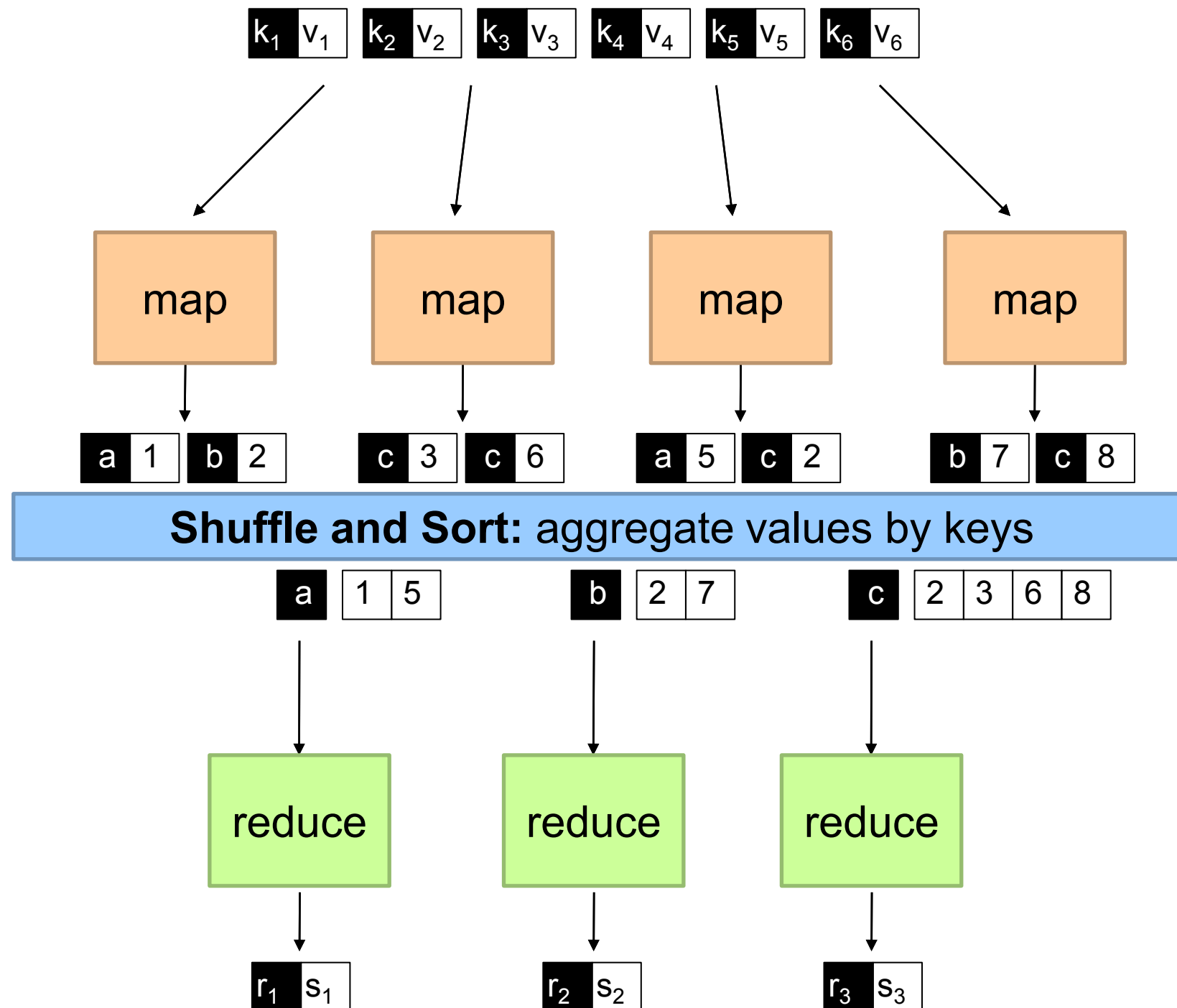
- Solution: Push computation to the data

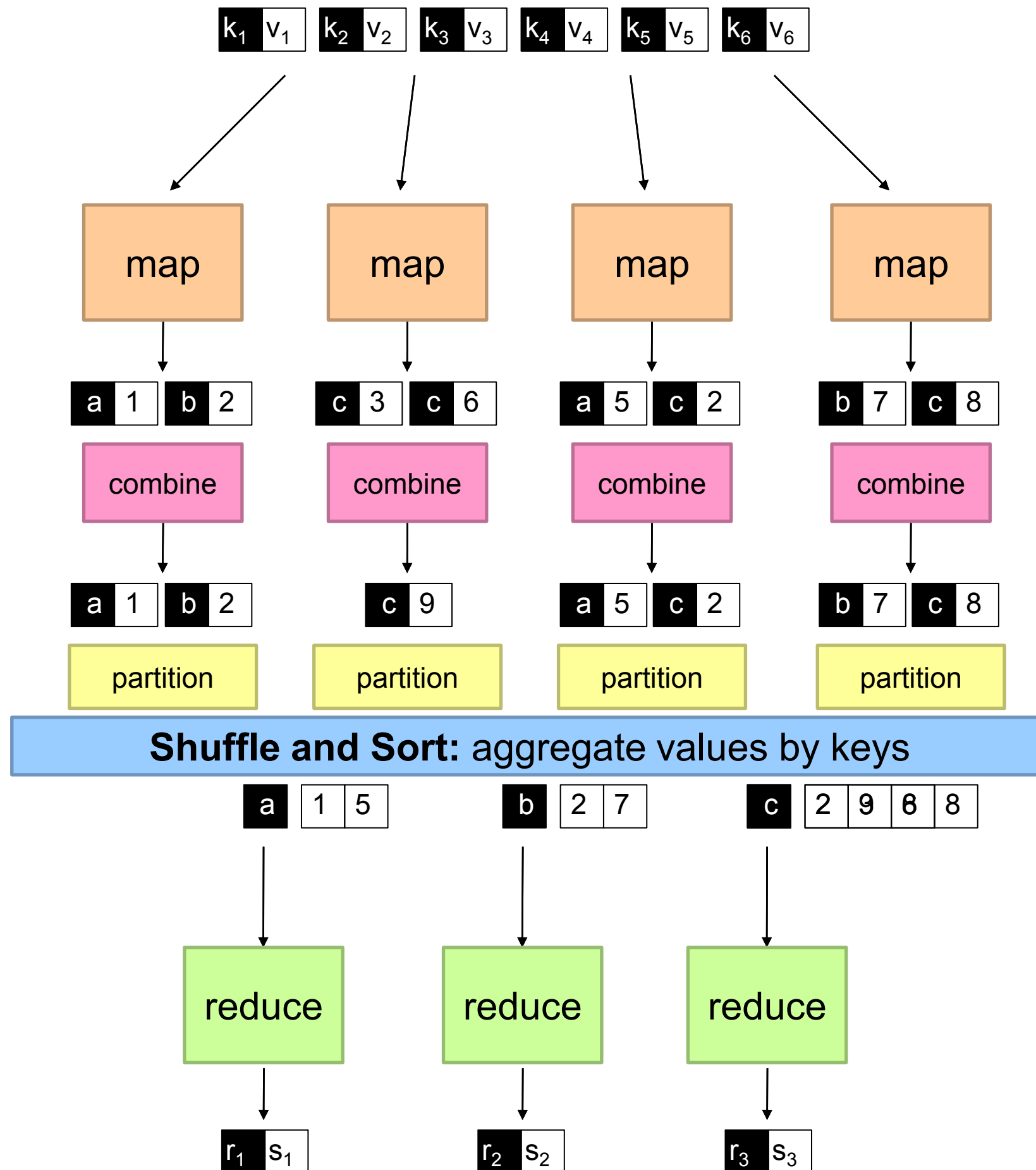
3. Programming distributed systems is hard

- Solution: Data-parallel programming model: users write “map” & “reduce” functions, system distributes work and handles faults

MapReduce

- Programmers specify two functions:
 - map** $(k, v) \rightarrow \langle k', v' \rangle^*$
 - reduce** $(k', v') \rightarrow \langle k'', v'' \rangle^*$
 - All values with the same key are reduced together
- The execution framework handles everything else...
- Not quite...usually, programmers also specify:
 - partition** $(k', \text{number of partitions}) \rightarrow \text{partition for } k'$
 - Often a simple hash of the key, e.g., $\text{hash}(k') \bmod n$
 - Divides up key space for parallel reduce operations
 - combine** $(k', v') \rightarrow \langle k', v' \rangle^*$
 - Mini-reducers that run in memory after the map phase
 - Used as an optimization to reduce network traffic





MapReduce “Runtime”

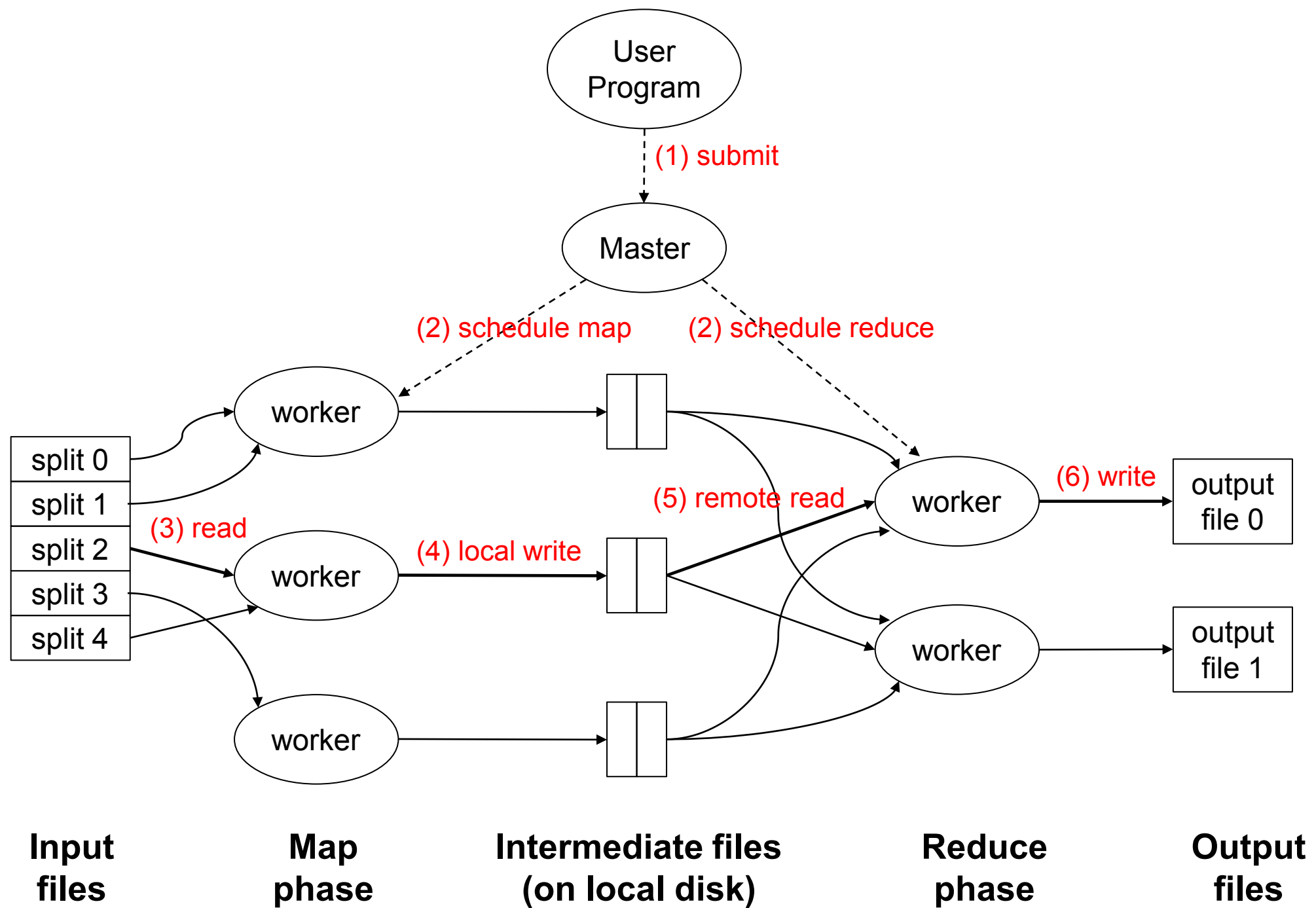
- Handles scheduling
 - Assigns workers to map and reduce tasks
- Handles “data distribution”
 - Moves processes to data
- Handles synchronization
 - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
 - Detects worker failures and restarts
- Everything happens on top of a distributed FS

Two more details...

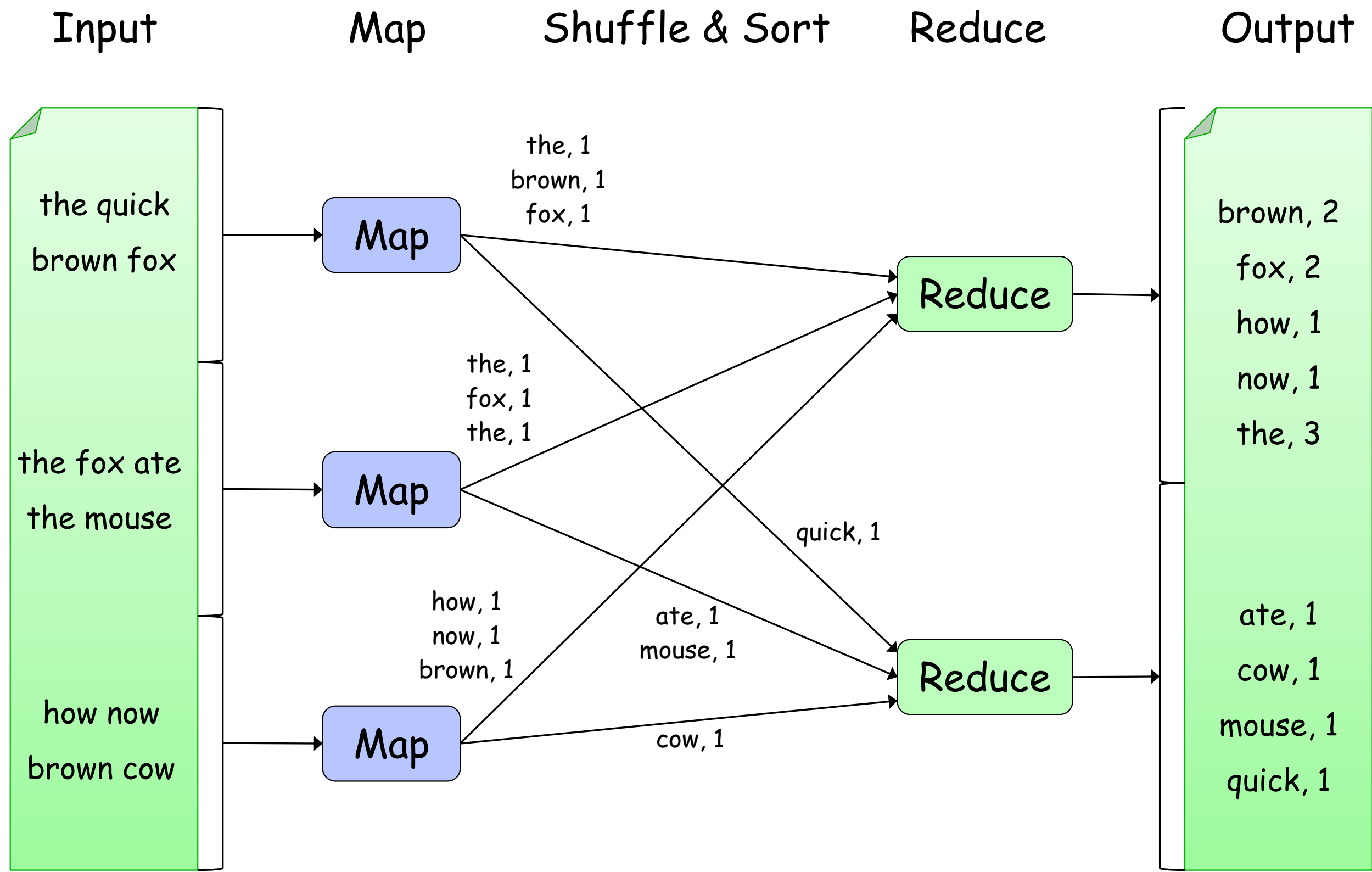
- Barrier between map and reduce phases
 - But we can begin copying intermediate data earlier
- Keys arrive at each reducer in sorted order
 - No enforced ordering *across* reducers

MapReduce Execution

- Single *master* controls job execution on multiple *slaves*
- Mappers preferentially placed on same node or same rack as their input block
 - Minimizes network usage
- Mappers save outputs to local disk before serving them to reducers
 - Allows recovery if a reducer crashes
 - Allows having more reducers than nodes

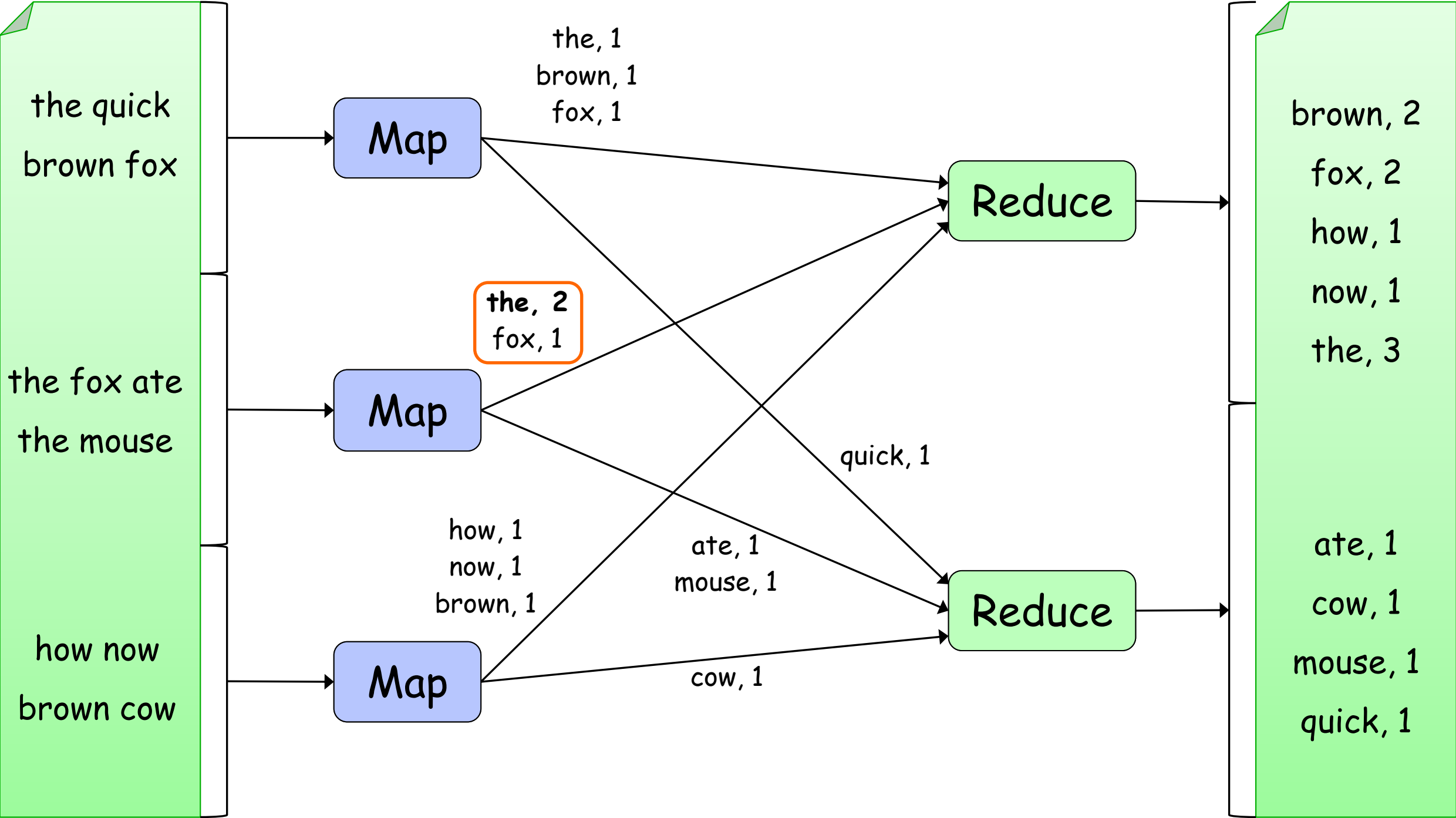


Word Count Execution



Word Count with Combiner

Input Map & Combine Shuffle & Sort Reduce Output



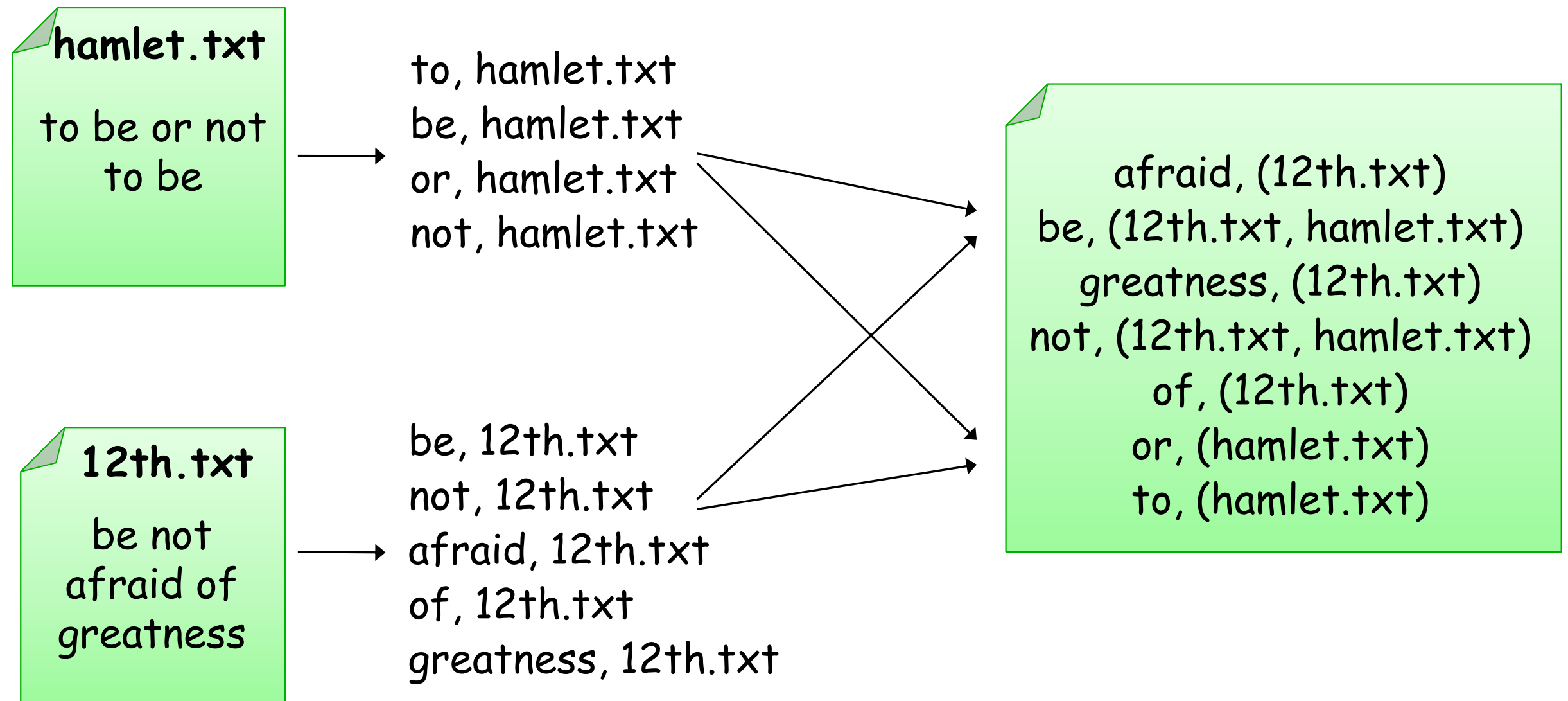
Inverted Index Example

- **Input:** (filename, text) records
- **Output:** list of files containing each word
- **Map:**

```
foreach word in text.split():  
    output(word, filename)
```
- **Combine:** uniquify filenames for each word
- **Reduce:**

```
def reduce(word, filenames):  
    output(word, sort(filenames))
```


Inverted Index Example



Hadoop Components

- **Distributed file system (HDFS)**

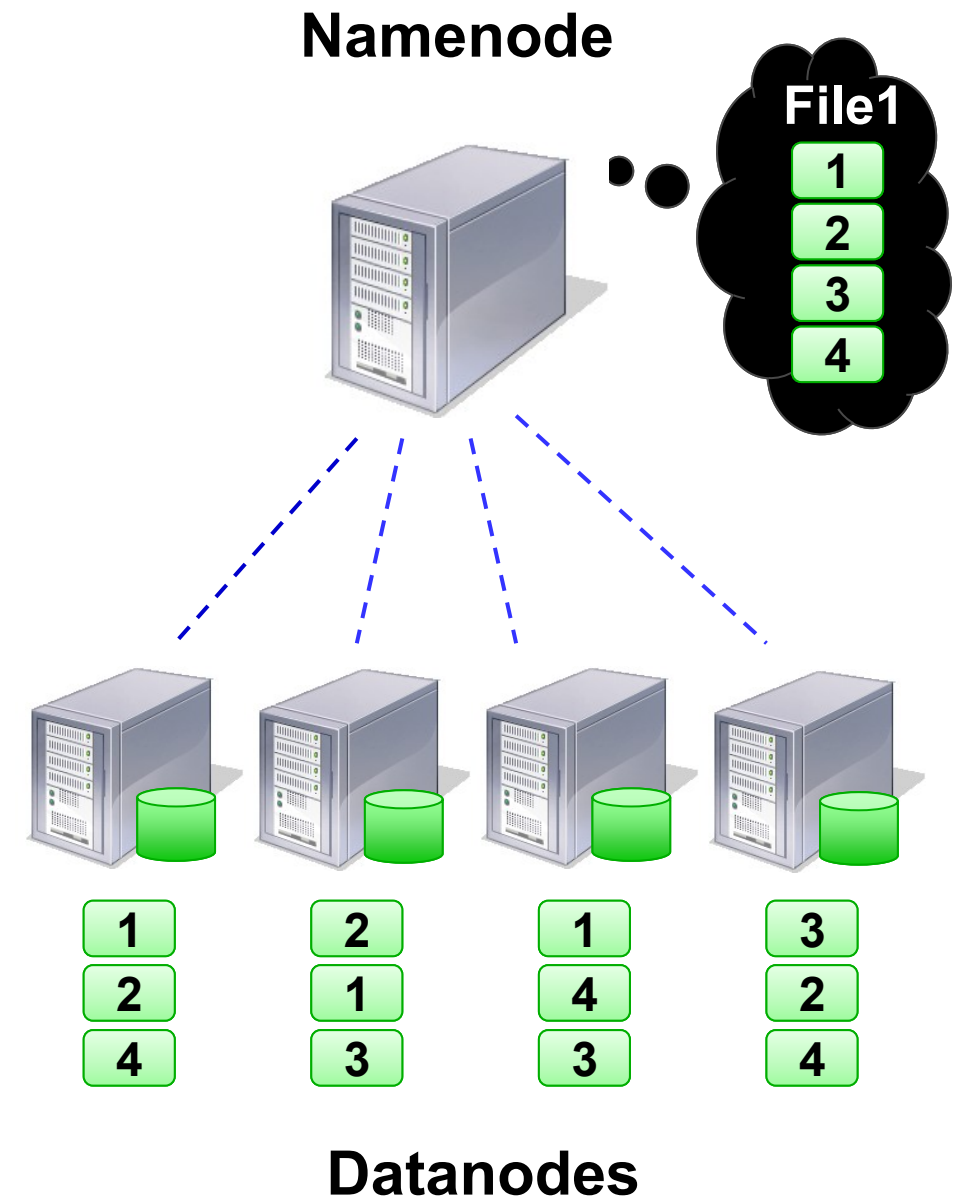
- Single namespace for entire cluster
- Replicates data 3x for fault-tolerance

- **MapReduce framework**

- Executes user jobs specified as “map” and “reduce” functions
- Manages work distribution & fault-tolerance

Hadoop Distributed File System

- Files split into 64MB *blocks*
- Blocks replicated across several *datanodes* (usually 3)
- Single *namenode* stores metadata (file names, block locations, etc)
- Optimized for large files, sequential reads
- Files are append-only





Sequoia

16.32 PFLOPS

98,304 nodes with 1,572,864 million cores

1.6 petabytes of memory

7.9 MWatts total power

Introduction to NoSQL, HBase

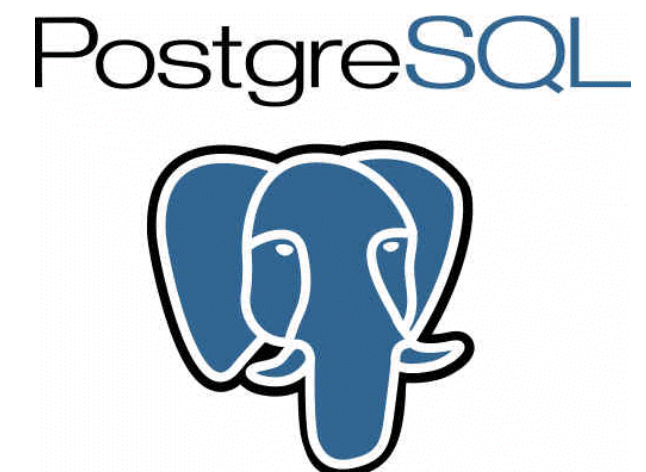
Material adapted from slides by :

Perry Hoekstra and Gary Dusbabek(Rackspace)

CS 525 **Indranil Gupta**

SQL

- Specialized data structures (think B-trees)
 - Shines with complicated queries
- Focus on fast query & analysis
 - Not necessarily on large datasets



Scaling Up

- Issues with scaling up when the dataset is just too big
- RDBMS were not designed to be distributed
- Began to look at multi-node database solutions
- Known as 'scaling out' or 'horizontal scaling'
- Different approaches include:
 - Master-slave
 - Sharding

What is NoSQL?

- Stands for **Not Only SQL**
- Class of non-relational data storage systems
- Usually do not require a fixed table schema nor do they use the concept of joins
- All NoSQL offerings relax one or more of the ACID properties (will talk about the CAP theorem)

How did we get here?

- Explosion of social media sites (Facebook, Twitter) with large data needs
- Rise of cloud-based solutions such as Amazon S3 (simple storage solution)
- Just as moving to dynamically-typed languages (Ruby/Groovy), a shift to dynamically-typed data with frequent schema changes
- Open-source community

More Programming and Less Database Design

Alternative to traditional relational DBMS

- + Flexible schema
- + Quicker/cheaper to set up
- + Massive scalability
- + Relaxed consistency → higher performance & availability
- No declarative query language → more programming
- Relaxed consistency → fewer guarantees

Challenge: Coordination

- The solution to availability and scalability is to decentralize and replicate functions and data...but how do we coordinate the nodes?
 - data consistency
 - update propagation
 - mutual exclusion
 - consistent global states
 - group membership
 - group communication
 - event ordering
 - distributed consensus
 - quorum consensus



Dynamo and BigTable

- Three major papers were the seeds of the NoSQL movement
 - BigTable (Google)
 - Dynamo (Amazon)
 - Gossip protocol (discovery and error detection)
 - Distributed key-value data store
 - Eventual consistency
 - CAP Theorem

CAP Theorem

- Proposed by Eric Brewer (Berkeley)
- Subsequently proved by Gilbert and Lynch
- In a distributed system you can satisfy at most 2 out of the 3 guarantees
 - 1. Consistency:** all nodes have same data at any time
 - 2. Availability:** the system allows operations all the time
 - 3. Partition-tolerance:** the system continues to work in spite of network partitions

Consistency

C

Fox&Brewer “CAP Theorem”:
C-A-P: choose two.

Claim: every distributed system is on one side of the triangle.

CA: available, and consistent, unless there is a partition.

CP: always consistent, even in a partition, but a reachable replica may deny service without agreement of the others (e.g., quorum).

A

Availability

AP: a reachable replica provides service even in a partition, but may be inconsistent if there is a failure.

P

Partition-resilience

Availability

- Traditionally, thought of as the server/process available five 9's (99.999 %).
- However, for large node system, at almost any point in time there's a good chance that a node is either down or there is a network disruption among the nodes.
 - Want a system that is resilient in the face of network disruption

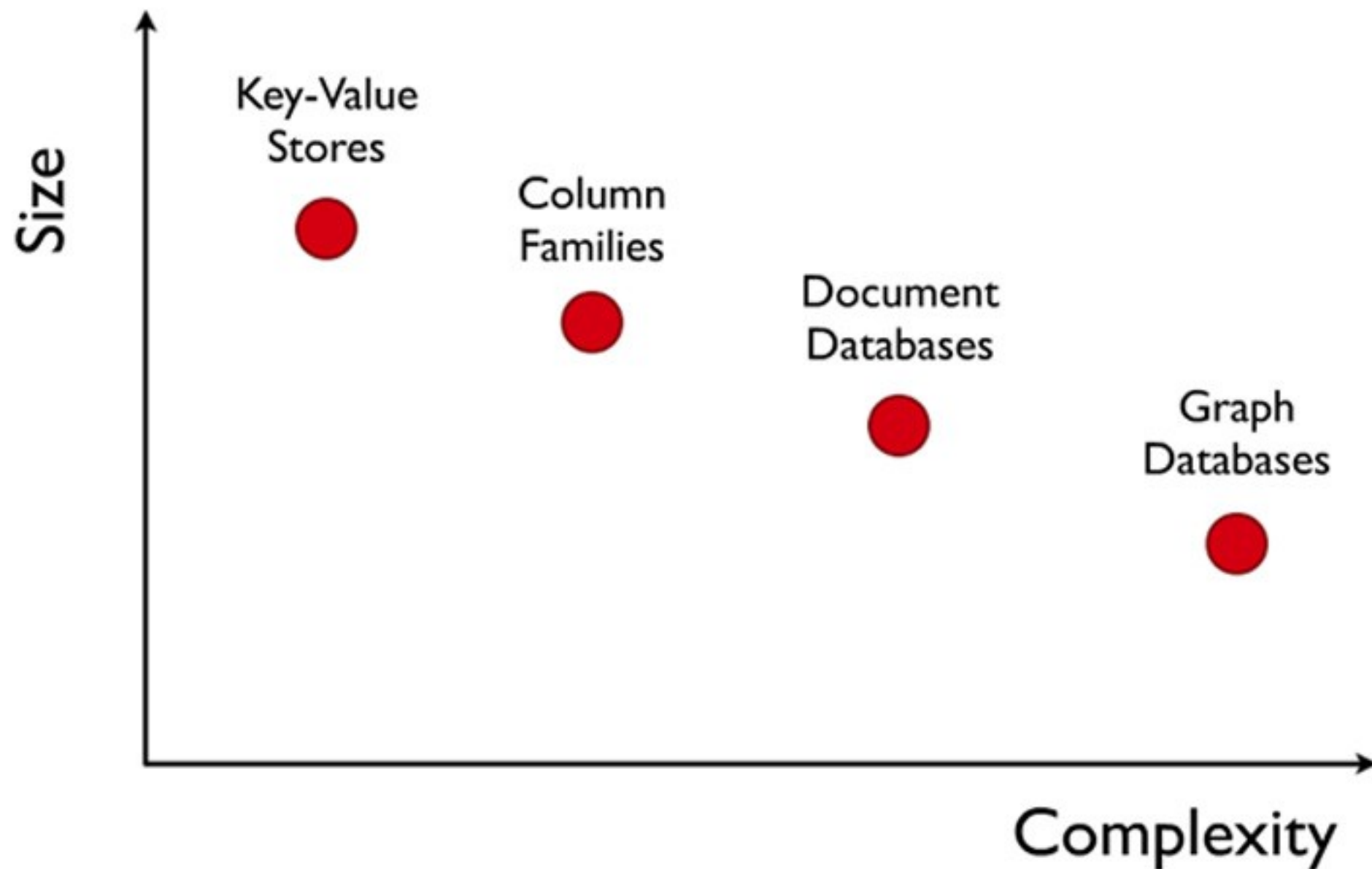
Consistency Model

- A consistency model determines rules for visibility and apparent order of updates.
- For example:
 - Row X is replicated on nodes M and N
 - Client A writes row X to node N
 - Some period of time t elapses.
 - Client B reads row X from node M
 - Does client B see the write from client A?
 - Consistency is a continuum with tradeoffs
 - For NoSQL, the answer would be: maybe
 - CAP Theorem states: Strict Consistency can't be achieved at the same time as availability and partition-tolerance.

Eventual Consistency

- When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
- For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service
- Known as BASE (**B**asically **A**vailable, **S**oft state, **E**ventual consistency), as opposed to ACID
 - Soft state: copies of a data item may be inconsistent
 - Eventually Consistent – copies becomes consistent at some later time if there are no more updates to that data item
 - Basically Available – possibilities of faults but not a fault of the whole system

NoSQL Categories



Categories of NoSQL databases

- Key-value stores
- Column NoSQL databases
- Document-based
- Graph database (neo4j, InfoGrid)
- XML databases (myXMLDB, Tamino, Sedna)

Key/Value

Pros:

- very fast
- very scalable
- simple model
- able to distribute horizontally

Cons:

- many data structures (objects) can't be easily modeled as key value pairs

Schema-Less

Pros:

- Schema-less data model is richer than key/value pairs
- eventual consistency
- many are distributed
- still provide excellent performance and scalability

Cons:

- typically no ACID transactions or joins

Common Advantages

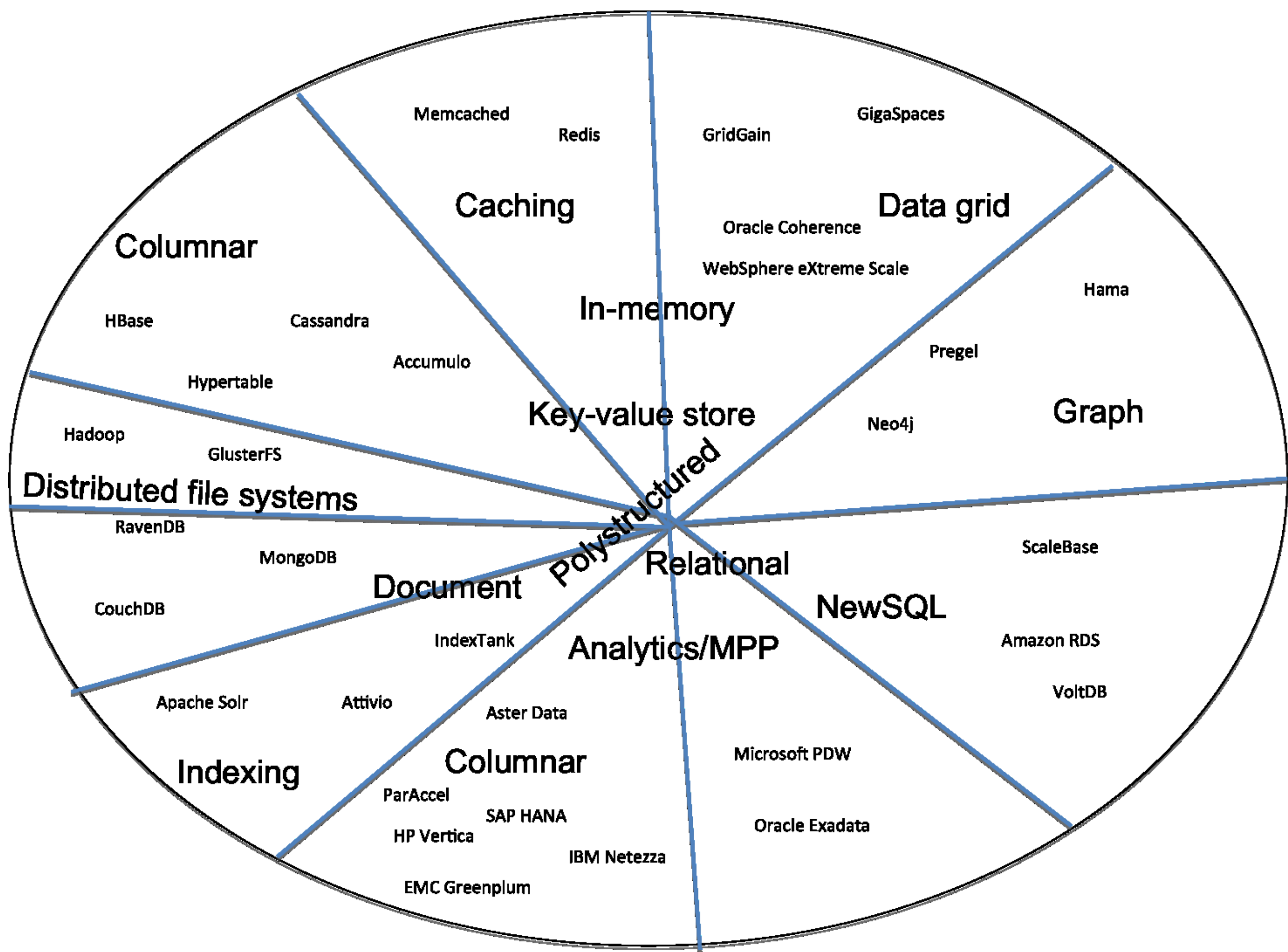
- Cheap, easy to implement (open source)
- Data are replicated to multiple nodes (therefore identical and fault-tolerant) and can be partitioned
 - Down nodes easily replaced
 - No single point of failure
- Easy to distribute
- Don't require a schema
- Can scale up and down
- Relax the data consistency requirement (CAP)

Typical NoSQL API

- Basic API access:
 - `get(key)` -- Extract the value given a key
 - `put(key, value)` -- Create or update the value given its key
 - `delete(key)` -- Remove the key and its associated value
 - `execute(key, operation, parameters)` -- Invoke an operation to the value (given its key) which is a special data structure (e.g. List, Set, Map etc).

What am I giving up?

- joins
- group by
- order by
- ACID transactions
- SQL as a sometimes frustrating but still powerful query language
- easy integration with other applications that support SQL





An Introduction to Hadoop HBase

HBase is ...

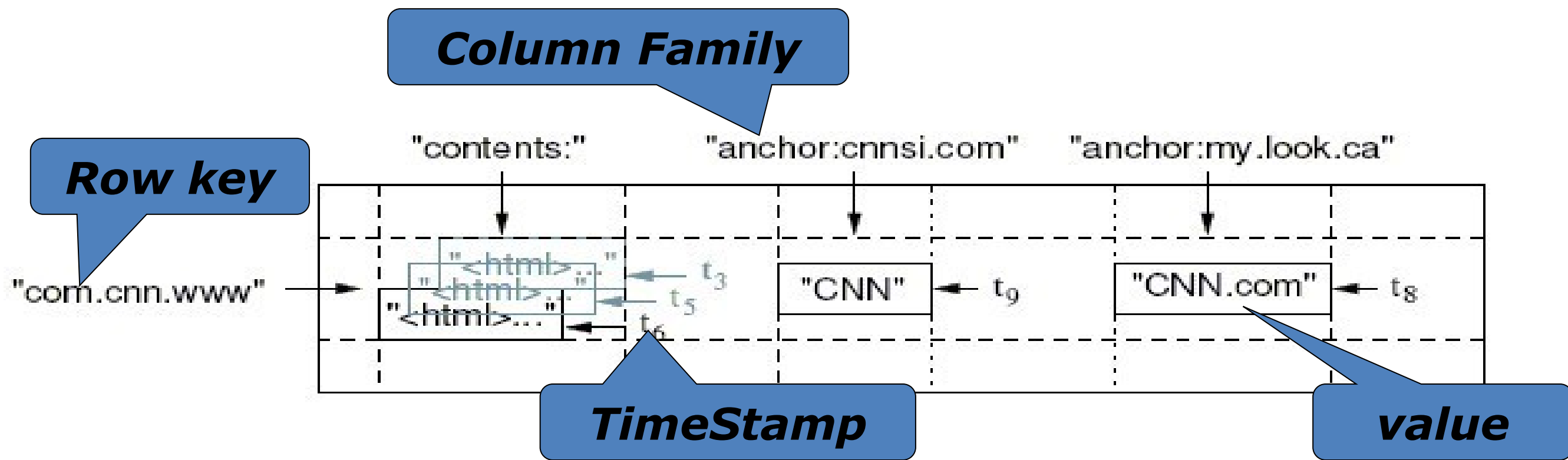
- A distributed data store that can scale horizontally to 1,000s of commodity servers and petabytes of indexed storage.
- Designed to operate on top of the Hadoop distributed file system (HDFS) or Kosmos File System (KFS, aka Cloudstore) for scalability, fault tolerance, and high availability.

Benefits

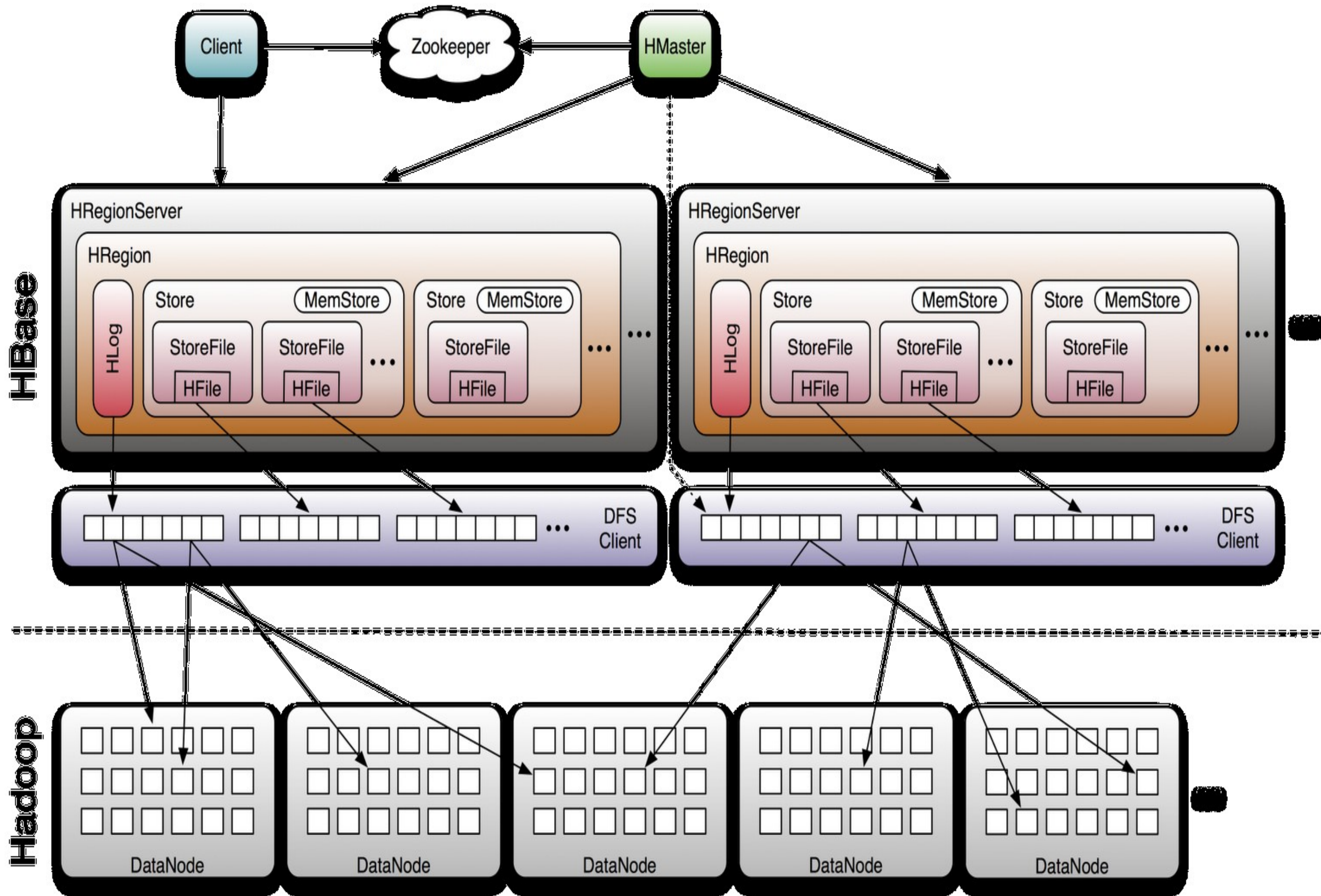
- Distributed storage
- Table-like in data structure
 - multi-dimensional map
- High scalability
- High availability
- High performance

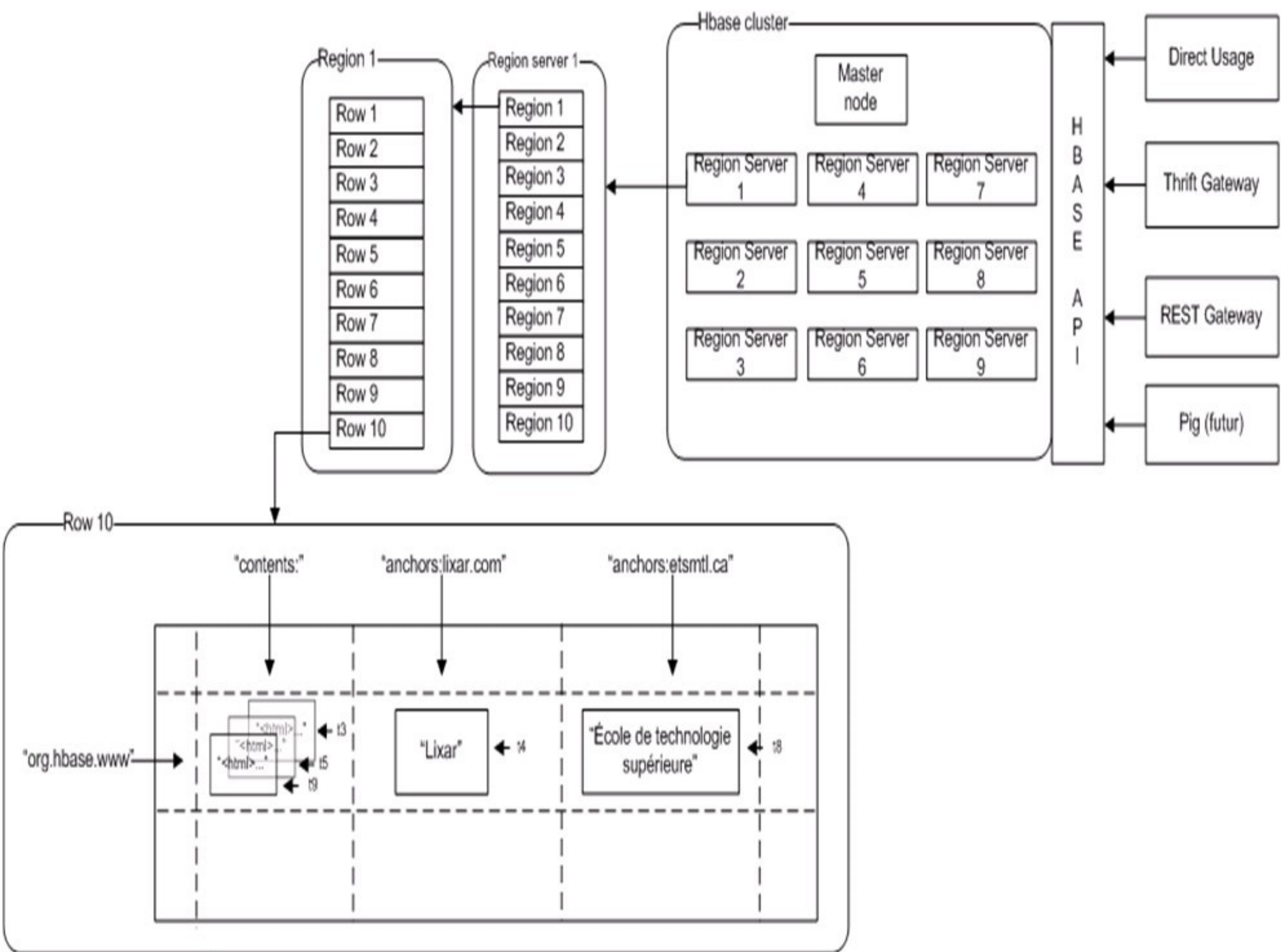
Data Model

- Tables are sorted by Row
- Table schema: *column families*
 - Each family consists of any number of columns
 - Each column consists of any number of versions
 - Columns only exist when inserted, NULLs are free.
 - Columns within a family are sorted and stored together
- Everything except table names are byte[]
- (Row, Family: Column, Timestamp) → Value

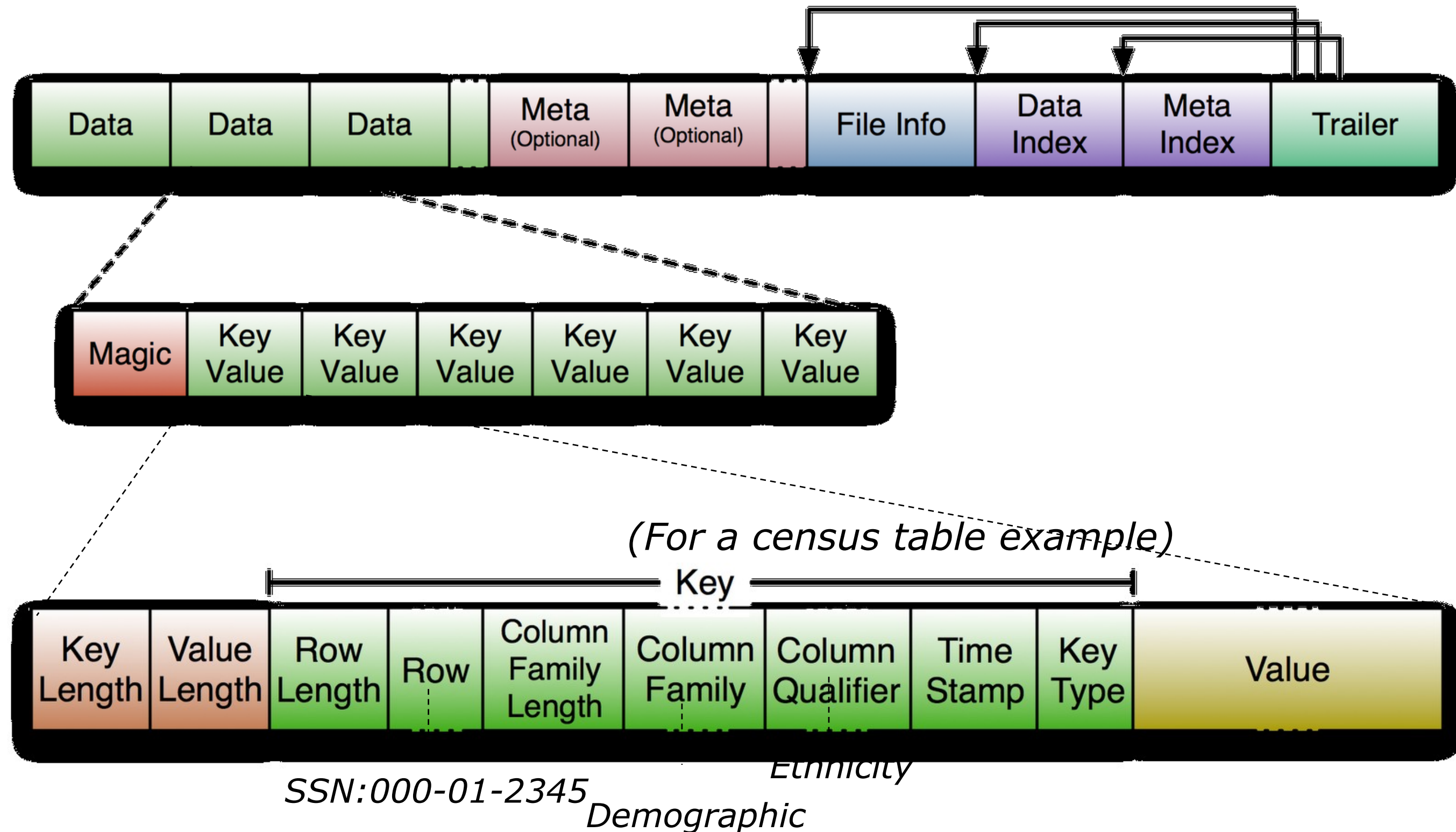


Architecture





HFile

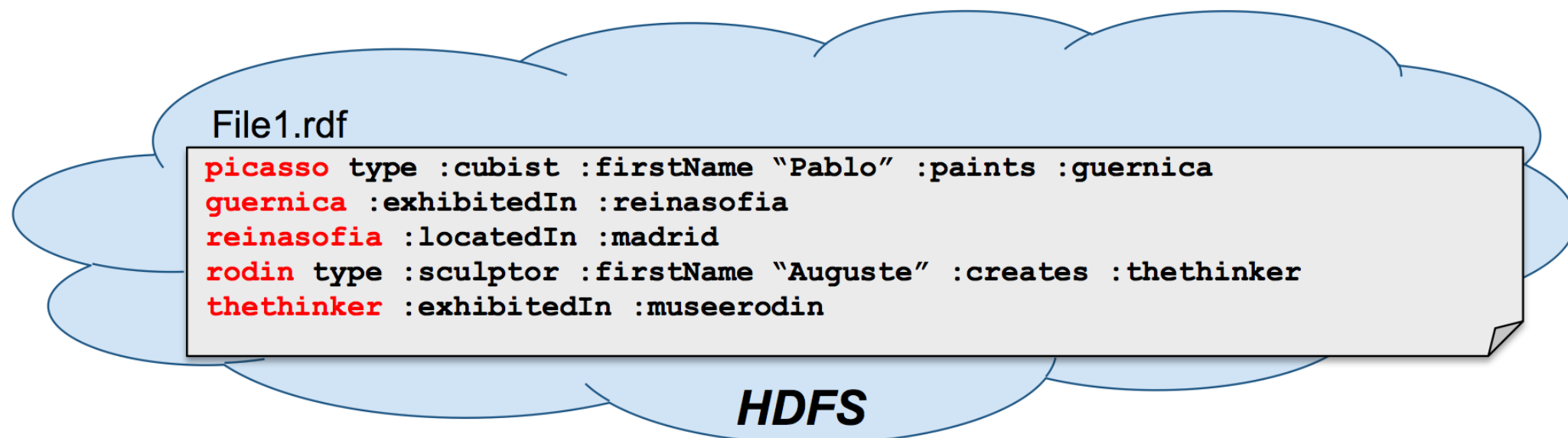


MapReduce-based engines

- ▶ Αποθήκευση RDF δεδομένων σε αρχεία HDFS
- ▶ Υλοποίηση join με χρήση MapReduce
- ▶ Αντιπροσωπευτικά συστήματα
 - SHARD [Rohloff 2010]
 - HadoopRDF [Husain 2011]
 - PigSPARQL [Schätzle 2012]

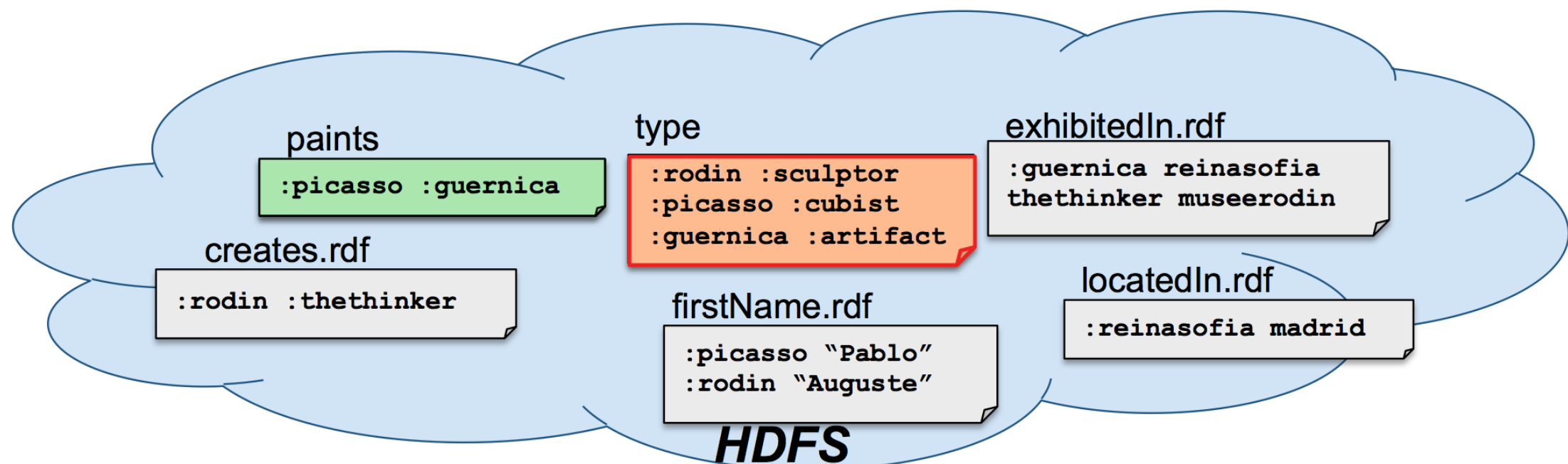
SHARD

- ▶ Αρχεία HDFS:
 - Μια γραμμή περιέχει όλες τις τριάδες που έχουν ένα συγκεκριμένο subject
- ▶ Query processing
 - Left deep join plans
 - Ένα MapReduce job για κάθε BGP του ερωτήματος
 - Μικρές δυνατότητες φιλτραρίσματος των RDF δεδομένων



HadoopRDF

- ▶ Αρχεία HDFS:
 - RDF τριάδες ομαδοποιημένες ανά predicate
 - Εσωτερική ομαδοποίηση των αρχείων με βάση το type του κάθε object
- ▶ Query processing
 - Επιλογή αρχείων που ταιριάζουν σε κάθε BGP
 - Πολλαπλά join ανά MapReduce job
 - Heuristic join planner



Pig SPARQL

- ▶ Αρχεία HDFS:
 - Ένα ενιαίο αρχείο που περιέχει όλα τα RDF δεδομένα
 - Χωρίς δυνατότητες ευρετηρίασης
- ▶ Query processing
 - Μετάφραση SPARQL σε Pig scripts
 - Εκτέλεση των PIG scripts με MapReduce jobs

NoSQL indexing

- ▶ Χρήση key-value store για δημιουργία ευρετηρίων
 - Αριθμός ευρετηρίων 1–6
 - Ανάκτηση δεδομένων για BGP:
 - Με lookups ή range scans
- ▶ Επεξεργασία ερωτημάτων
 - Με τοπική επεξεργασία
 - Με χρήση MapReduce
- ▶ Αντιπροσωπευτικά συστήματα
 - Stratustore [Stein 2010]
 - Rya [Punnoose 2012]
 - H2RDF [Papailiou 2012]
 - H2RDF+ [Papailiou 2013]
 - MAPSIN [Schätzle 2012]

NoSQL indexing

SPO

Key	(attribute, value)
:picasso,:firstName,"Pablo"	-

POS

Key	(attribute, value)
:picasso,:paintedIn,:reinasofia,:guernica	-
:picasso,type,:guernica,:firstName,"Pablo",:picasso,	-

OSP

Key	(attribute, value)
:paints,:guernica	-
type,:cubist	-
...	-
:cubist,:picasso,type,	-
:guernica,:picasso,:paints	-
"Pablo",:picasso,:firstName	-
:reinasofia,:guernica,:exhibitedIn	-
...	-

NoSQL indexing

- ▶ Stratustore (Amazon SimpleDB)
 - S|P|O 1 hash index
- ▶ Rya (Apache Accumulo)
 - SPO|-, POS|-, OSP|- 3 sorted indexes
- ▶ H2RDF (Apache HBase)
 - SPO|-, POS|-, OSP|- 3 sorted indexes
- ▶ H2RDF+ (Apache HBase)
 - SPO|-, SOP|-, POS|-, PSO|-, OPS|-, OSP|- 6 sorted indexes
- ▶ MAPSIN (Apache HBase)
 - SPO|, OPS|- 2 sorted indexes

Rya

- ▶ Αποθήκευση δεδομένων
 - Apache Accumulo
 - 3 sorted indexes
 - Ανάκτηση δεδομένων με lookup για όλα τα BGP patterns
- ▶ Επεξεργασία ερωτημάτων
 - Index nested loops
 - Αναζήτηση του accumulo index για κάθε κλειδί του join
 - Αποδοτικό για μικρού μεγέθους join
 - Απαιτεί πολλά index lookups για non selective ερωτήματα

H2RDF

- ▶ Αποθήκευση δεδομένων
 - Apache HBase
 - 3 sorted indexes
 - Ανάκτηση δεδομένων με lookup για όλα τα BGP patterns
- ▶ Επεξεργασία ερωτημάτων
 - Partial Input Hash joins
 - Διαλέγει τον κατάλληλο join αλγόριθμο ανάλογα με το μέγεθος δεδομένων των BGP
 - Αν υπάρχει μικρό pattern μόνο αυτό χρησιμοποιείται ως είσοδος
 - Κατανεμημένη (MapReduce) ή κεντρική εκτέλεση των join

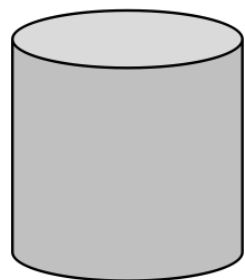
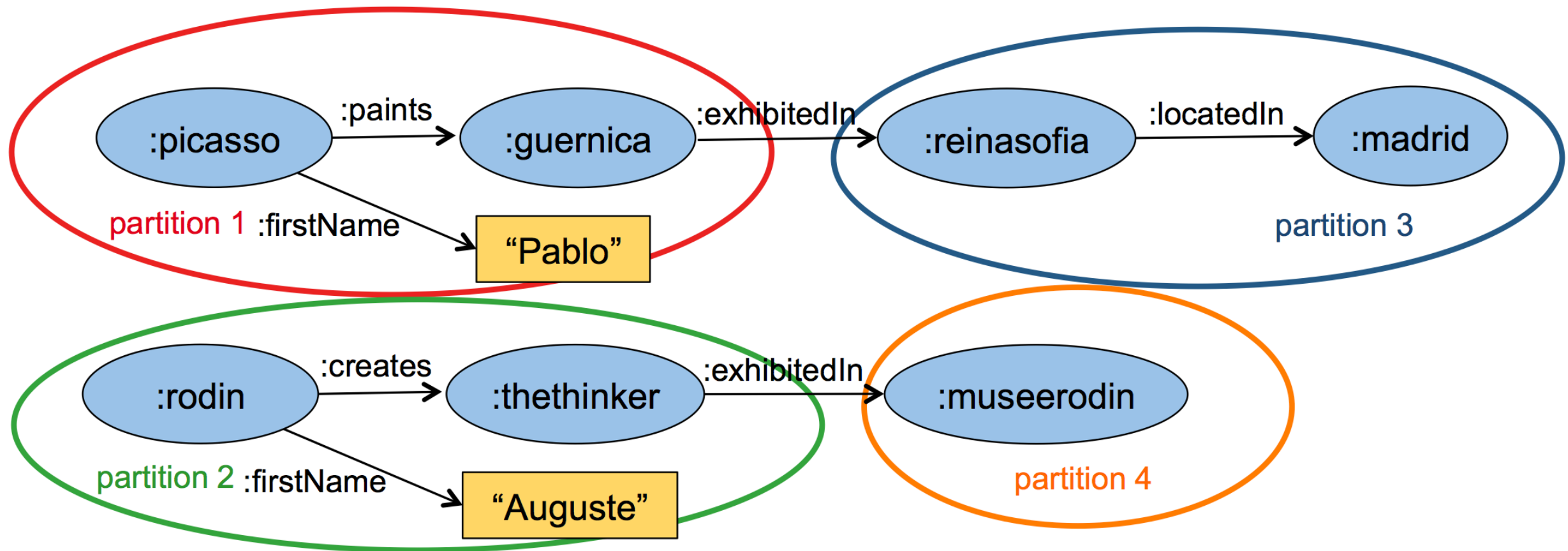
H2RDF+

- ▶ Αποθήκευση δεδομένων
 - Apache HBase
 - 6 sorted indexes και aggregated statistics
 - Συμπίεση των ευρετηρίων
 - Ανάκτηση ταξινομημένων δεδομένων για όλα τα BGP patterns
- ▶ Επεξεργασία ερωτημάτων
 - Αξιοποιεί τα 6 index για την εκτέλεση Merge join
 - Multi-way Merge και Sort-Merge joins
 - Κατανεμημένη (MapReduce) ή κεντρική εκτέλεση των join
 - Μοντέλο κόστους για τα joins
 - Ελαστική επιλογή των resources

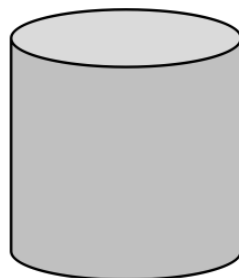
MAPSIN

- ▶ Αποθήκευση δεδομένων
 - Apache HBase
 - 2 sorted indexes
 - Ανάκτηση δεδομένων για τα BGP patterns με γνωστό predicate
- ▶ Επεξεργασία ερωτημάτων
 - MapReduce map phase join algorithm
 - Lookup operations για κάθε κλειδί του join

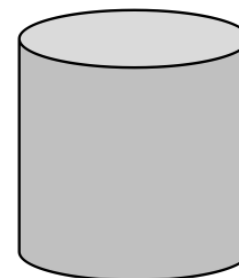
Graph partitioning



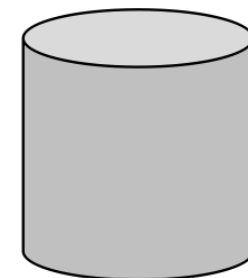
node 1



node 2



node 3

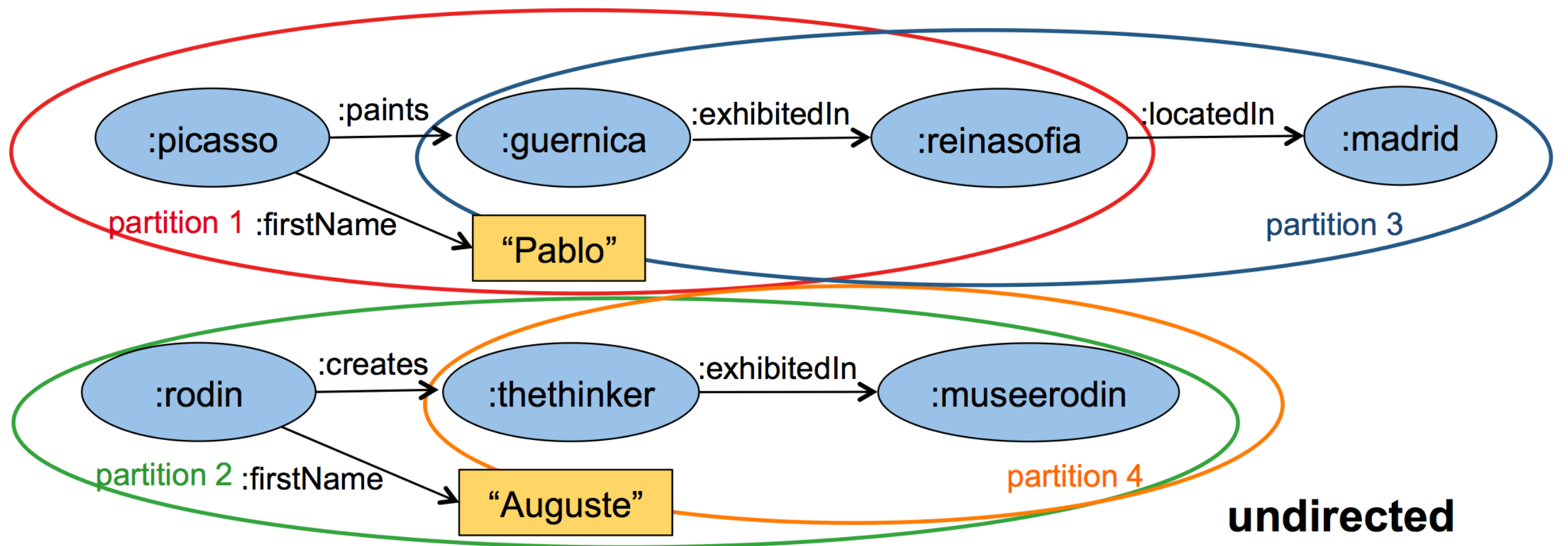


node 4

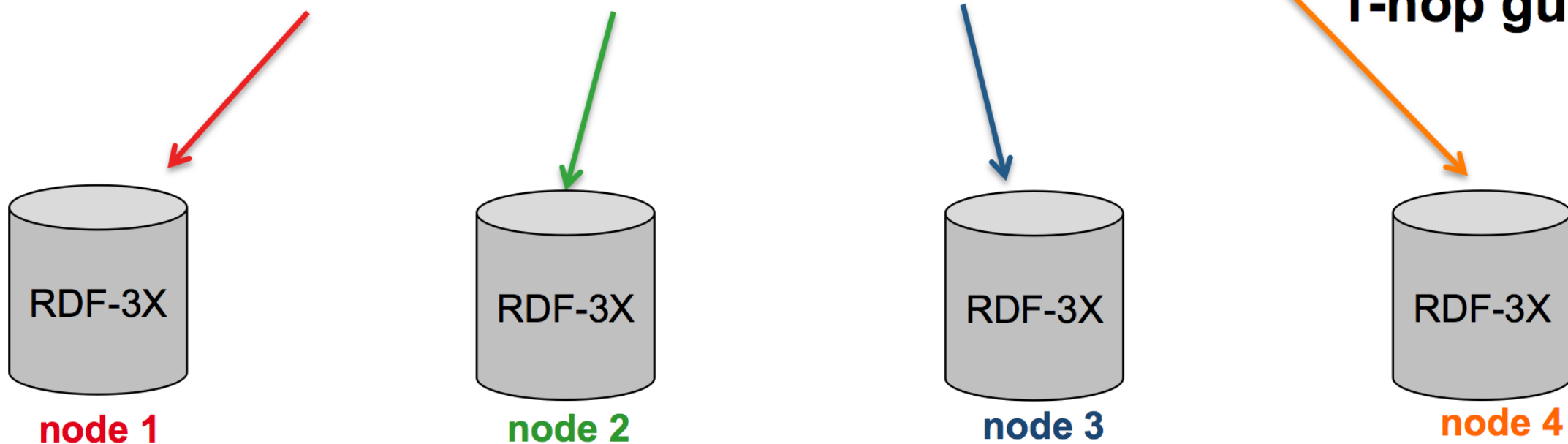
Graph partitioning

- ▶ Graph partitioning [Huang et al. 11]
 - Graph partitioning για διαμερισμό των RDF δεδομένων
 - Κάθε κόμβος χρησιμοποιεί ένα RDF-3X για τα δεδομένα του δικού του graph partition
 - n-hop replication scheme
 - Εκτός από τα δεδομένα του partition του ο κάθε κόμβος κάνει replicate και δεδομένα που είναι έως n βήματα μακριά
 - Παράλληλη εκτέλεση για ερωτήματα με διάμετρο μικρότερη του n
 - Τα μεγαλύτερα ερωτήματα χωρίζονται σε μικρότερα, διαμέτρου n
 - Τα αποτελέσματα των υποερωτημάτων συνδυάζονται με χρήση MapReduce

1-hop guarantee



**undirected
1-hop guarantee**

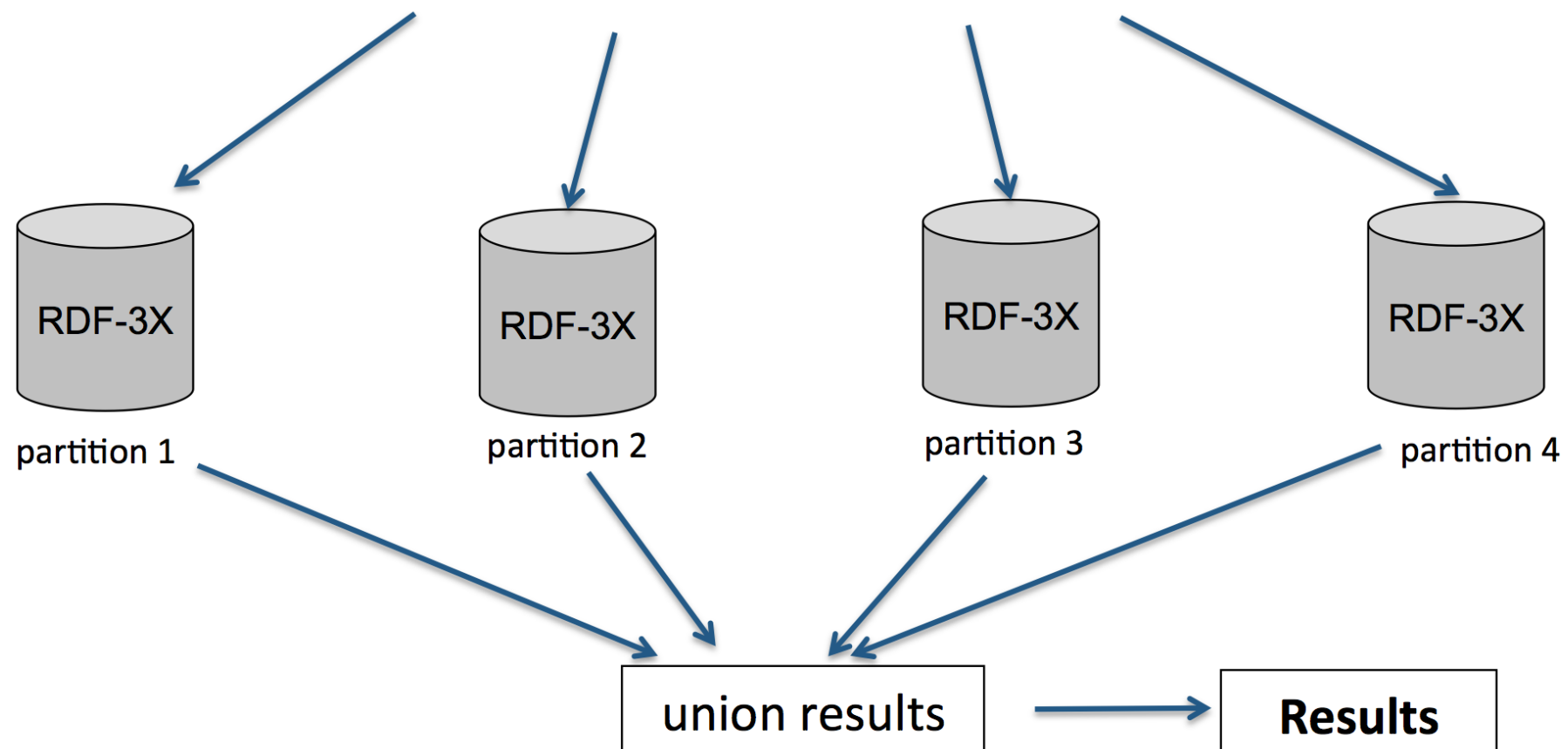


Graph partitioning

replication with
1-hop guarantee

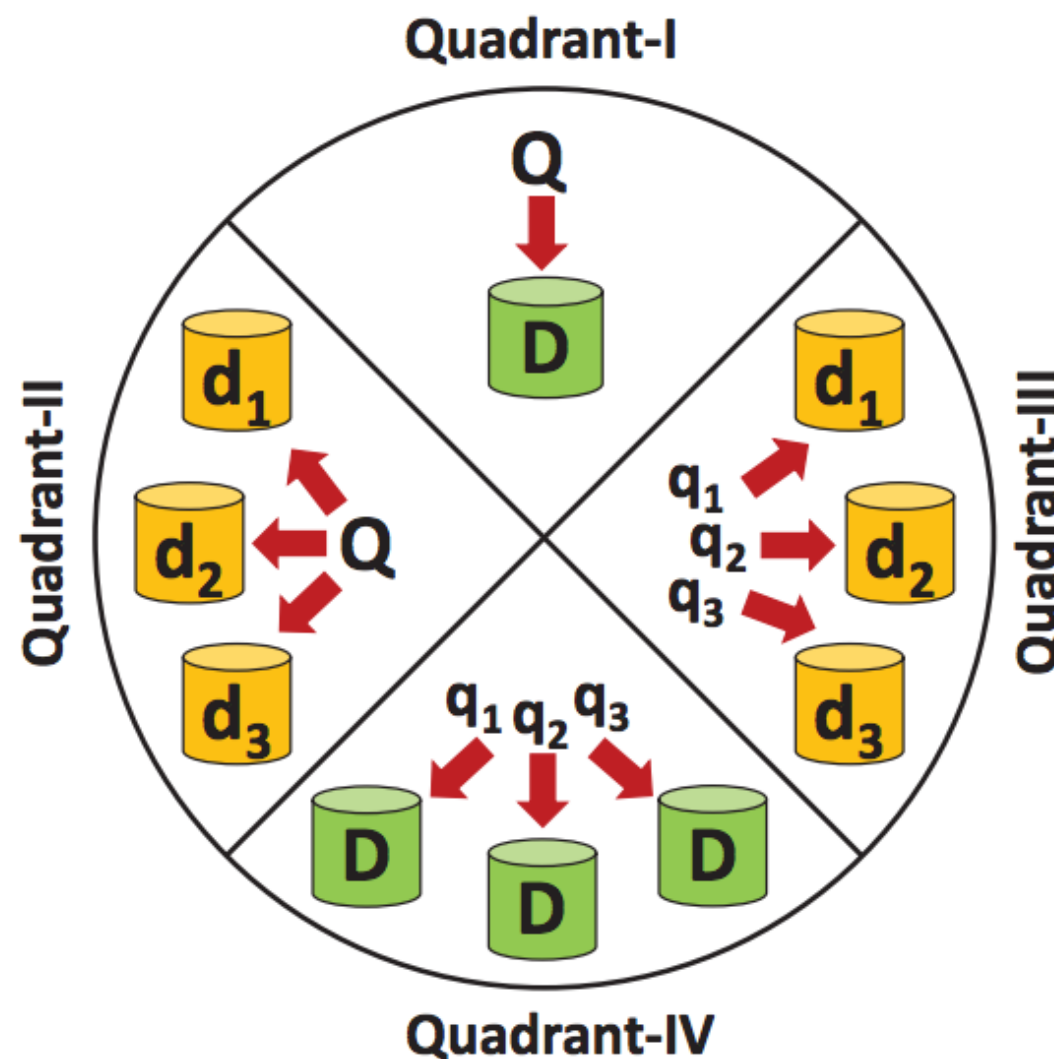
PWOC query

```
SELECT ?x ?y ?z
WHERE {
  ?x type :artist .
  ?x :firstName ?y .
  ?x :creates ?z .}
```



DREAM

- ▶ DREAM [Hammoud 2015]
 - Διαχωρισμός συστημάτων ανάλογα με την κατανομή των δεδομένων και της επεξεργασίας
 - Υλοποίηση συστήματος που ανήκει στο Quadrant-IV



Distributed Main Memory

- ▶ Memory cloud: TrinityRDF [Zeng et al. 2013]
 - Αποθηκεύει τα RDF δεδομένα στην κύρια μνήμη όλων των κόμβων του cluster
 - Κάθε κόμβος κρατάει ένα memory hash-map των RDF δεδομένων που του αντιστοιχούν
 - Query execution
 - Graph exploration με μηνύματα μεταξύ των κόμβων
 - Ουσιαστικά κάνει ένα semi-join processing για το query
 - Δεν κάνει πλήρες reduction για ερωτήματα με κύκλους
 - Στο τέλος τα reduced αποτελέσματα μαζεύονται σε ένα κεντρικό server που κάνει το τελικό processing

Distributed Main Memory

- ▶ TriAD [Gurajada 2014]
 - Hash partitioning
 - Αποθηκεύει τα RDF δεδομένα στην κύρια μνήμη όλων των κόμβων
 - 6 indexes + aggregated statistics
 - MPI-based asynchronous join execution
 - Βασισμένο στο RDF-3X για join planning

Ερωτήσεις

