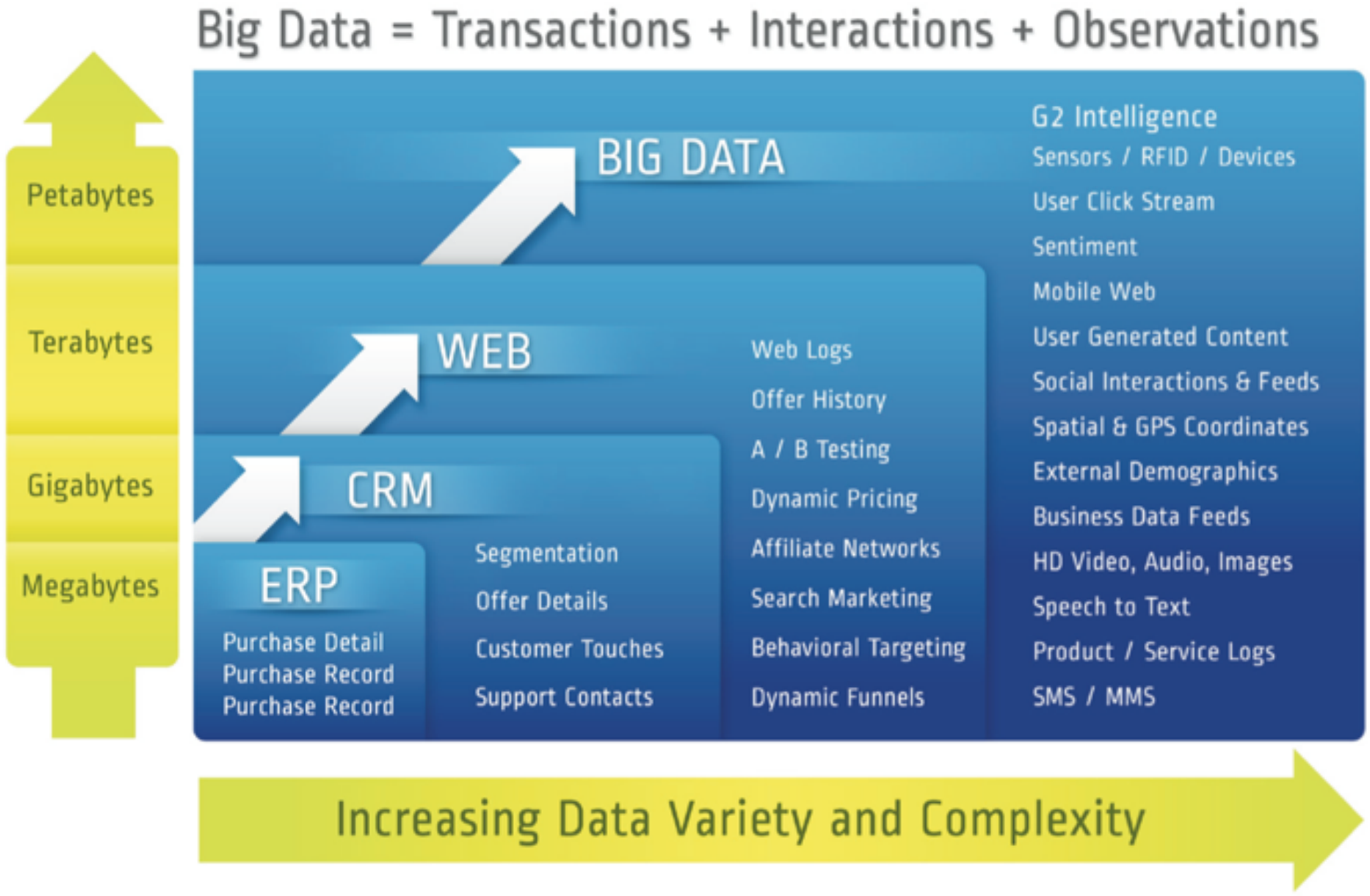
Introduction to MapReduce

Big Data

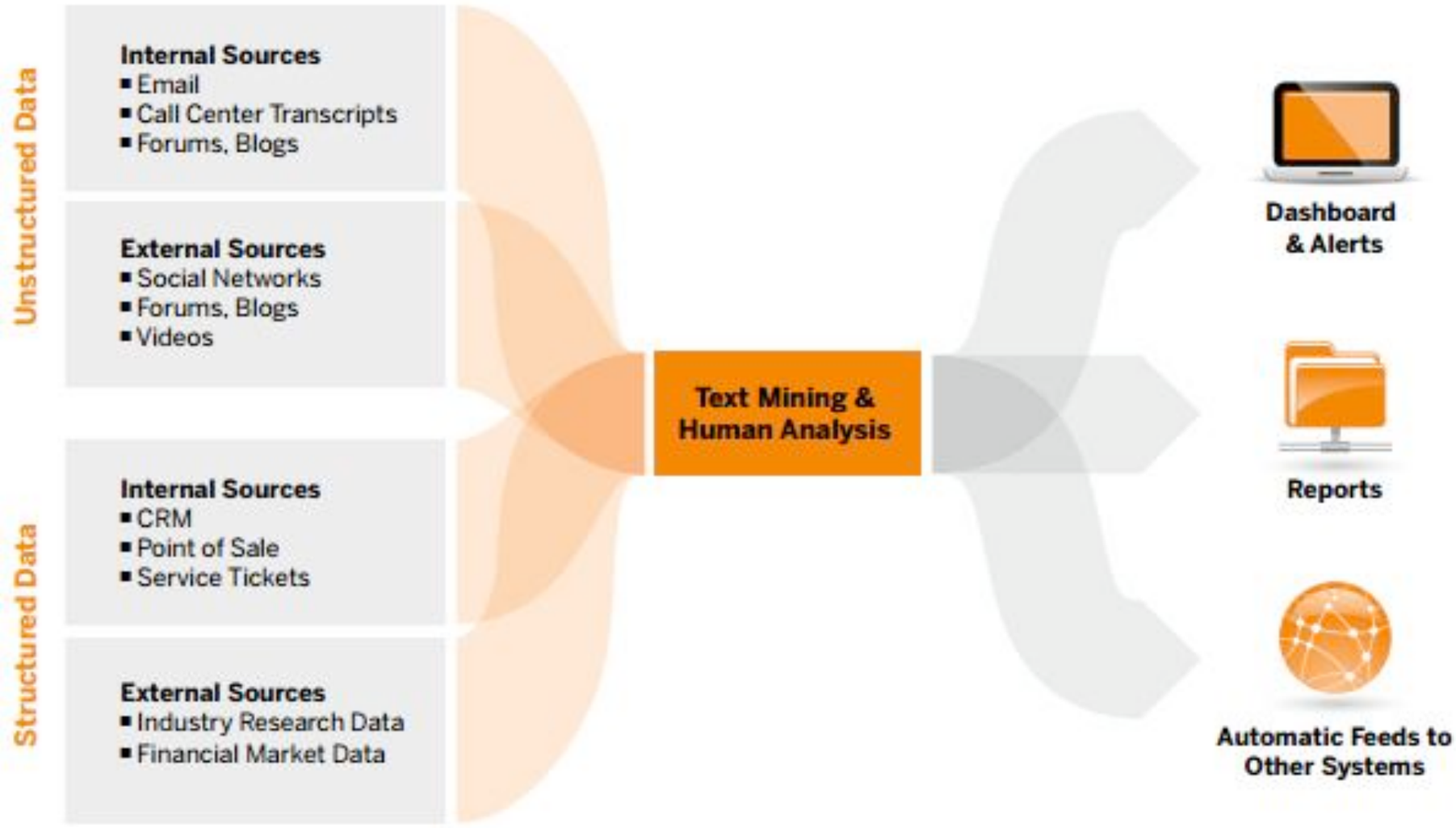
What is **"Big Data"** and what does it mean?

- Other than a buzzword?
- Two things
    - Data so large it cannot be managed by a single computer (scale out vs. scale up)
- Tackles new types of data
    - **Structured Data** = High degree of organization, readily searchable and fits into relational tables
    - **Unstructured Data** = No pre-defined data model and/or does not fit into relational tables

Big Data


Big Data = Transactions + Interactions + Observations
Petabytes
Terabytes
Gigabytes
Megabytes
BIG DATA
WEB
CRM
ERP
Purchase Detail
Purchase Record
Purchase Record
Web Logs
Offer History
A / B Testing
Dynamic Pricing
Affiliate Networks
Search Marketing
Behavioral Targeting
Dynamic Funnels
Segmentation
Offer Details
Customer Touches
Support Contacts
G2 Intelligence
Sensors / RFID / Devices
User Click Stream
Sentiment
Mobile Web
User Generated Content
Social Interactions & Feeds
Spatial & GPS Coordinates
External Demographics
Business Data Feeds
HD Video, Audio, Images
Speech to Text
Product / Service Logs
SMS / MMS
Increasing Data Variety and Complexity

Structured vs. Unstructured



Big Data

What is **"Big Data"** and what does it mean?

- Other than a buzzword?
- Two things
    - Data so large it cannot be managed by a single computer (scale out vs. scale up)
- Tackles new types of data
    - **Structured Data** = High degree of organization, readily searchable and fits into relational tables
    - **Unstructured Data** = No pre-defined data model and/or does not fit into relational tables

New Problems = New Tools

Thought Experiment

1TB Hard Drive
Average transfer speed: 100 MB/s

Let's say your dataset is 1TB in size. How long would it take to read in the entire dataset using a single 1TB hard drive?

1TB of data / 100MB/s = 2.91 *hours*

Thought Experiment

1TB Hard Drive
Average transfer speed: 100 MB/s

How about with 10 1TB hard drives?

Aggregate transfer speed: 1000 MB/s

1TB of data / 1,000MB/s = **17 minutes**

With 100 1TB hard drives?

Aggregate transfer speed: 10,000 MB/s

1TB of data / 10,000MB/s = **105 seconds**

With 1,000 1TB hard drives?

Aggregate transfer speed: 100,000 MB/s

1TB of data / 100,000MB/s = **10 seconds**

Thought Experiment

Sounds like a lot!

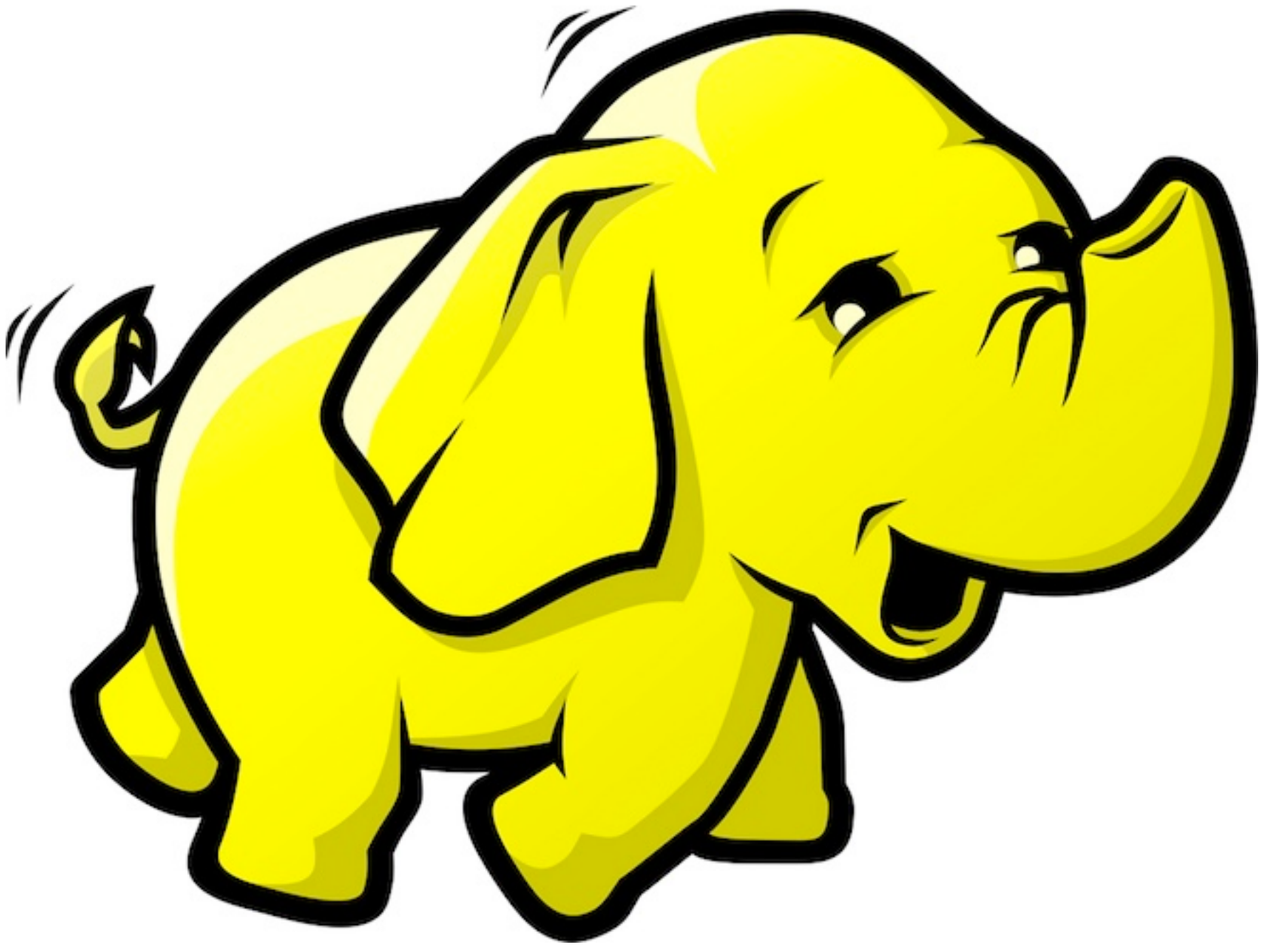
- But really it's not!
- Typical Hadoop nodes have 24 x 1TB drives in 2U
- 1,000 HDDs / 24 = 42 nodes = 84U = 2 racks!

Thought Experiment

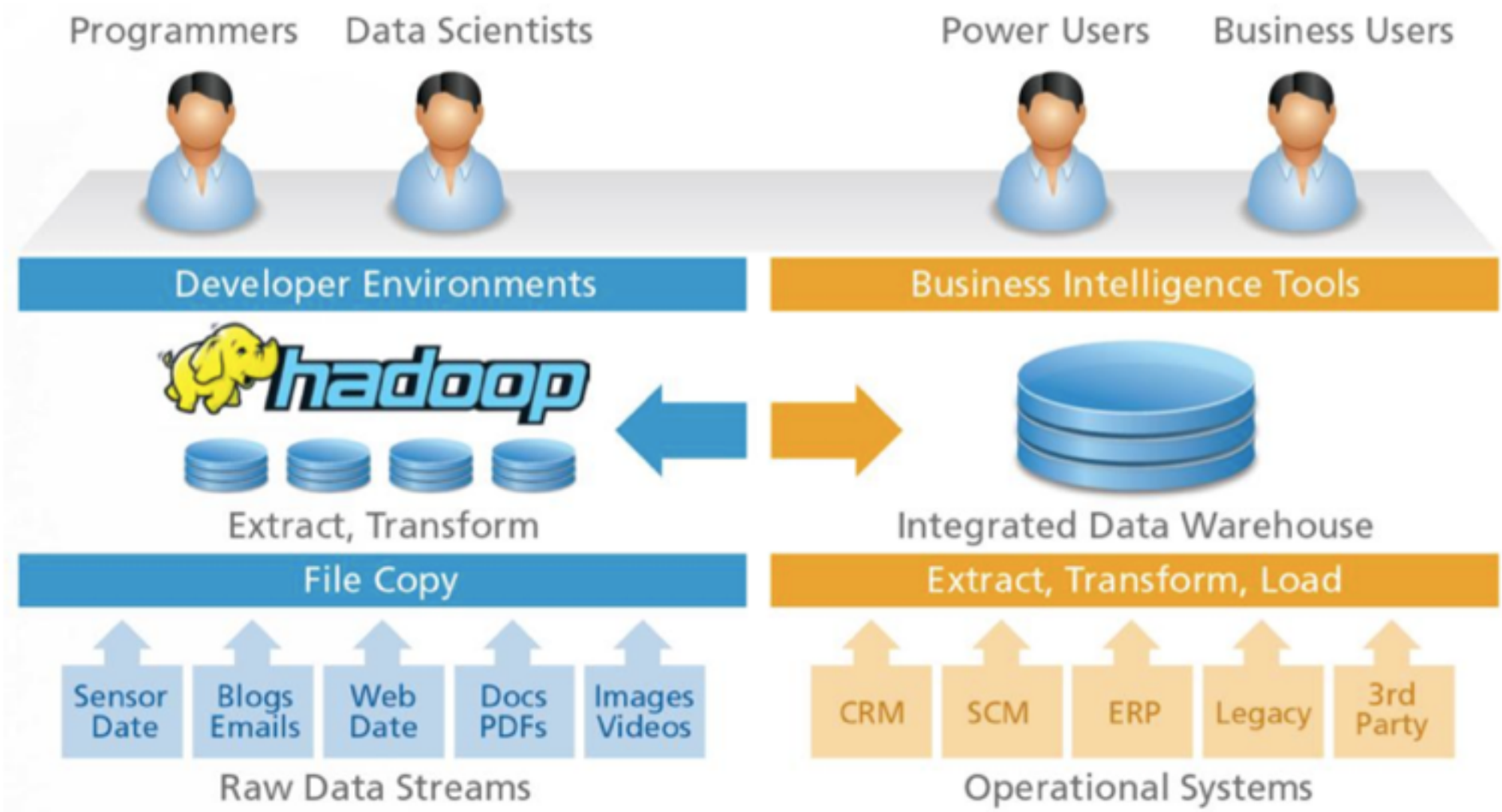
Sounds like a lot!

- But really it's not!
- Typical Hadoop nodes have 24 x 1TB drives in 2U
- 1,000 HDDs / 24 = 42 nodes = 84U = 2 racks!

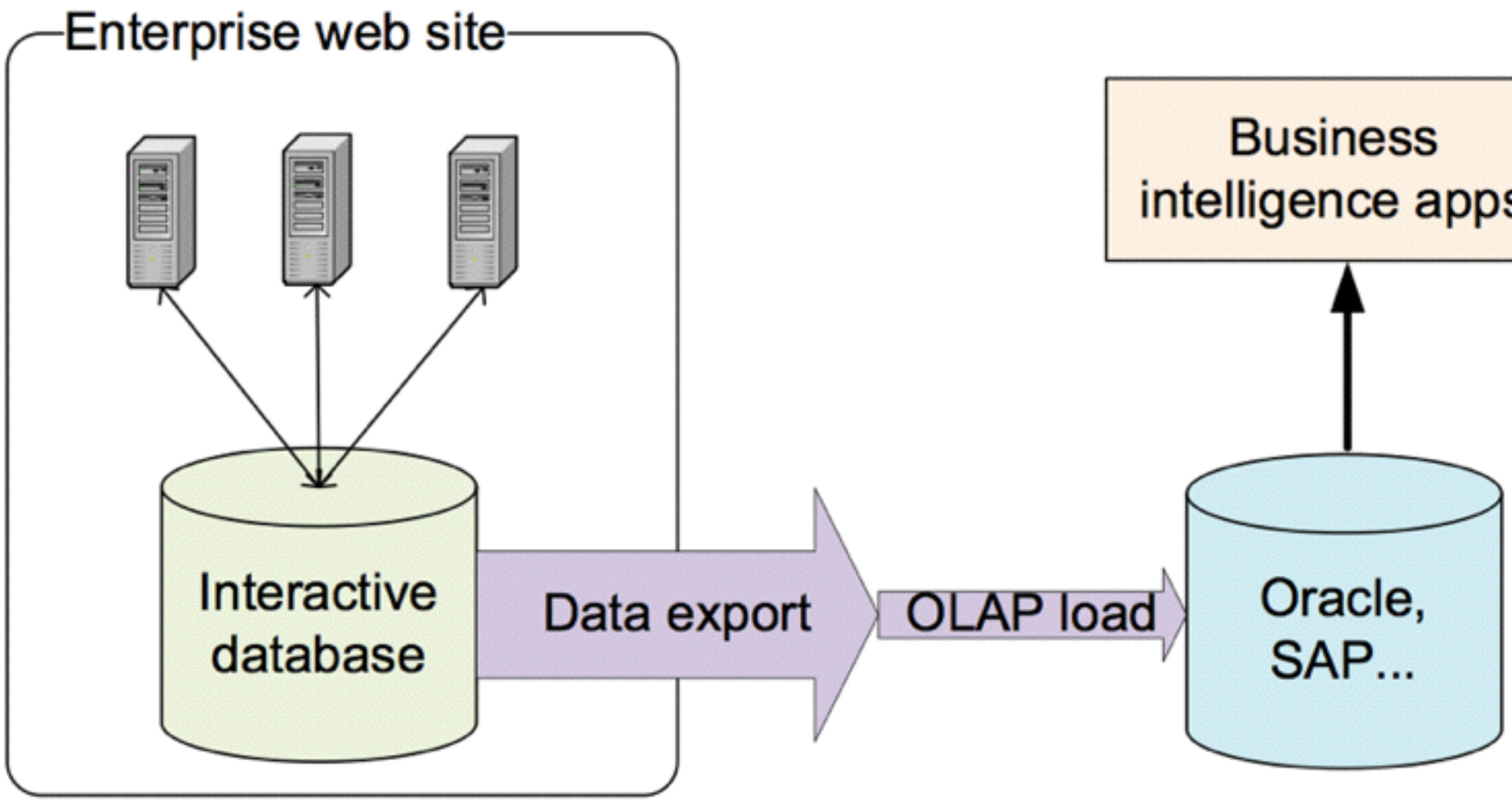
We need this little guy



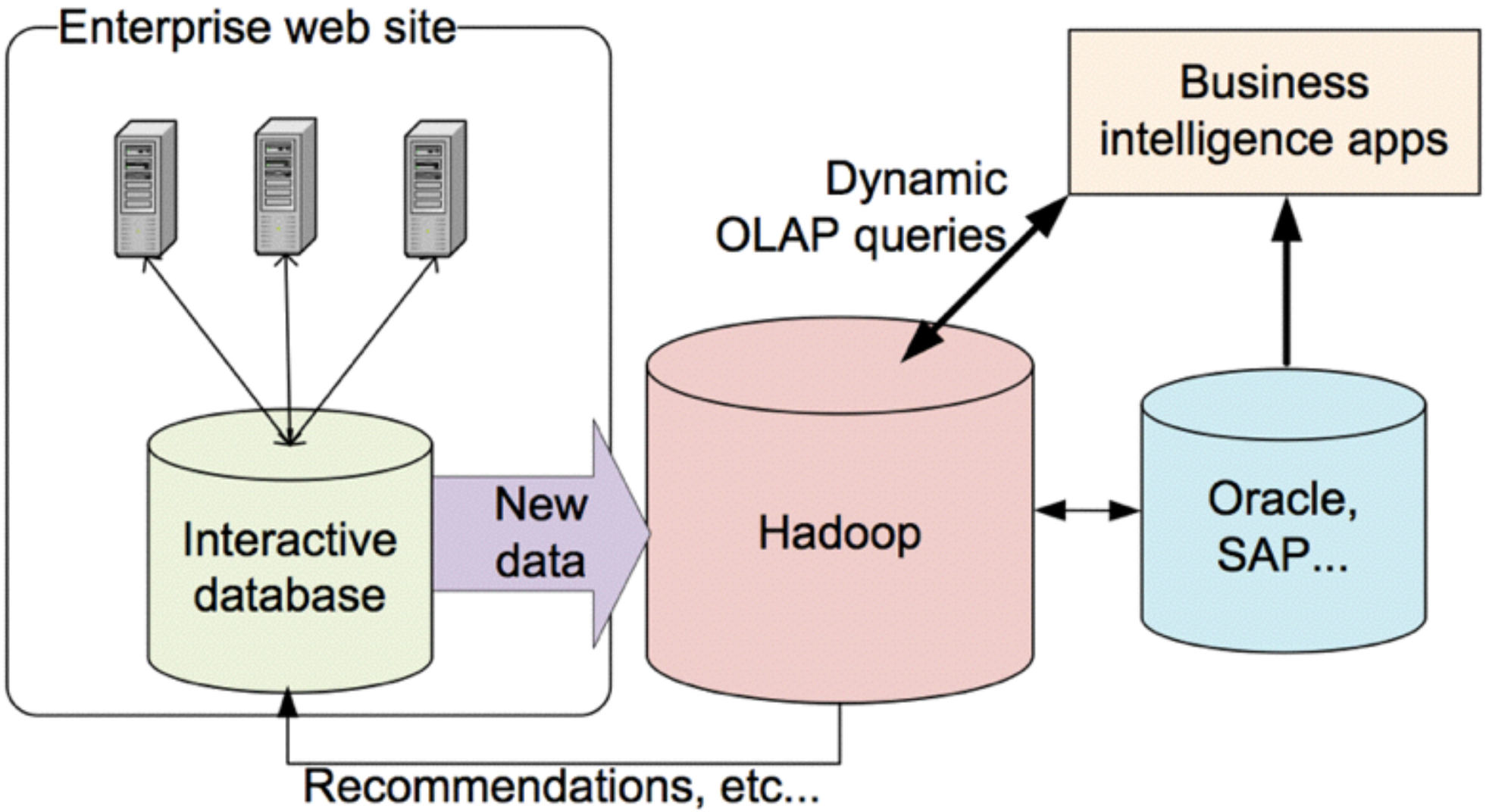
Data Sources



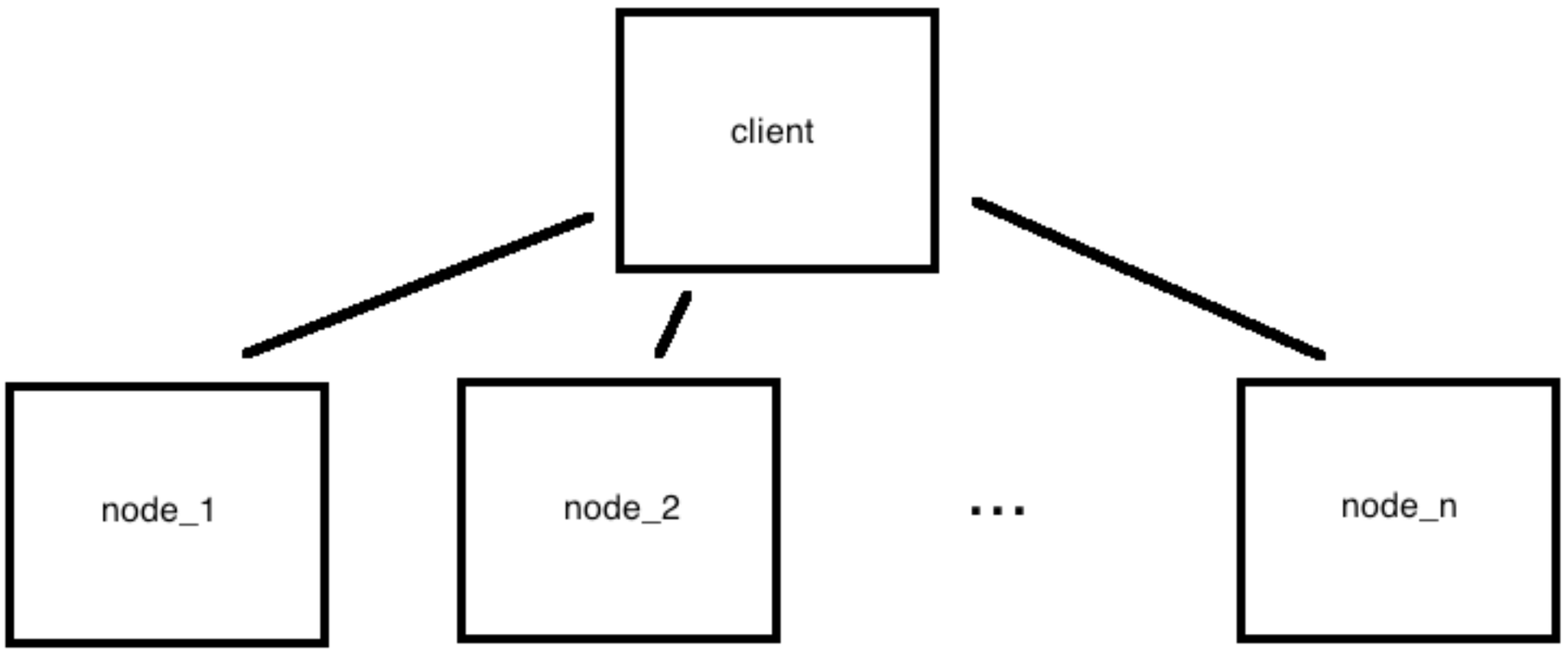
Data Sources



Data Sources



We can visualize this horizontal cluster architecture as a single client-multiple server relationship



If you're working in a distributed system, you need to solve the following problem:
How do I do process the data?

There are two options:

1. **Move the data to the code** (and processing power)
2. **Move the code to the data** (MapReduce)

Divide & Conquer

1. Split task into many subtasks
2. Solve these tasks independently
3. Recombine the subtasks into a final result

Divide & Conquer

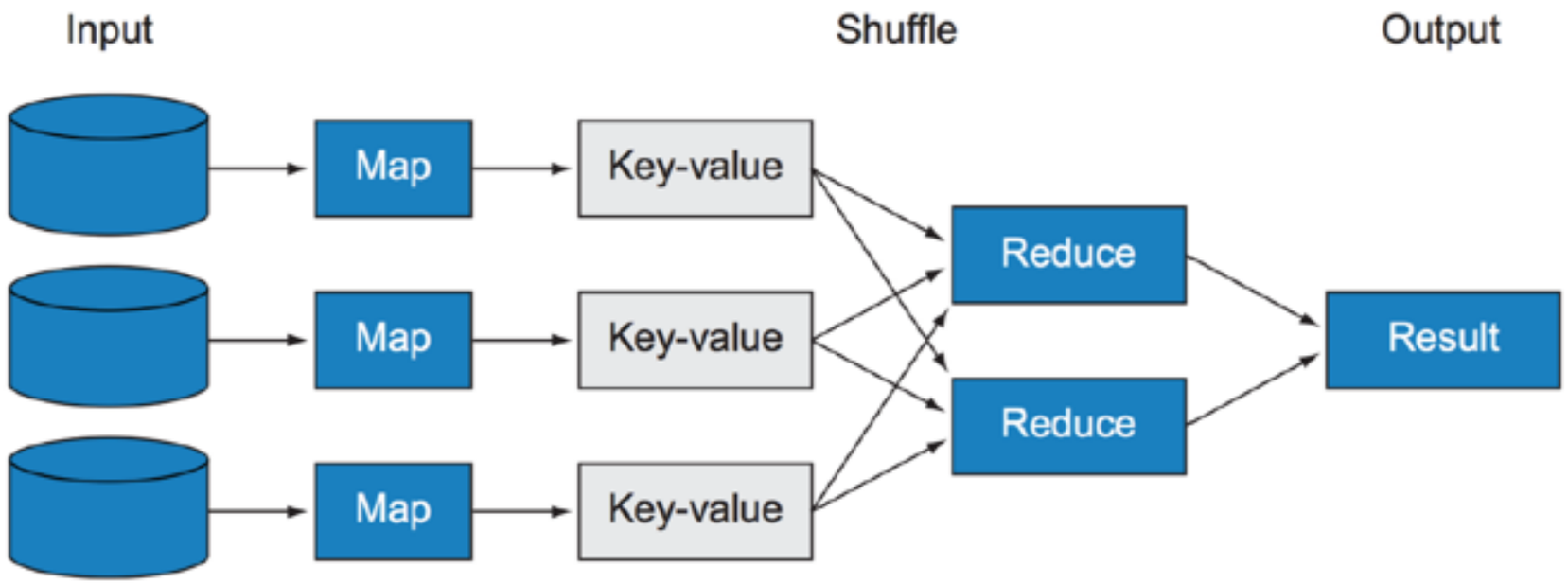
Which tasks are suitable for this approach?
Tasks that can be broken down into independent subtasks, such as:

count, sum, avg
grep, sort, graph traversals
some* ML algorithms (which ones?)

- Recall that map-reduce splits a problem into subtasks to be processed in parallel

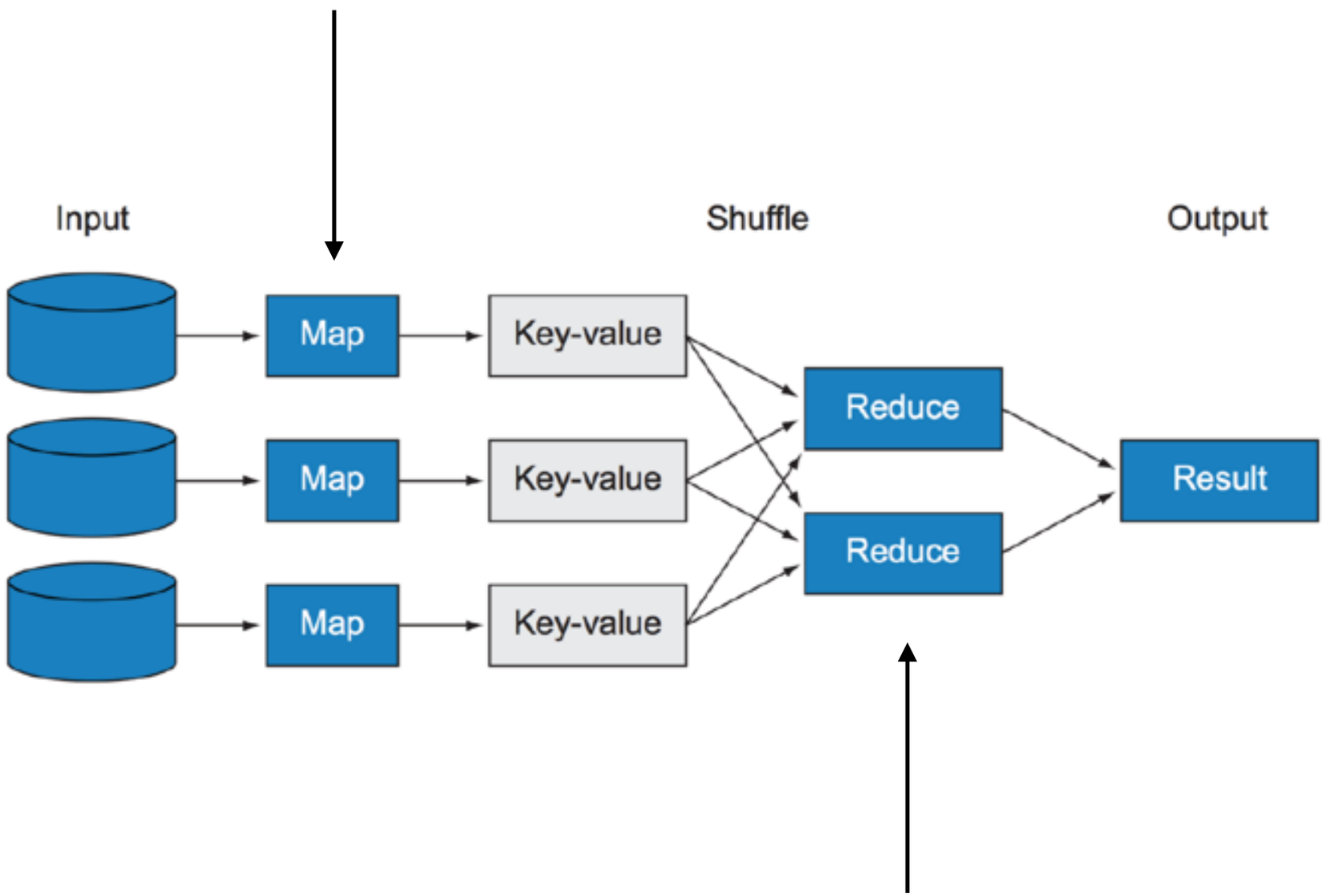
- This happens in two places:
  - The **reducer** phase
  - The **mapper** phase

MapReduce


Input
Shuffle
Output
Map
Map
Map
Key-value
Key-value
Key-value
Reduce
Reduce
Result

MapReduce

filter & transform data



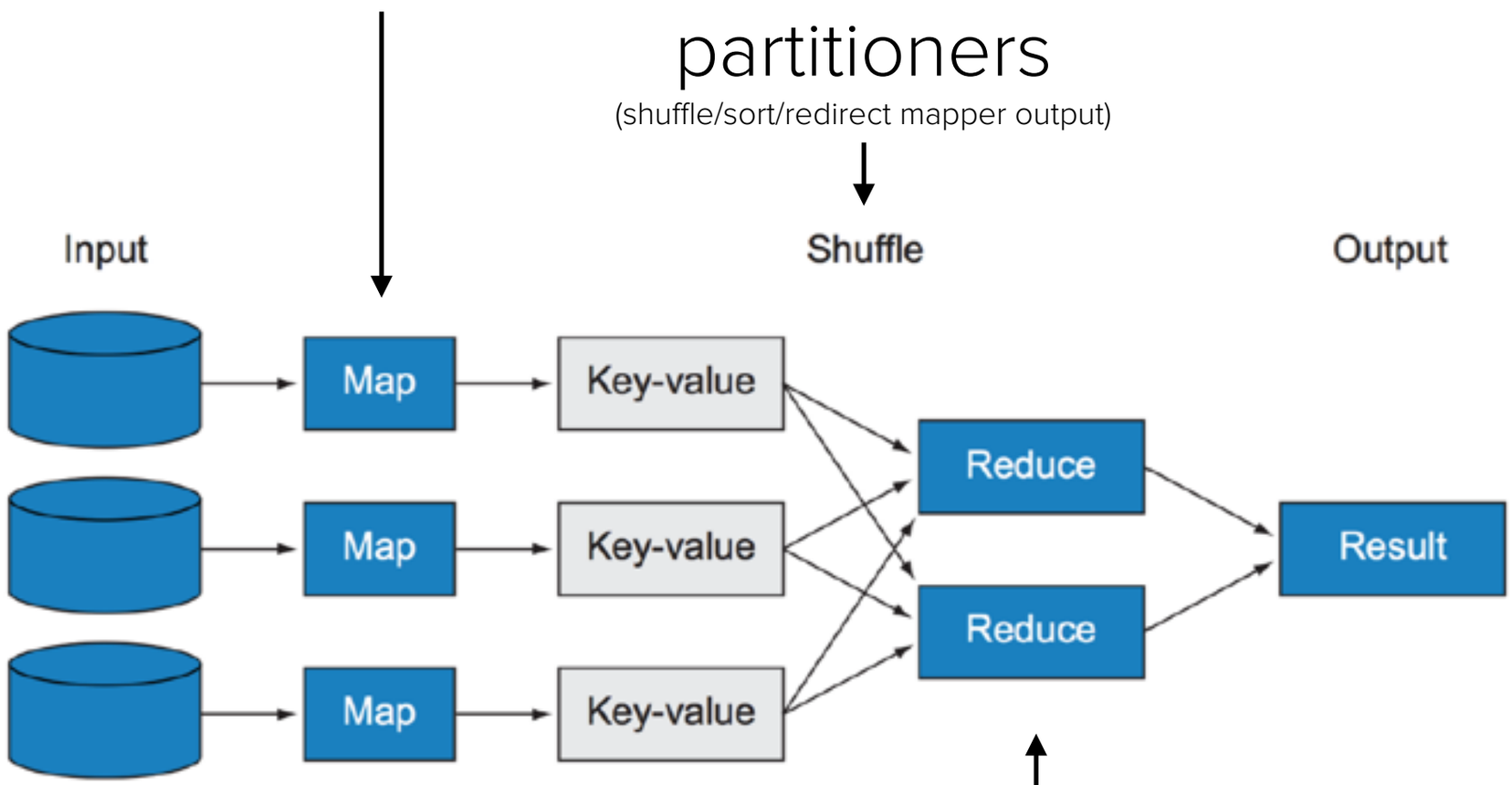
aggregate results

MapReduce

filter & transform data

partitioners

(shuffle/sort/redirect mapper output)



combiners

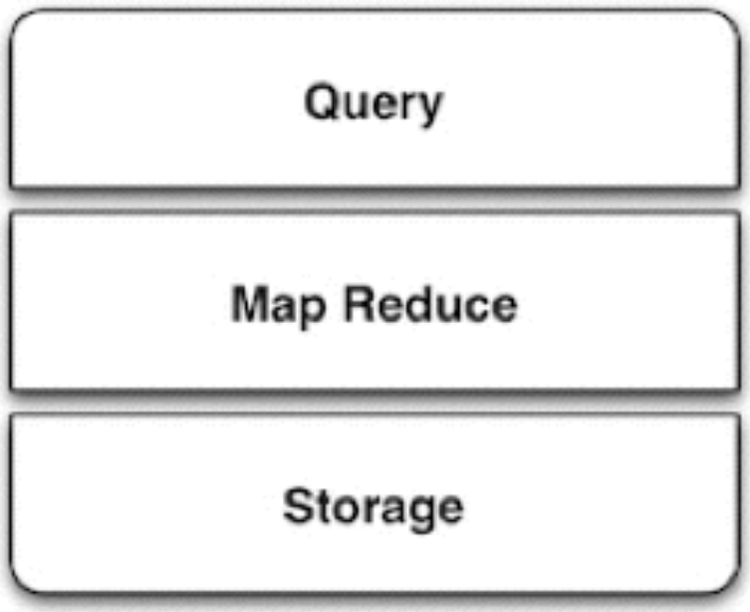
(perform reducer step on intermediate data)

aggregate results

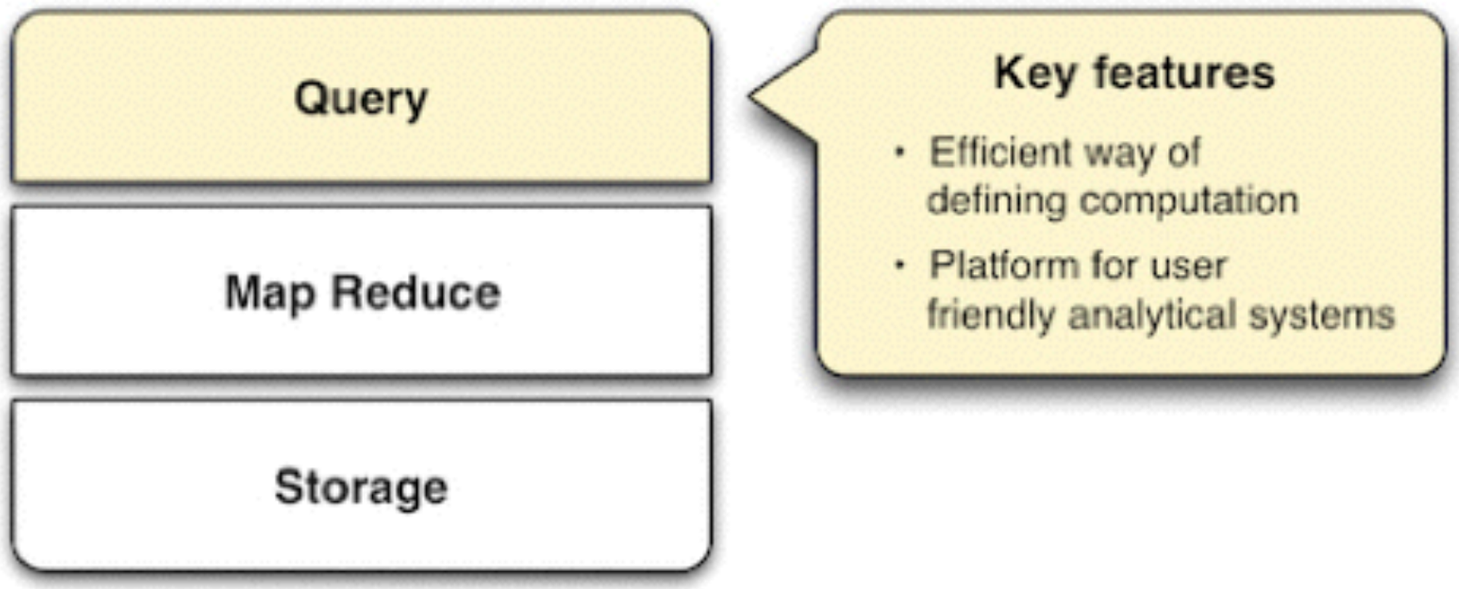
Map-reduce handles a lot of the messy details and does most of the heavy lifting around:

1. parallelization & distribution (input splits)
2. partitioning (shuffle/sort)
3. fault-tolerance
4. scheduling & resource Management
5. status & monitoring

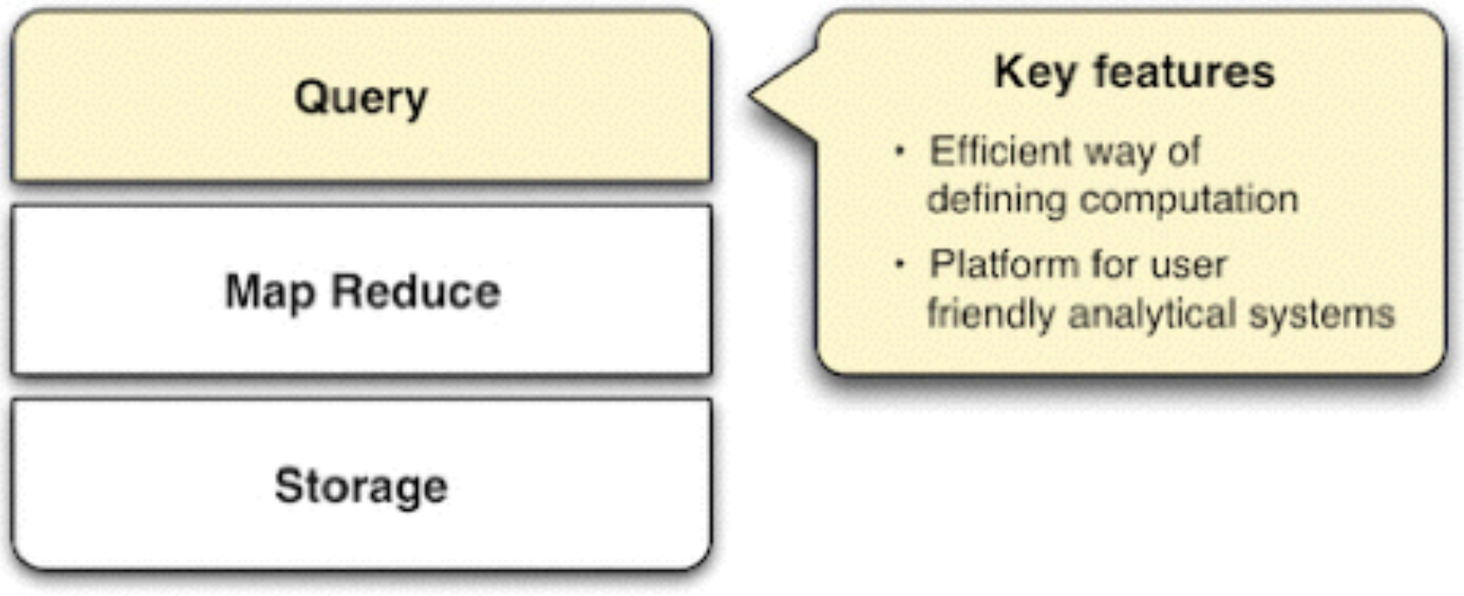
Hadoop Components



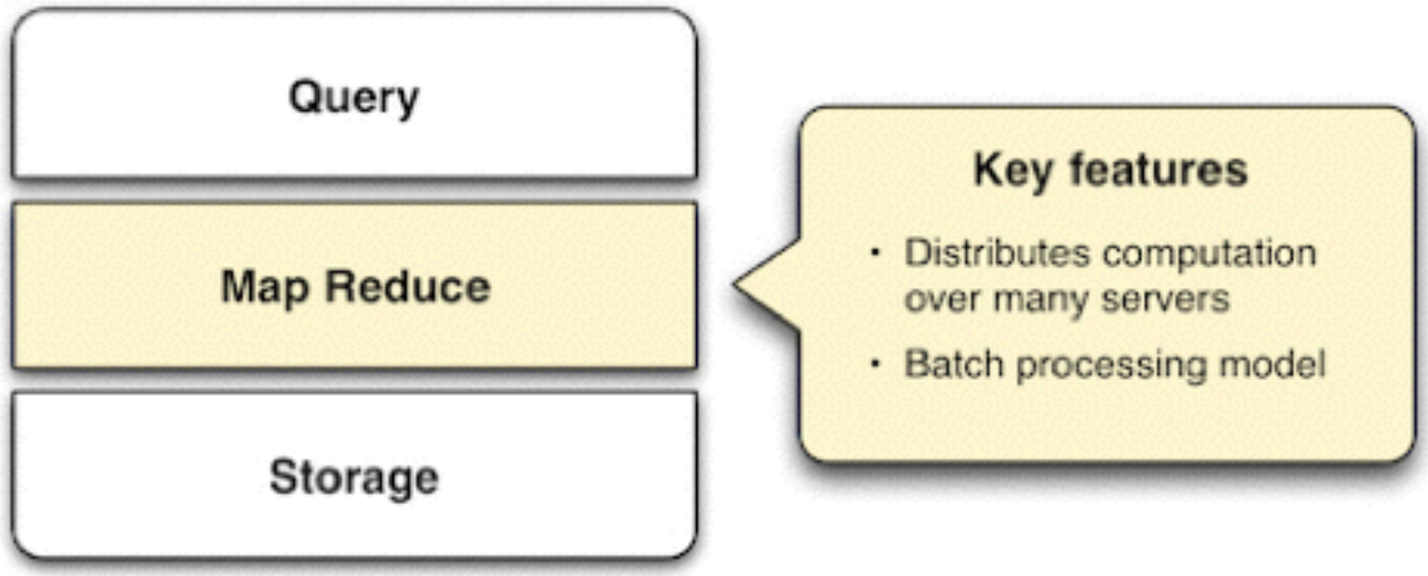
Hadoop Components



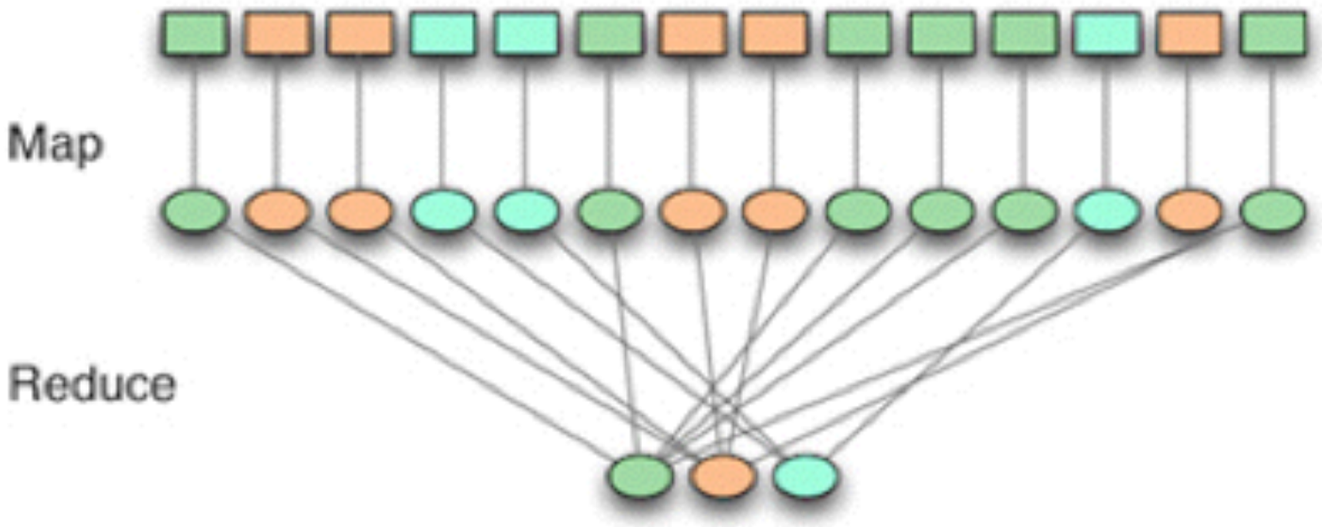
Hadoop Components

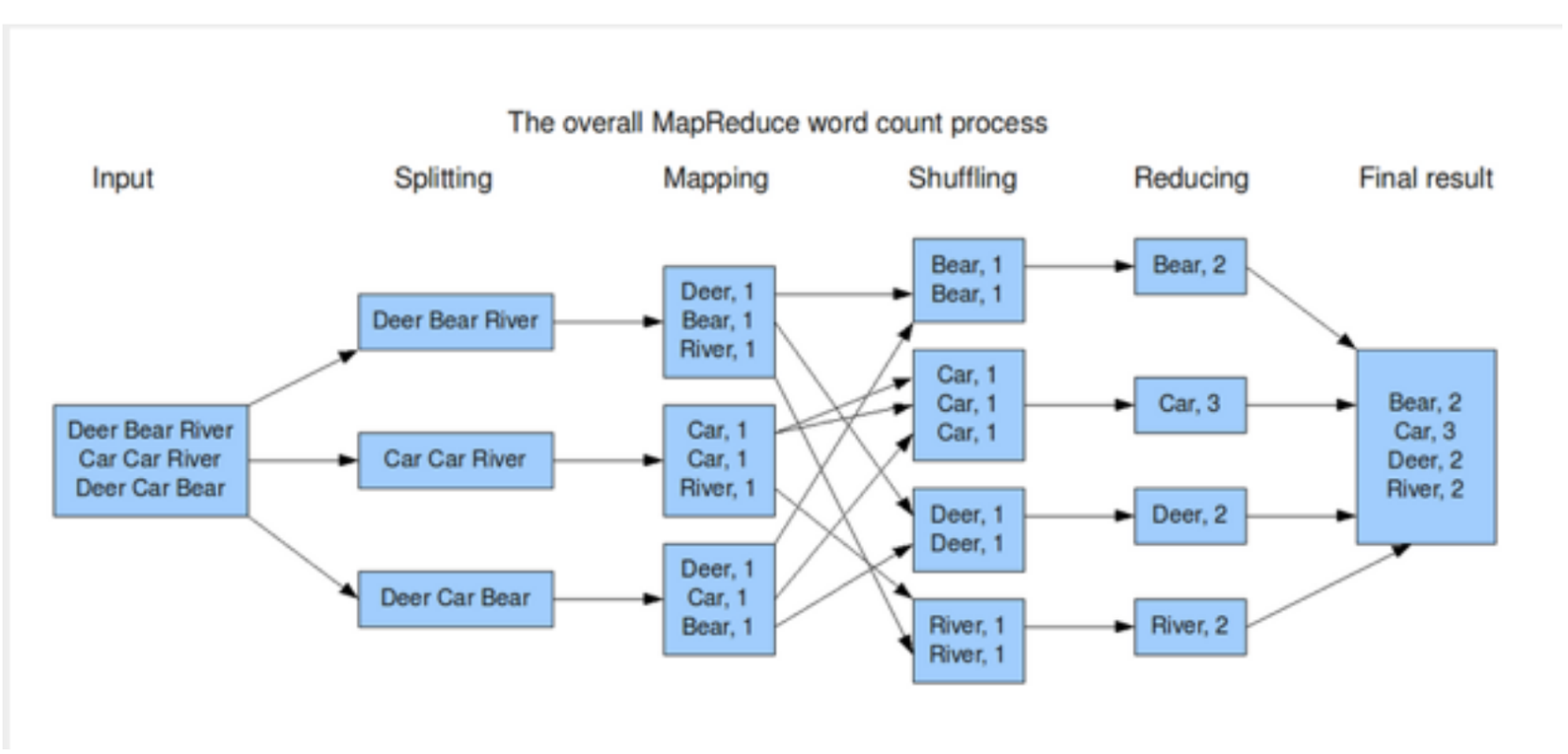


Hadoop Components



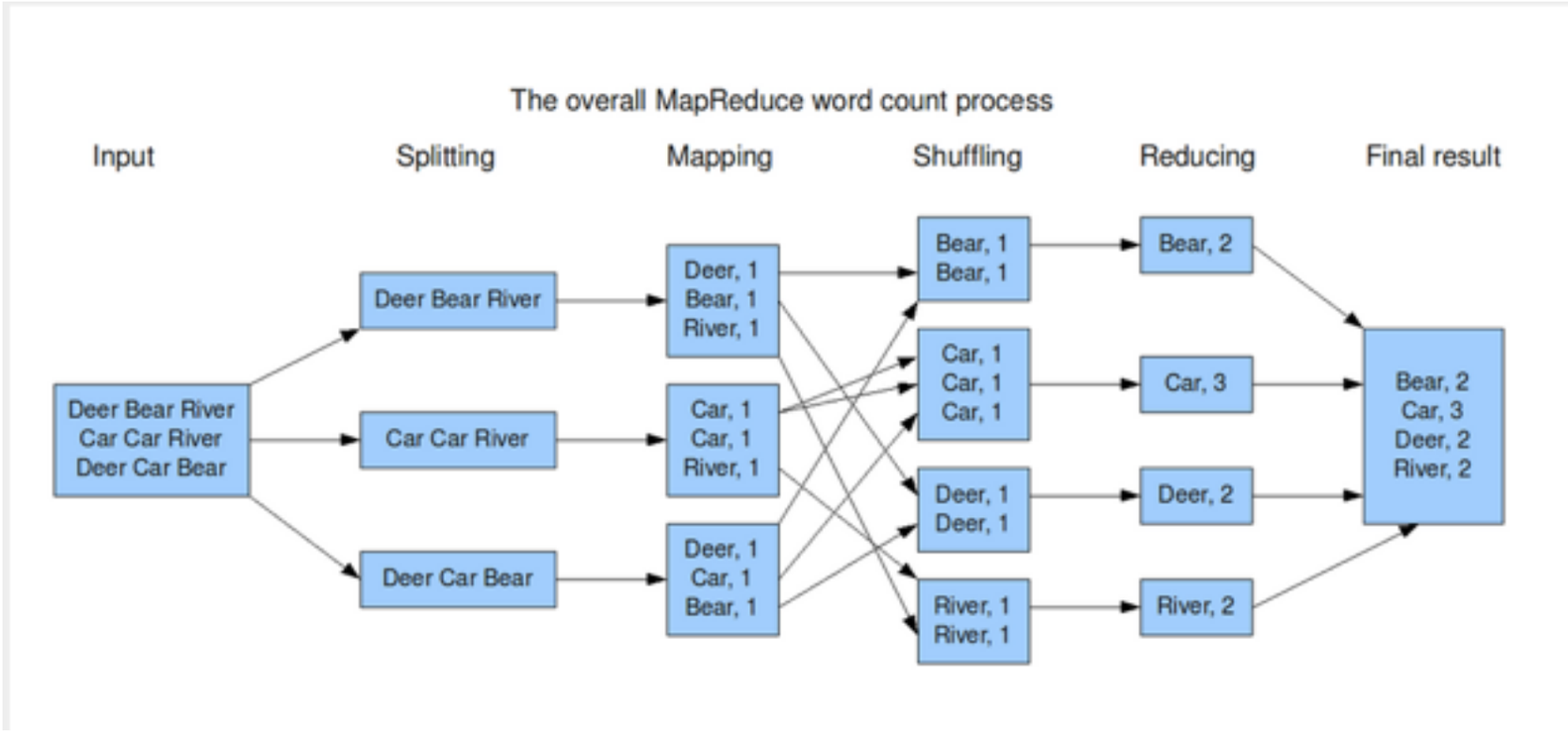
MapReduce


Map
Reduce


The overall MapReduce word count process
Input
Splitting
Mapping
Shuffling
Reducing
Final result
Deer Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2

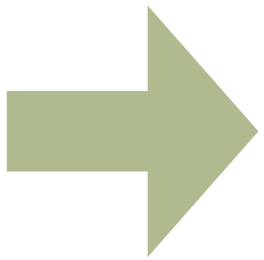
MapReduce

```
input            <k1, v1>

mapper           <k1, v1> ->  <k2, v2>

(partitioner)    <k2, v2> -> <k2, [all k2 values]>

reducer          <k2, [all k2 values]> ->  <k3, v3>
```



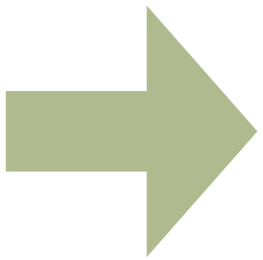
MapReduce

3 : the
3 : and
3 : you
4 : then
4 : what
4 : when
5 : steve
5 : where
8 : savannah
8 : research



They get grouped as:

3 : [the, and, you]
4 : [then, what, when]
5 : [steve, where]
8 : [savannah, research]



3 : 3
4 : 3
5 : 2
8 : 2

MapReduce

Person -> List of Friends

A -> B C D
B -> A C D E
C -> A B D E
D -> A B C E
E -> B C D

map(A -> B C D) :
(A B) -> B C D
(A C) -> B C D
(A D) -> B C D

map(B -> A C D E) :
(A B) -> A C D E
(B C) -> A C D E
(B D) -> A C D E
(B E) -> A C D E

map(C -> A B D E) :
(A C) -> A B D E
(B C) -> A B D E
(C D) -> A B D E
(C E) -> A B D E

map(D -> A B C E) :
(A D) -> A B C E
(B D) -> A B C E
(C D) -> A B C E
(D E) -> A B C E

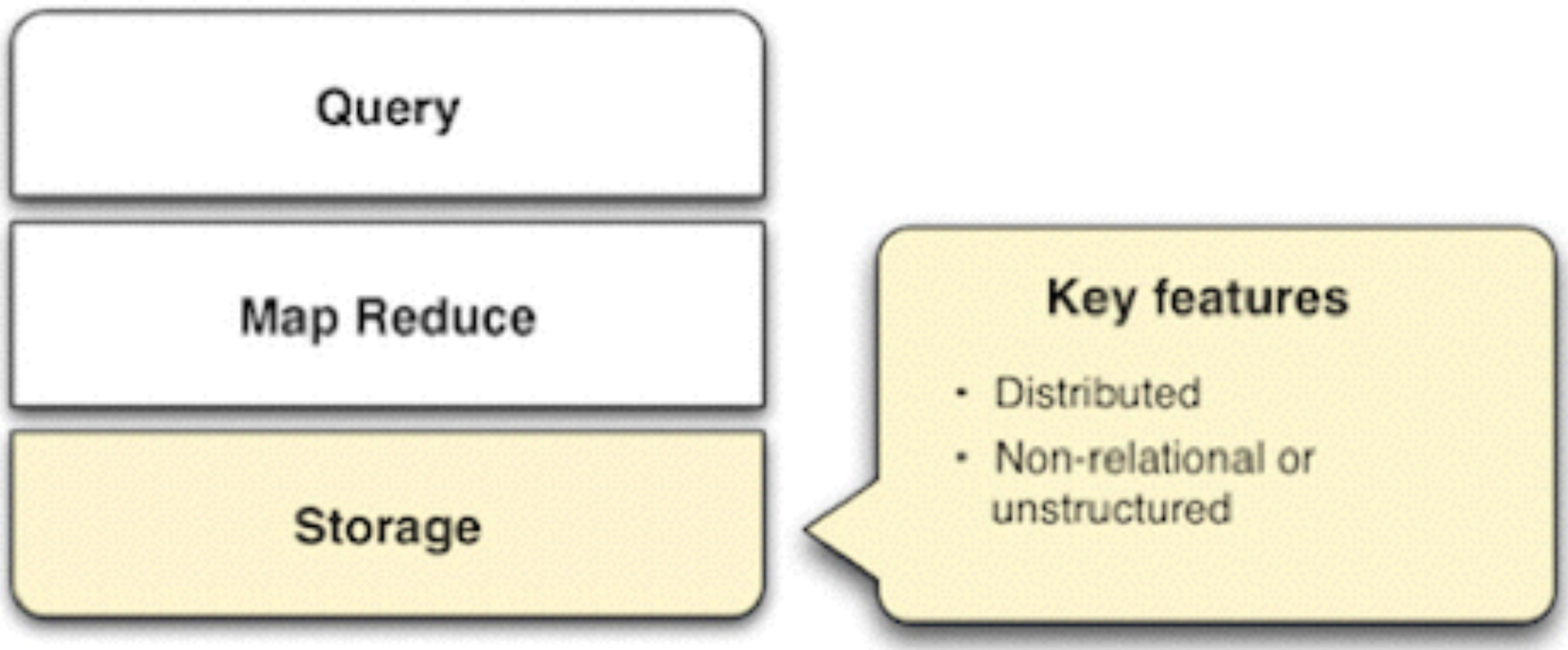
(A B) -> (A C D E) (B C D)
(A C) -> (A B D E) (B C D)
(A D) -> (A B C E) (B C D)
(B C) -> (A B D E) (A C D E)
(B D) -> (A B C E) (A C D E)
(B E) -> (A C D E) (B C D)
(C D) -> (A B C E) (A B D E)
(C E) -> (A B D E) (B C D)
(D E) -> (A B C E) (B C D)

The result after reduction is:

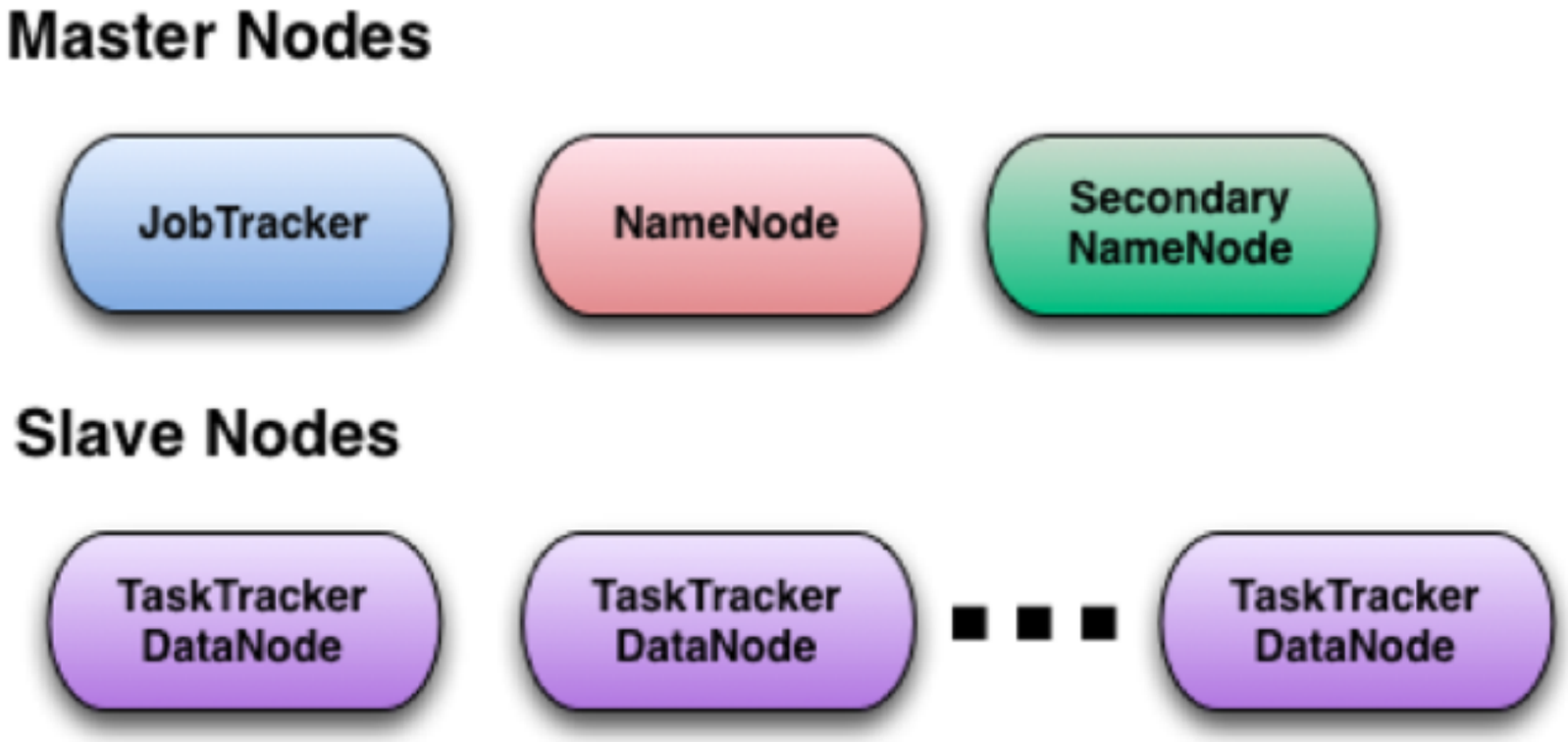
(A B) -> (C D)
(A C) -> (B D)
(A D) -> (B C)
(B C) -> (A D E)
(B D) -> (A C E)
(B E) -> (C D)
(C D) -> (A B E)
(C E) -> (B D)
(D E) -> (B C)

Now when D visits B's profile, we can quickly look up (B D) and see that they have three friends in common, (A C E).

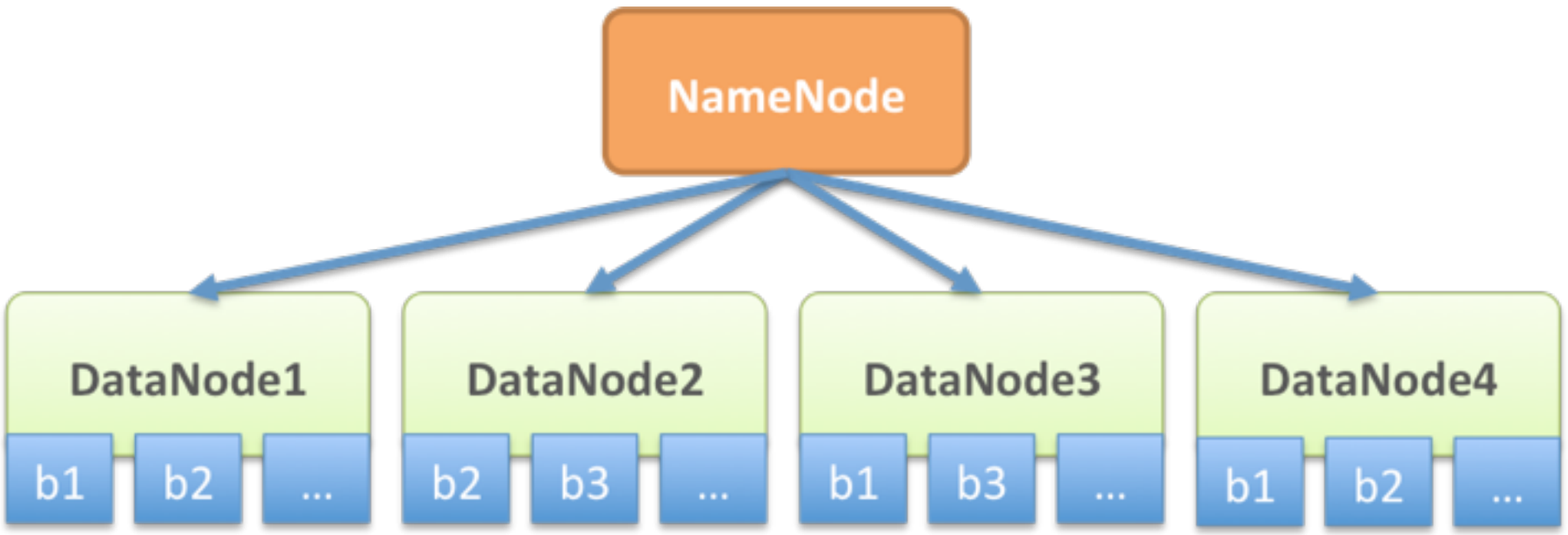
Hadoop Components



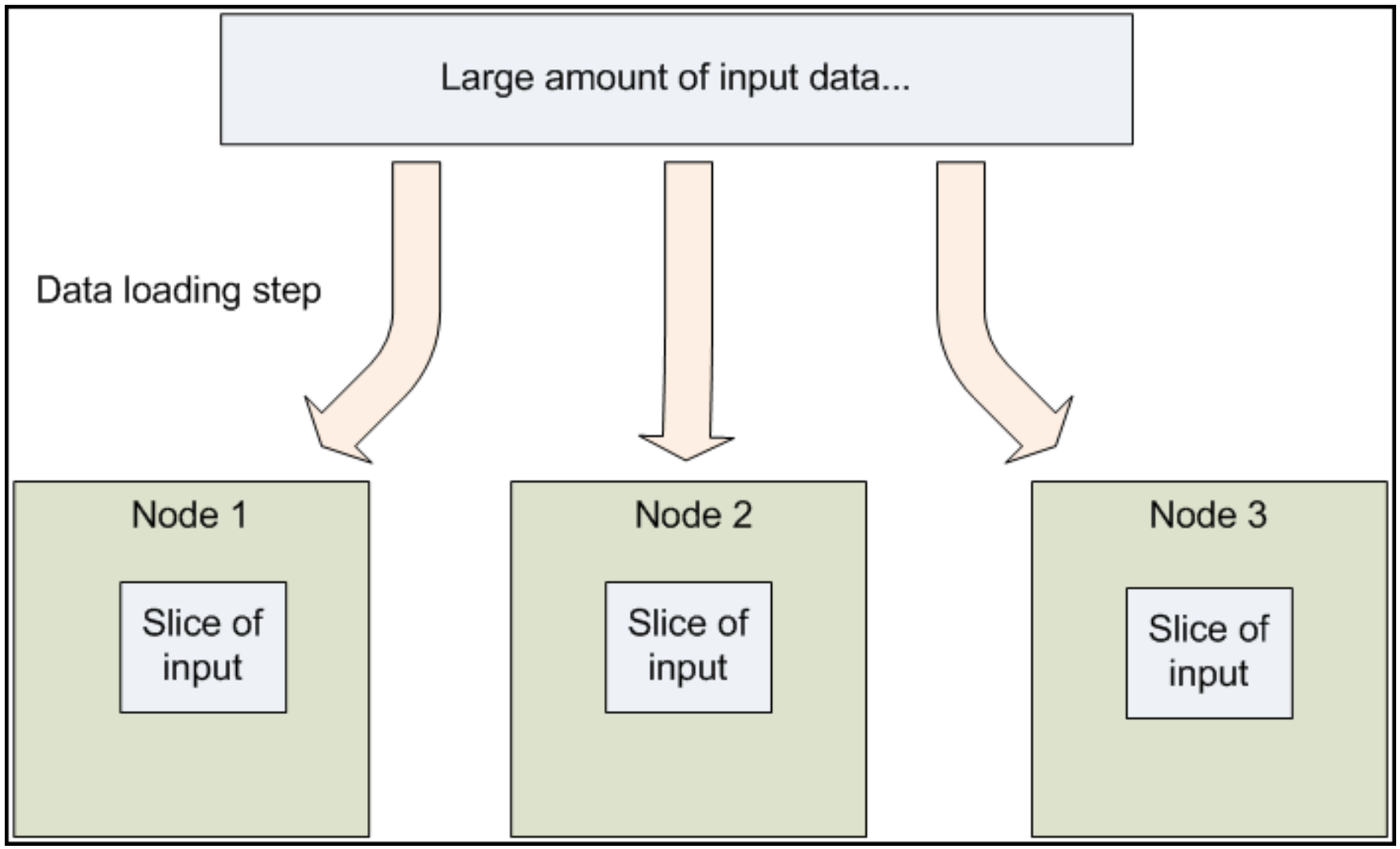
Hadoop Components



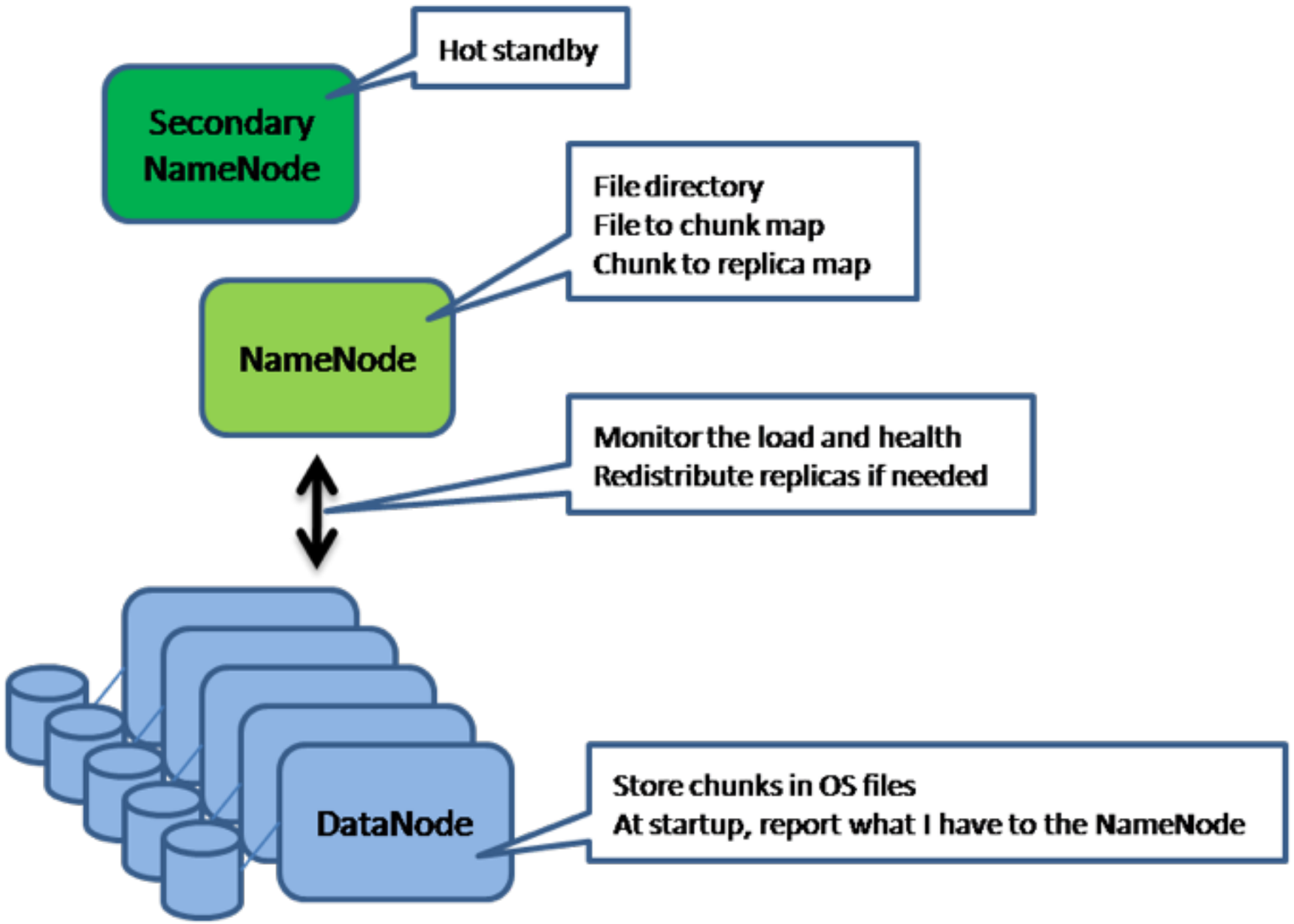
Hadoop Components



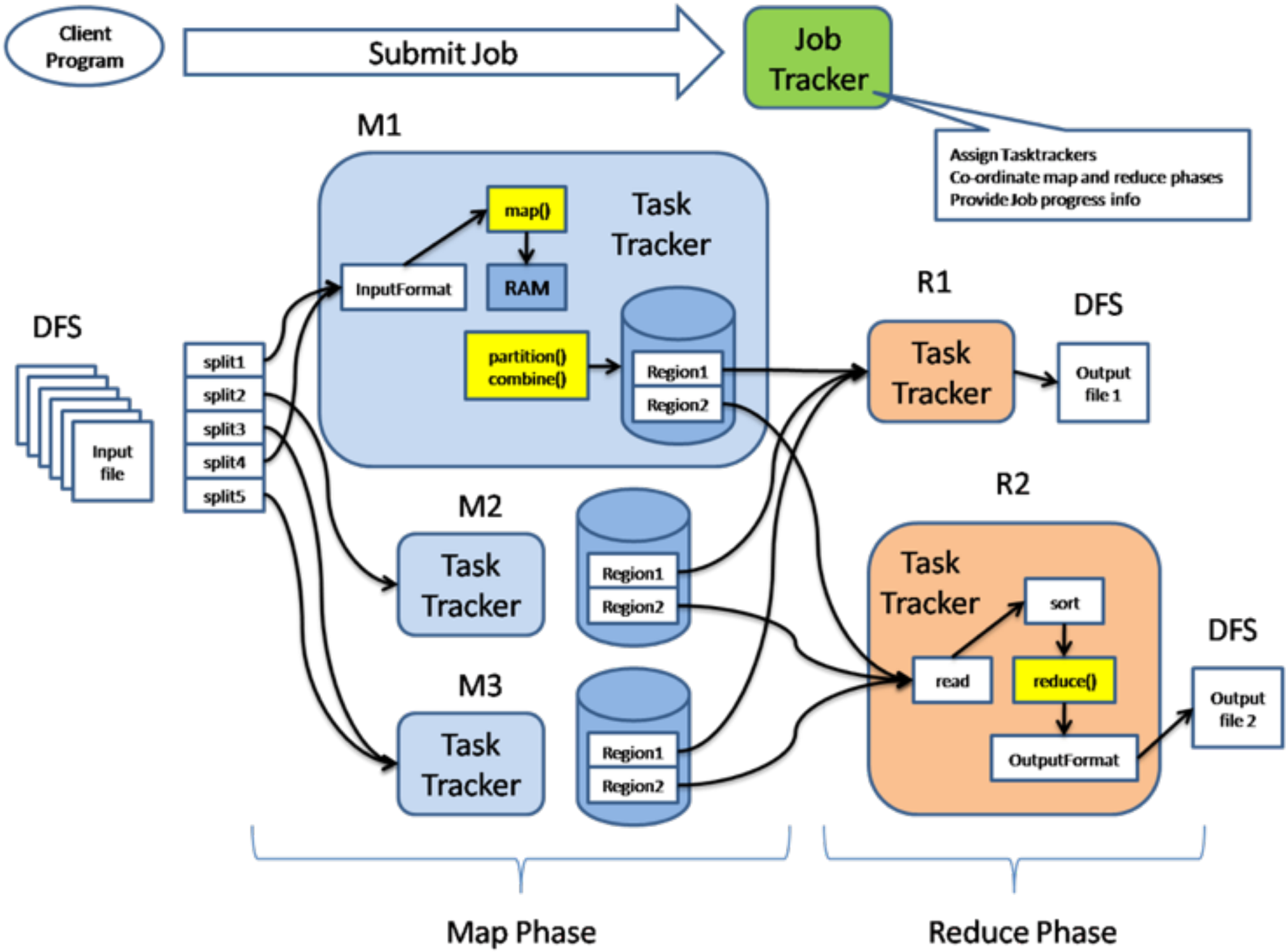
Hadoop Components



Hadoop Components



Hadoop Components



Beyond Hadoop

Beyond MapReduce

- Driven by two forces:
  - Users often frustrated with how long Hadoop jobs take to finish
    - Batch-processing
    - Spinning up a JVM for each task is expensive
- Writing MapReduce jobs in Java is painful and error-prone if you're not an expert
  - We want to give data access to as many people as possible, not just our Java developers
  - Analysts are typically familiar with SQL

Beyond MapReduce

- These forces have caused a few developments
  - Higher-level query languages that implement SQL-like semantics on top of MapReduce
  - Alternative data processing engines
    - In-memory
    - Intelligent pipelining
    - Offer near real-time access speeds

Beyond MapReduce

- Hive was developed by Facebook so their SQL analysts could write MR jobs
- Data queries are constructed with a SQL-like language called HiveQL
- Used for data summation, ad-hoc queries, and projecting structure onto data

Beyond MapReduce


```
CREATE TABLE raw_daily_stats_table (redirect_title STRING, dates
STRING, total_pageviews BIGINT, monthly_trend DOUBLE)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE;

INSERT OVERWRITE TABLE pages
SELECT redirect_table.page_id, redirect_table.redirect_title
FROM redirect_table
JOIN raw_daily_stats_table ON redirect_table.redirect_title;
```

Beyond MapReduce

- Pig was developed at Yahoo! for ad-hoc queries
- Data queries are constructed with a SQL-like language called *Pig Latin*
- Used for joining datasets, grouping data, datasets with many elements
- Supports intermediate data storage and functions such as FOREACH

Beyond MapReduce

```
input_lines = LOAD '/tmp/my-copy-of-all-pages-on-internet' AS
(line:chararray);

-- Extract words from each line and put them into a pig bag
-- datatype, then flatten the bag to get one word on each row
words = FOREACH input_lines GENERATE FLATTEN(TOKENIZE(line)) AS word;

-- filter out any words that are just white spaces
filtered_words = FILTER words BY word MATCHES '\\w+';

-- create a group for each word
word_groups = GROUP filtered_words BY word;

-- count the entries in each group
word_count = FOREACH word_groups GENERATE COUNT(filtered_words) AS
count, group AS word;

-- order the records by count
ordered_word_count = ORDER word_count BY count DESC;
STORE ordered_word_count INTO '/tmp/number-of-words-on-internet';
```

Beyond MapReduce

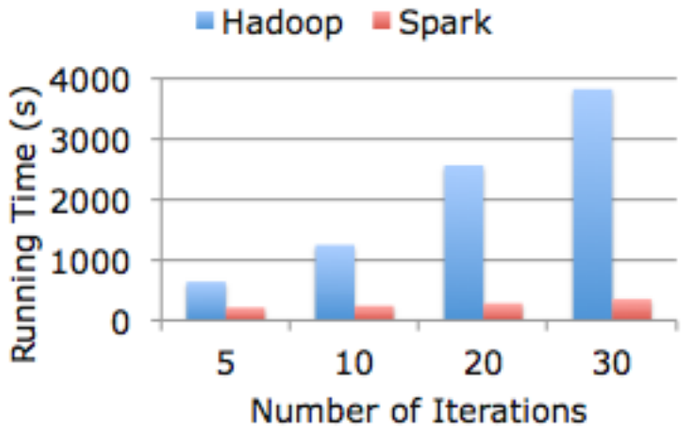
- **Key take-aways**
  - Both Hive and Pig compile down to MapReduce code which runs just like any other MR code
  - Usually only one is used, sometimes both, i.e. Pig to manipulate unstructured data, Hive to query that structured data
- **Depends on the skillset of the target users**
  - SQL background? Will gravitate towards Hive
  - No SQL background? Often choose Pig.

SQL still isn’t fast enough

Beyond MapReduce

Lightning-Fast Cluster Computing

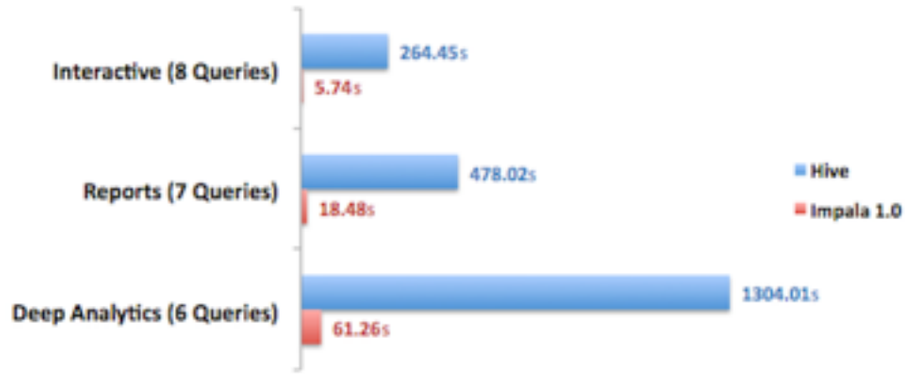
```
file = spark.textFile("hdfs://...")

file.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
```



- Spark jobs keep the majority of data in-memory for repeated querying
- Useful for *iterative* algorithms (machine learning) and *interactive* data mining
- Can be up to 100x faster than MapReduce for some applications

Beyond MapReduce

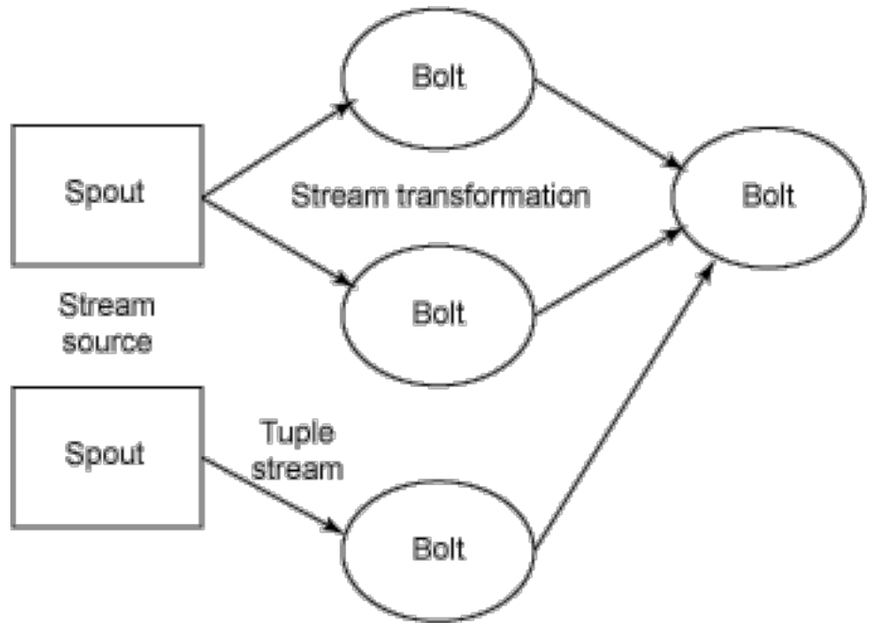
cloudera®
IMPALA



- Distributed MPP query engine written in C++
- In-memory data transfers, avoids costs of spinning up JVMs
- Runs on every node in the cluster, bypassing normal MR
- Open-source, but currently only supported by Cloudera

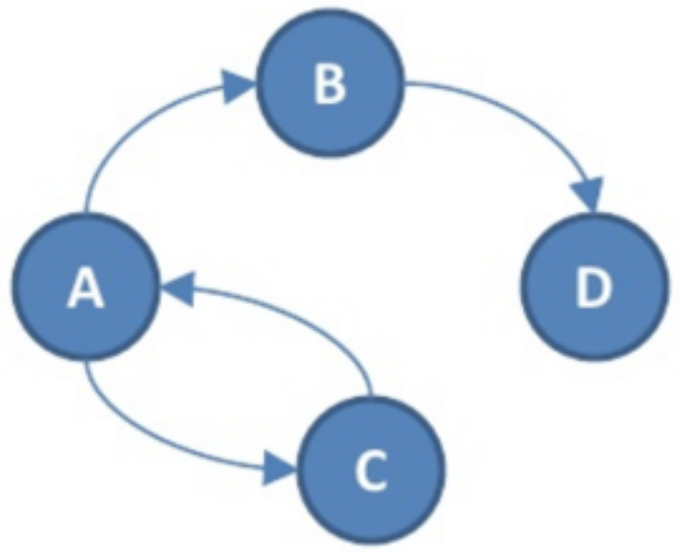
In-memory isn’t fast enough!

Real-time Processing



- **Twitter Storm** was built to do real-time data processing on Hadoop
- Allows construction of topologies that transform unterminated streams of data
- These streams never stop, and unlike MR jobs, continue to process data as it arrives
- Implemented on Spark!

Graph Processing



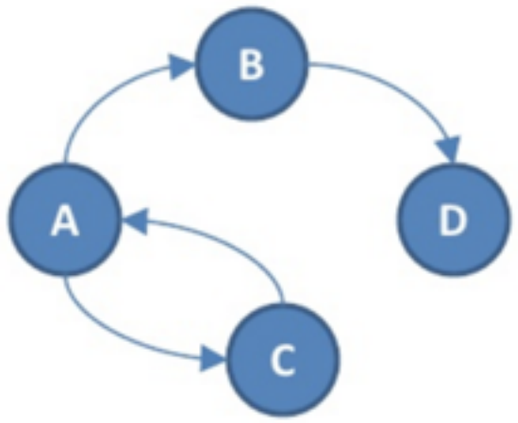
- **Apache Giraph** is an open-source implementation of Google Pregel
- Uses Bulk Synchronous Processing (BSP) to efficiently process graphs
- Interesting sidenote, while MR was developed to run PageRank, it is now entirely run on Pregel and is a couple order of magnitudes faster!

Graph recap

graph: abstract representation of a set of objects (**vertices**), where some pairs of these objects are connected by links (**edges**), which can be directed or undirected

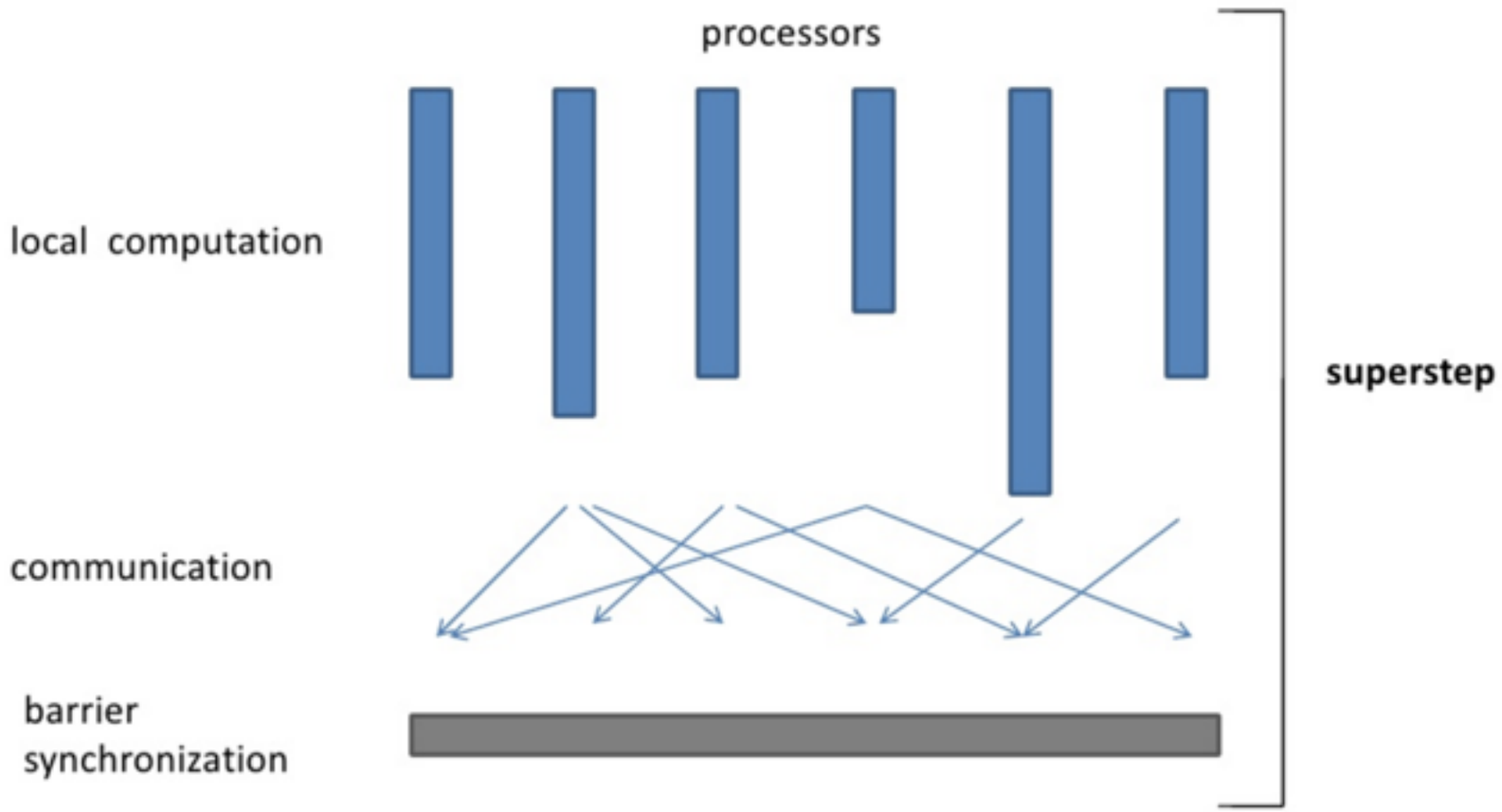
Graphs can be used to model arbitrary things like road networks, social networks, flows of goods, etc.

Majority of graph algorithms are iterative and traverse the graph in some way



Source: http://www.slideshare.net/sscdotopen/introducing-apache-giraph-for-large-scale-graph-processing

Bulk Synchronous Parallel (BSP)



Source: http://www.slideshare.net/sscdotopen/introducing-apache-giraph-for-large-scale-graph-processing

Hadoop Ecosystem

- **Mahout**
  - Distributed ML Library
- **Flume**
  - Framework for streaming event integration, used mostly for log files
- **Sqoop**
  - Literally "SQL & Hadoop"
  - Batch exchange of relational database tables
- **Oozie**
  - Process orchestration and basic scheduling

BE WARNED

"Hadoop makes easy things hard and hard things possible."

"When all you have is a hammer, everything looks like a nail."

Hadoop is not useful in *every* situation.
If you can get away with not using it,
don't.