



Ingeniería Industrial

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

INTRODUCTION TO BIG DATA

Juan D. Velásquez

Felipe E. Vildoso



Ingeniería Industrial
FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

CHAPTER 2



Ingeniería Industrial

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

LECTURE 10

Chapter 2



Ingeniería Industrial
FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

BATCH LAYER

BATCH LAYER

You have learned how to form a data model for your dataset and how to store your data in the batch layer in a scalable way.

Now you'll take the **next step** of learning **how to compute arbitrary functions on that data**.

EXAMPLE 1: PAGEVIEWS

- Dataset of pageviews.
- Each pageview record contains a URL and timestamp.

Total number of pageviews of a URL for a range given in hours.

How can this query be written in pseudo-code?

```
function pageviewsOverTime(masterDataset, url, startHour, endHour) {  
    pageviews = 0  
    for(record in masterDataset) {  
        if(record.url == url && record.time >= startHour &&  
record.time <= endHour) {  
            pageviews += 1  
        }  
    }  
    return pageviews  
}
```


EXAMPLE 2: GENDER INFERENCE

- Dataset of name records.

Predict the likely gender for a person.

How can this query be written in pseudo-code?

The algorithm first performs **semantic normalization on the names** for the person, doing conversions like Bob to Robert and Bill to William.



```
function genderInference(masterDataset, personId) {  
    names = new Set()  
    for(record in masterDataset) {  
        if(record.personId == personId) {  
            names.add(normalizeName(record.name))  
        }  
    }  
}
```

The algorithm then makes use of a model that provides the **probability of a gender for each name**.

```
maleProbSum = 0.0
for(name in names) {
    maleProbSum += maleProbabilityOfName(name)
}
maleProb = maleProbSum / names.size()
if(maleProb > 0.5) {
    return "male"
}
else {
    return "female"
}
}
```

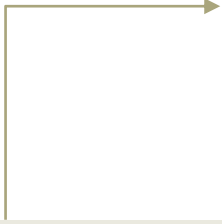
EXAMPLE 3: INFLUENCE SCORE

- Twitter-inspired dataset containing *reaction* records.
- Each reaction record contains **sourceId** and **responderId** fields, indicating that **responderId** retweeted or replied to **sourceId**'s post.

Determine an influencer score for each person.

How can this query be written in pseudo-code?

The top influencer for each person is selected based on the number of reactions the influencer caused in that person.



```
function influence_score(masterDataset, personId) {  
    influence = new Map()  
    for(record in masterDataset) {  
        curr = influence.get(record.responderId)  
        curr[record.sourceId] += 1  
        influence.set(record.sourceId, curr)  
    }  
}
```

someone's influence score is set to the number of people for which he or she was the top influencer.

```
score = 0
for(entry in influence) {
    if(topKey(entry.value) == personId) {
        score += 1
    }
}
return score
}
```



Ingeniería Industrial

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

COMPUTING ON THE BATCH LAYER

COMPUTING ON THE BATCH LAYER

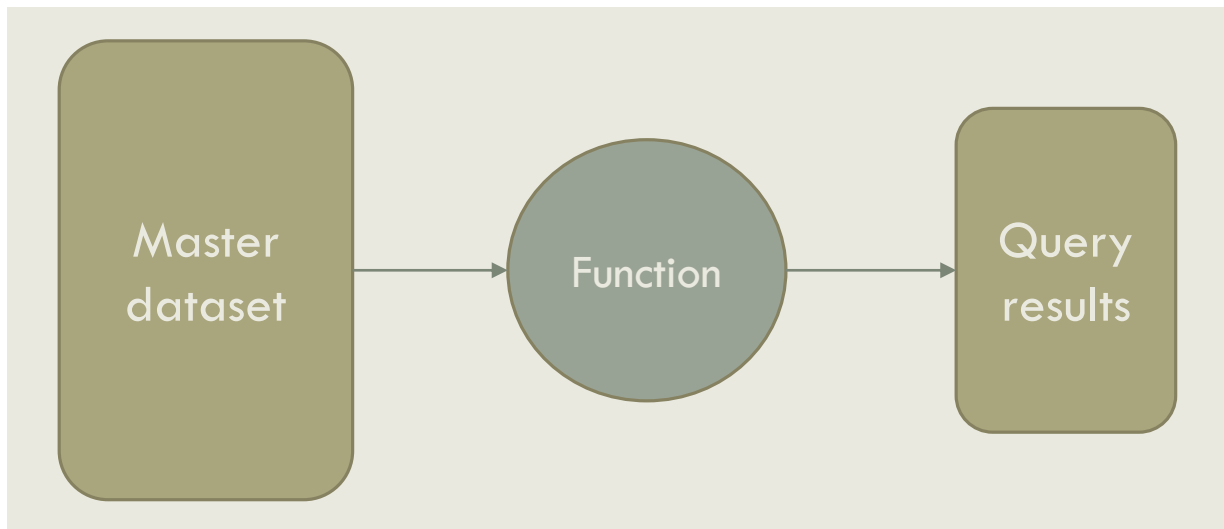
Recall, the Batch Layer runs functions over the master dataset to precompute intermediate data called ***batch views***.

The batch views are loaded by the serving layer, which indexes them to allow rapid access to that data.

The **speed layer compensates for the high latency** of the batch layer by providing low-latency updates using data that has yet to be precomputed into a batch view.

COMPUTING ON THE BATCH LAYER

Queries are then satisfied by processing data from the serving layer views and the speed layer views, and merging the results.



COMPUTING ON THE BATCH LAYER

But, which queries should be precomputed?

A naive strategy for computing on the batch layer would be to precompute all possible queries and cache the results in the serving layer.

Unfortunately **you can't always precompute everything**.

COMPUTING ON THE BATCH LAYER

Consider the pageviews-over-time query as an example.

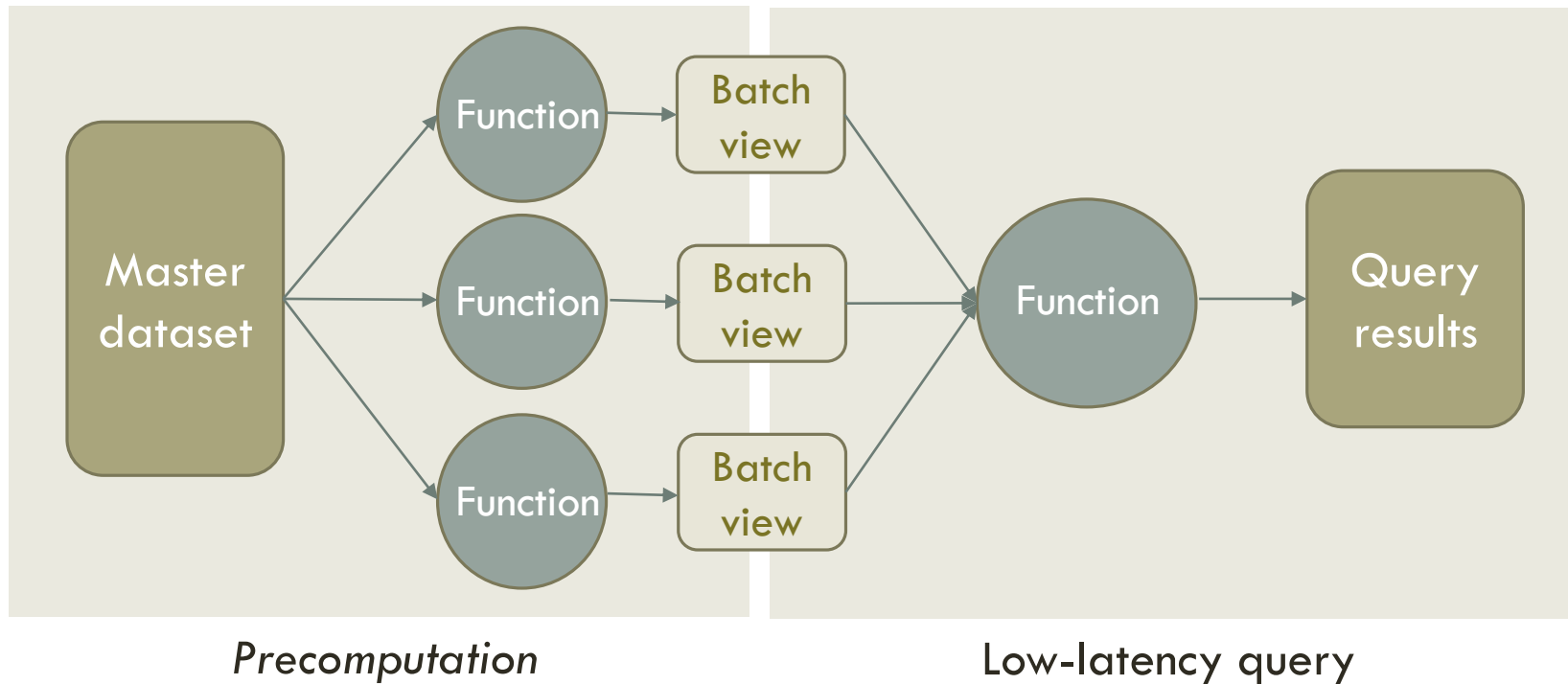
If you wanted to precompute every potential query, you'd need to determine the answer for every possible range of hours for every URL.

How many range of hours are in a year?

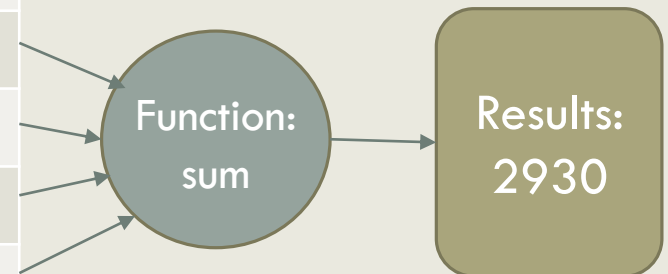
COMPUTING ON THE BATCH LAYER

It's not necessary to do it all once. Instead, you can **precompute intermediate results** and then use these results to complete queries on the fly.

COMPUTING ON THE BATCH LAYER



URL	Hour	#Pageviews
foo.com/blog	2012/12/08 15:00	876
foo.com/blog	2012/12/08 16:00	987
foo.com/blog	2012/12/08 17:00	762
foo.com/blog	2012/12/08 18:00	413
foo.com/blog	2012/12/08 19:00	1098
foo.com/blog	2012/12/08 20:00	657
foo.com/blog	2012/12/08 21:00	101



COMPUTING ON THE BATCH LAYER

Thinking on the example before.

For a single year, how many values you would need to compute on the fly? only need to precompute

This is certainly a more manageable number. Don't you think?



Ingeniería Industrial

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

RECOMPUTATION ALGORITHMS VS. INCREMENTAL ALGORITHMS

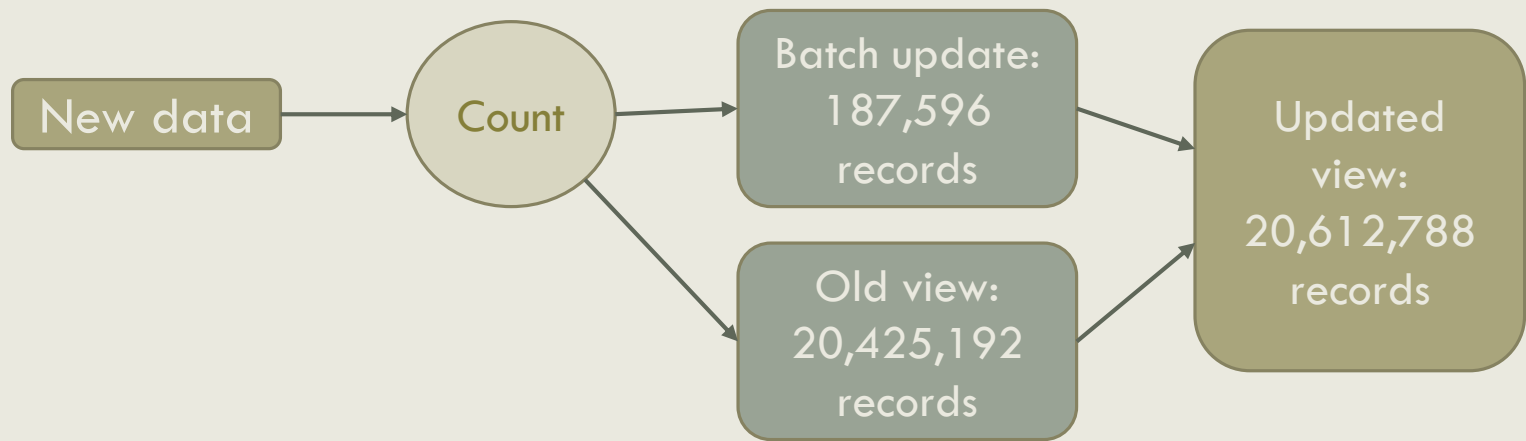
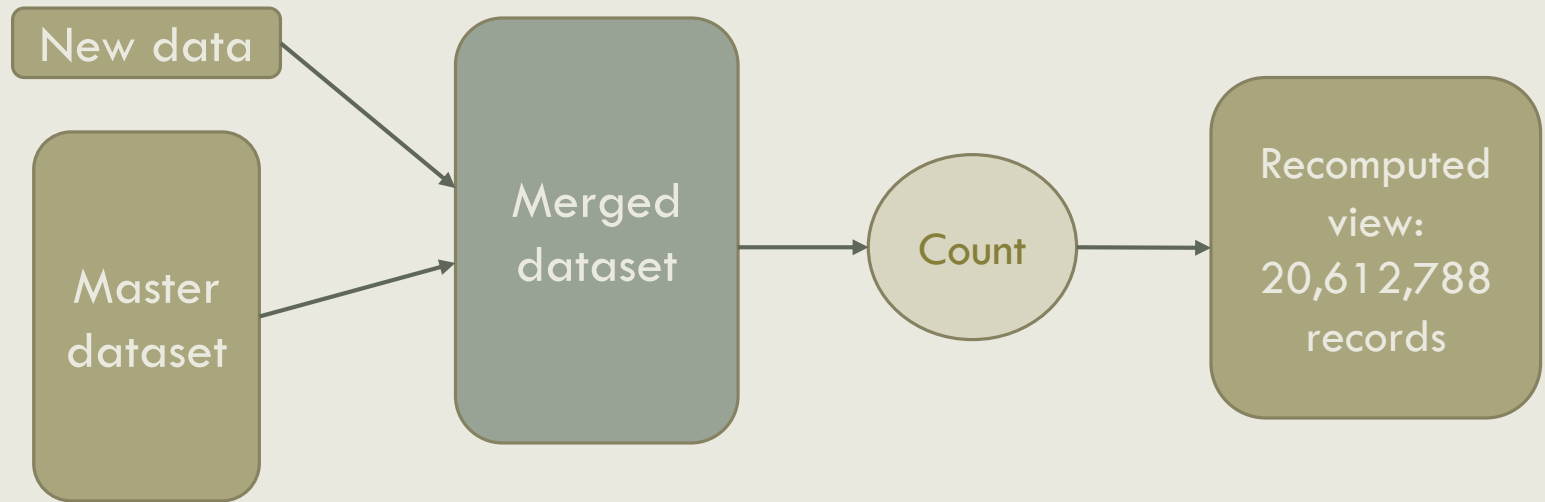
RECOMPUTATION ALGORITHMS VS. INCREMENTAL ALGORITHMS

Because your master dataset is continually growing, you must have a **strategy for updating your batch views** when new data becomes available.

You have two options:

- Recomputation algorithms.
- Incremental algorithms.

Which one do you think is a better option?



RECOMPUTATION ALGORITHMS VS. INCREMENTAL ALGORITHMS

Why you would ever use a recomputation algorithm when you can use a vastly more efficient incremental algorithm instead.

Is efficiency is the only factor to be considered?

The key trade-offs between the two approaches are:

- Performance
- Human-fault tolerance
- The generality of the algorithm

PERFORMANCE

There are two aspects to the performance of a batch-layer algorithm:

- The amount of resources required to update a batch view with new data.
- The size of the batch views produced.

PERFORMANCE

An **incremental algorithm** almost always uses **significantly less resources** to update a view because it uses new data and the current state of the batch view to perform an update.

However, the **size of the batch view** for an incremental algorithm can be significantly **larger** than the corresponding batch view for a recomputation algorithm.

PERFORMANCE

URL	#Unique visitors
foo.com	2217
foo.com/blog	1899
foo.com/about	524
foo.com/careers	413
foo.com/faq	1212
...	...

Recomputation batch view

URL	#Unique visitors	Visitor IDs
foo.com	2217	1,4,5,7,10,12,...
foo.com/blog	1899	2,3,5,17,22,...
foo.com/about	524	3,6,7,19,...
foo.com/careers	413	12,17,19,29,...
foo.com/faq	1212	8,10,37,39,...
...

Incremental batch view

HUMAN-FAULT TOLERANCE

The lifetime of a data system is extremely long, and bugs can and will be deployed to production during that time period.

You therefore must consider how your batch update algorithm will tolerate such mistakes.

In this regard, **recomputation algorithms are inherently human-fault tolerant**, whereas with an incremental algorithm, human mistakes can cause serious problems.

GENERALITY OF THE ALGORITHMS

Although incremental algorithms can be faster to run, they must often be tailored to address the problem at hand.

Incremental algorithm for computing the number of unique visitors can generate prohibitively large batch views.

GENERALITY OF THE ALGORITHMS

Probabilistic counting algorithms, such as HyperLog-Log, that store intermediate statistics to estimate the overall unique count.

- This reduces the storage cost of the batch view
- Algorithm approximate instead of exact.

GENERALITY OF THE ALGORITHMS

Incremental algorithms shift complexity to on-the-fly computations.

As you improve your semantic normalization algorithm, you'll want to see those improvements reflected in the results of your queries. Yet, if you do the **normalization as part of the precomputation**, your batch view will be **out of date** whenever you improve the normalization.

Recomputation Algorithms

Incremental Algorithms

Performance	Requires computational effort to process the entire dataset	Requires less computational resources but may generate much larger batch views
Human-fault Tolerance	Extremely tolerant of human errors because the batch views are continually rebuilt	Doesn't facilitate repairing errors in the batch views; repairs are ad hoc and may require estimates
Generality	Complexity of the algorithm is addressed during precomputation, resulting in simple batch views and low-latency, on-the-fly processing	Requires special tailoring; may shift complexity to on-the-fly query processing
Conclusion	Essential to supporting a robust data-processing system	Can increase the efficiency of your system, but only as a supplement to recomputation algorithms



Ingeniería Industrial

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

SCALABILITY IN THE BATCH LAYER

SCALABILITY IN THE BATCH LAYER

Scalability is the ability of a system to **maintain performance under increased load by adding more resources**.

Load in a Big Data context:

- Total amount of data you have.
- How much new data you receive every day.
- How many requests per second your application serves.

More important than a system being scalable is a system being **linearly scalable**, instead of being *nonlinearly scalable*.

MAPREDUCE

MapReduce is a **distributed computing paradigm** originally pioneered by Google that **provides primitives for scalable and fault-tolerant batch computation**.

With Map-Reduce, you write your computations in terms of *map* and *reduce* functions that manipulate key/value pairs.

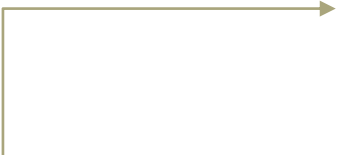
MAPREDUCE

These primitives are expressive enough to implement nearly any function, and the **MapReduce framework executes those functions** over the master dataset in a **distributed** and **robust** manner.

The canonical MapReduce example is *word count*.

MAP FUNCTION OF THE WORD COUNT EXAMPLE

MapReduce then arranges the output from the map functions so that **all values from the same key are grouped together**.



```
function word_count_map(sentence) {  
    for(word in sentence.split(" ")) {  
        emit(word, 1)  
    }  
}
```

REDUCE FUNCTION OF THE WORD COUNT EXAMPLE

```
function word_count_reduce(word, values) {  
    sum = 0  
    for(val in values) {  
        sum += val  
    }  
    emit(word, sum)  
}
```

The reduce function then **takes the full list of values sharing the same key** and **emits new key/value pairs** as the final output.

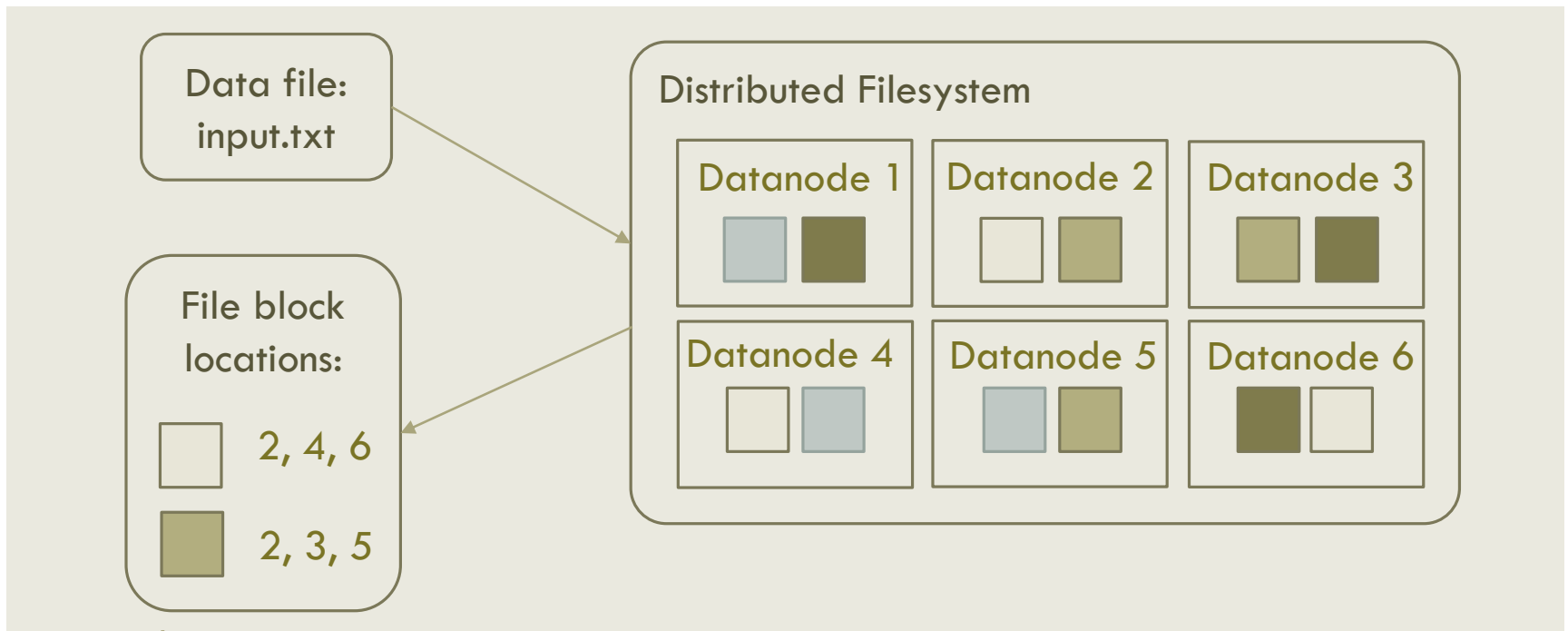
SCALABILITY

Programs written in terms of MapReduce are **inherently scalable**.

A program that runs on 10 gigabytes of data will also run on 10 petabytes of data.

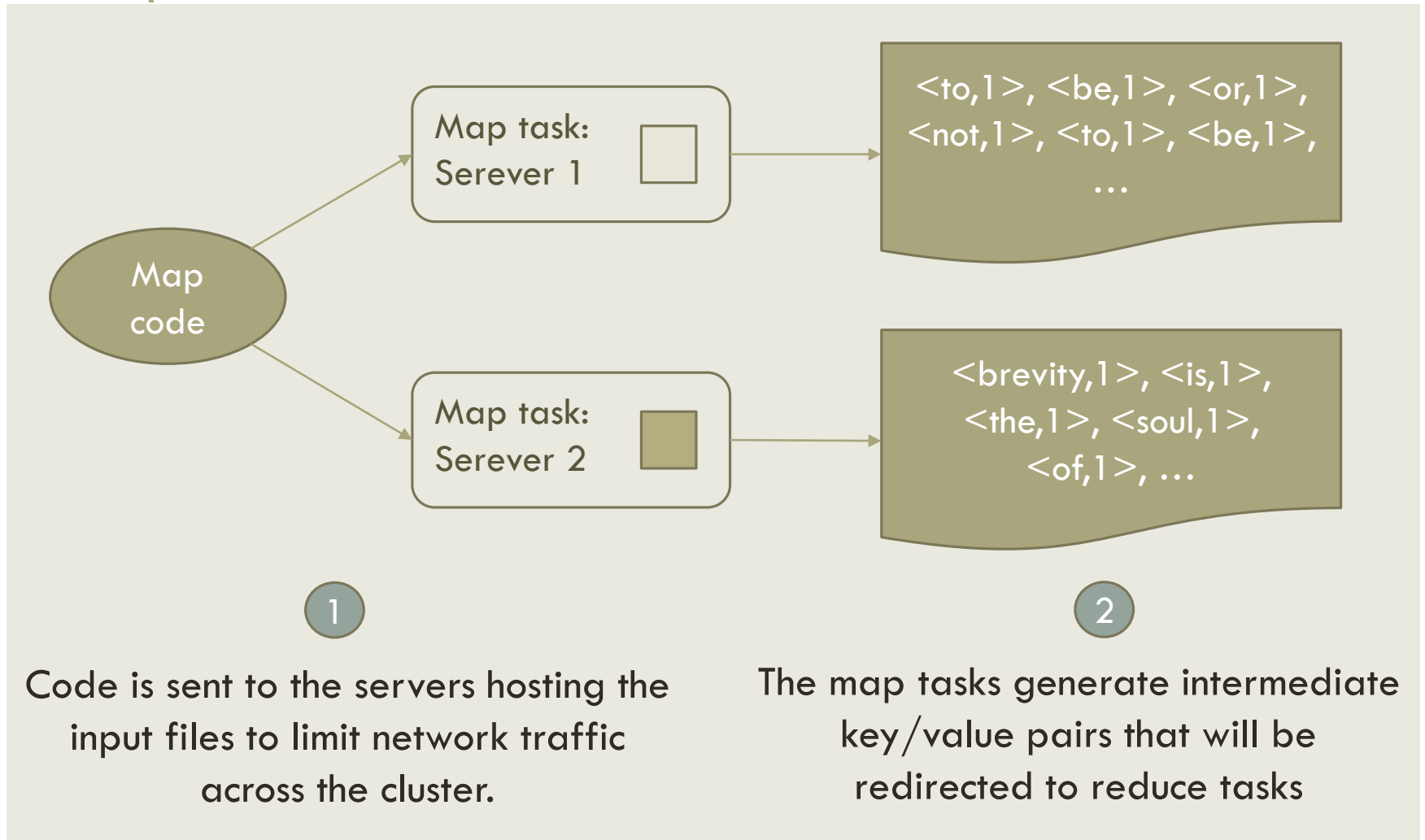
MapReduce automatically parallelizes the computation across a cluster of machines regardless of input size.

Let's walk through how a program like word count executes on a MapReduce cluster.



After determining the locations of the input, MapReduce launches a number of map tasks proportional to the input data size.

Each of these tasks is assigned a subset of the input and executes your map function on that data.



MAPREDUCE

Because the amount of the code is typically far less than the amount of the data, MapReduce attempts to **assign tasks to servers that host the data to be processed**.

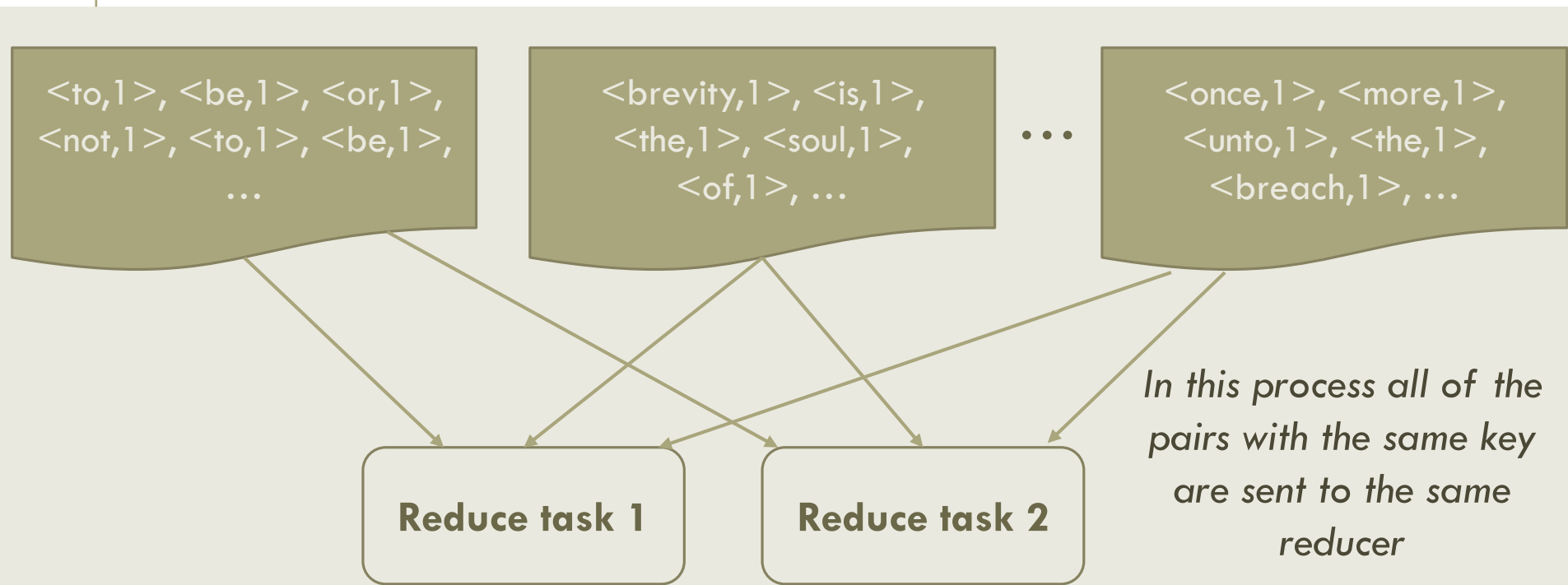
Like map tasks, there are also reduce tasks spread across the cluster.

MAPREDUCE

Like map tasks, there are also reduce tasks spread across the cluster.

Because the reduce function requires all values associated with a given key, a reduce task can't begin until all map tasks are complete.

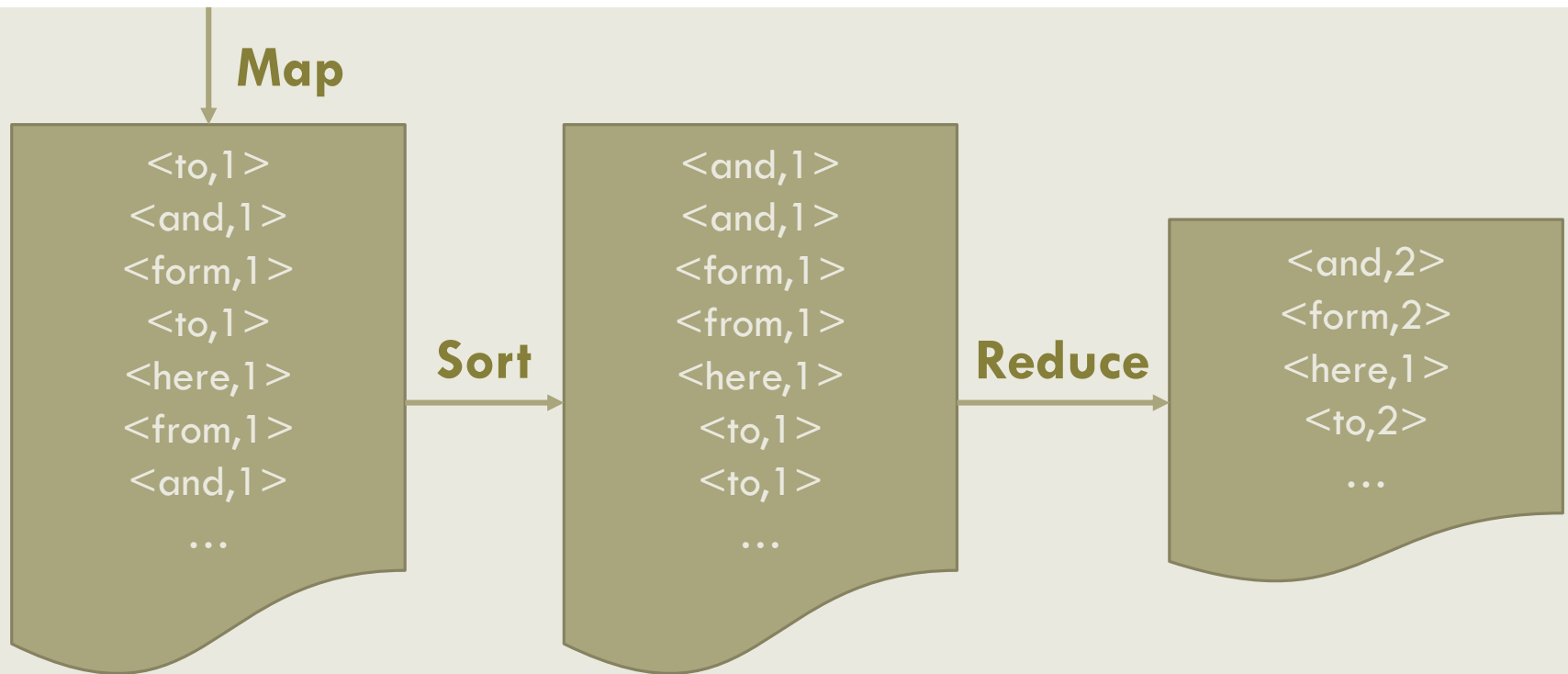
Once the map tasks finish executing, each emitted key/value pair is sent to the reduce task responsible for processing that key. This **transfer of the intermediate key/value pairs** is called **shuffling**.



MAPREDUCE

Once a reduce task receives all of the key/value pairs from every map task, it **sorts the key/value pairs by key**. This has the effect of organizing all the values for any given key to be together. The reduce function is then called for each key and its group of values.

MAPREDUCE SUMMARY



MAPREDUCE SUMMARY

- MapReduce programs execute in a **fully distributed** fashion with no central point of contention.
- MapReduce is **scalable**: the map and reduce functions you provide are **executed in parallel** across the cluster.
- The challenges of **concurrency** and **assigning tasks** to machines is **handled for you**.

FAULT-TOLERANCE

Network partitions, server crashes, and disk failures are relatively rare for a single server, but the likelihood of something going wrong greatly increases when coordinating computation over a large cluster of machines.

Thankfully, in addition to being easily parallelizable and inherently scalable, **MapReduce computations are also fault tolerant.**

FAULT-TOLERANCE

A program can fail for a variety of reasons:

- A hard disk can reach capacity.
- The process can exceed available memory.
- The hardware can break down.

MapReduce **watches for these errors** and **automatically retries** that portion of the computation on another node.

FAULT-TOLERANCE

An entire application (commonly called a *job*) will fail only if a task fails more than a configured number of times—typically four.

The idea is that a single failure may arise from a server issue, but **a repeated failure is likely a problem with your code.**

FAULT-TOLERANCE

Because tasks can be retried, MapReduce **requires that your map and reduce functions be deterministic.**

This means that given the same inputs, your functions must always produce the same outputs. It's a relatively light constraint but important for MapReduce to work correctly.

Higher performance: Spark is able to cache that data in memory rather than read it from disk every time.

Model: “resilient distributed datasets”.

MAPREDUCE VS. SPARK

Spark isn't any more general or scalable than MapReduce.

New computation system that has gained a lot of attention.



Ingeniería Industrial

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

QUESTIONS?
SEE YOU ON THE NEXT CLASS!