



Google

MapReduce

I gave you all concept in this section please hear me 😊

Lecturer : farzad salimi jazi




MAPREDUCE BASICS

MapReduce Basics

- divide and conquer 

The only feasible approach to tackling large-data problems today

The basic idea is  partition a large problem into smaller subproblems

they can be tackled in parallel by different workers

Worker  (threads , core , processors , machines)

- general principles behind divide-and-conquer algorithms are broadly applicable

divide and conquer implementation issues

- How do we break up a large problem
- How do we assign tasks to distributed workers
- How do we ensure that the workers get the data they need?
- How do we coordinate synchronization among the different workers?
- How do we share partial results from one worker that is needed by another?
- How do we accomplish all of the above in the face of software errors and hardware
- faults?

Face to issue

- traditional parallel or distributed programming environments
- Shared memory programming
- Language extensions
 - OpenMP for shared memory parallelism
 - (MPI) for cluster-level parallelism
 - provide logical abstractions
 - developers are burdened to keep track of how resources are made available to workers
 - Elementary support for dealing with very large amounts of input data

But by Mapreduce 😊

- provides an abstraction that **hides** many **system-level details**
- organizing and coordinating large amounts of **computation** is only part of the challenge
- **bringing data and code together** for computation to occur
 - Now what is the Mapreduce idea ?!!

Mapreduce idea 😊

- instead of moving large amounts of data around, it is far more efficient, if possible, to move the code to the data
 - spreading data across the local disks of nodes in a cluster
 - running processes on nodes that hold the data
 - managing is typically handled by a distributed file system that sits underneath MapReduce

☺ this section ...

We introduces the MapReduce programming model and the underlying distributed file system step by step by :

- › an overview of functional programming
- › introducing the basic programming model, focusing on mappers and reducers
- › discussing the role of the execution framework in actually running MapReduce programs (called jobs)
- › introducing partitioners and combiners
- › Details for distributed file system that manages the data being processed
- › Describing complete cluster architecture

FUNCTIONAL PROGRAMMING ROOTS

- MapReduce **has its roots** in functional programming (**Lisp and ML**)
- A key feature of functional languages is the concept of higherorder functions, or **functions that can accept other functions as arguments**
- Two common built-in higher order functions are **map** and **fold**

Map and Fold

Trace computing the sum of squares of a list of integers 😊

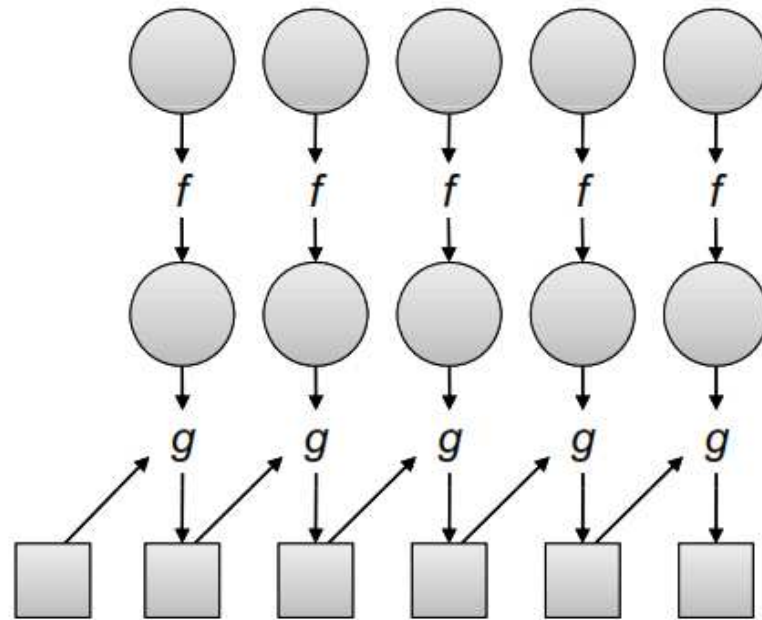


Figure 2.1: Illustration of *map* and *fold*, two higher-order functions commonly used together in functional programming: *map* takes a function *f* and applies it to every element in a list, while *fold* iteratively applies a function *g* to aggregate results.

In the other word ...

- **view map as a** concise way to represent the transformation of a dataset (as defined by the function f)
- **view fold as** an aggregation operation, as defined by the function g .

Map and fold parallelism

- Application of map to each item in a list can be parallelized **because they are isolated**
- The fold operation, on the other hand, has more restrictions on data locality
 - If elements in the list can be divided into groups, the fold aggregations can also proceed in parallel
- operations that are **commutative** and **associative**

What is Mapreduce ?

Mapreduce	Functional programming
map	map
reduce	fold

- MapReduce codifies a generic “recipe” for processing large datasets that consists of two stages
 - ✓ Two user-specified computation
 - ✓ programmer defines these two
 - ✓ framework coordinates the actual processing

That is very useful don't underestimate it

MapReduce can refer to three distinct but related concepts

- MapReduce is a **programming model**
- can refer to the **execution framework**
- can refer to the software **implementation** of the programming model and the execution framework
 - Google's proprietary implementation
 - open-source Hadoop implementation in Java
 - for **multi-core** processors , **GPGPUs**, **CELL** architecture

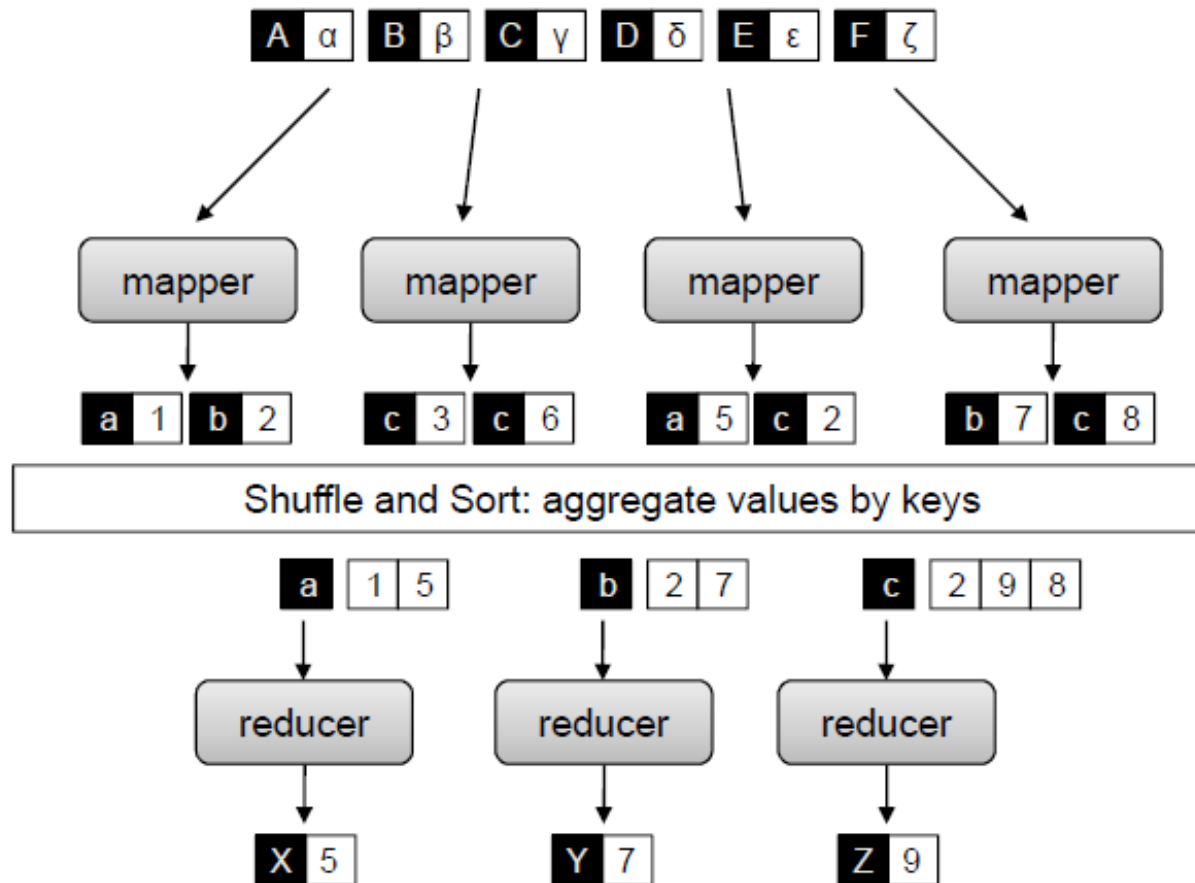
MAPPERS AND REDUCERS

- **Key-value pairs** form the basic data structure
 - primitives like integers ..
 - arbitrarily complex structures like list ...
- Library for help 😊
 - **Buffers** , **Thrift** and **Avro**
- imposing the key-value structure on arbitrary datasets **is important**
 - ✓ collection of web pages \longrightarrow (URL, content)
 - ✓ a graph \longrightarrow (node ids , adjacency lists)
 - ✓ input keys are not particularly meaningful
 - ✓ input keys are used to uniquely identify

mapper and a reducer structure

map: $(k1; v1) \longrightarrow [(k2; v2)]$

reduce: $(k2; [v2]) \longrightarrow [(k3; v3)]$



Simplified view of MapReduce

Mappers are applied to all input key-value pairs, which generate an arbitrary number of intermediate key-value pairs. Reducers are applied to all values associated with the same key. Between the map and reduce phases lies a barrier that involves a large distributed sort and group by

Word Count Example

- **Input:** Large number of text documents
- **Task:** Compute word count across all the document

Solution

- **Mapper:**
 - For every word in a document output (word, "1")
- **Reducer:**
 - Sum all occurrences of words and output (word, total_count)

```

1: class MAPPER
2:     method MAP(docid  $a$ , doc  $d$ )
3:         for all term  $t \in$  doc  $d$  do
4:             EMIT(term  $t$ , count 1)

1: class REDUCER
2:     method REDUCE(term  $t$ , counts [ $c_1, c_2, \dots$ ])
3:          $sum \leftarrow 0$ 
4:         for all count  $c \in$  counts [ $c_1, c_2, \dots$ ] do
5:              $sum \leftarrow sum + c$ 
6:             EMIT(term  $t$ , count  $sum$ )

```

Pseudo-code for the word count algorithm in MapReduce

The mapper emits an intermediate key-value pair for each word in a document. The reducer sums up all counts for each word.

Hadoop VS Google's implementation

Google	Hadoop
not allowed to change the key in the reducer	there is no such restriction
allows the programmer to specify a secondary sort key	not allowed

other variations ...

- MapReduce programs can contain **no reducers**
The
 - parse a large text collection
 - analyze a large number of images
- **converse is not possible**
 - simply pass input key-value pairs to the reducers for **sorting** and **regrouping** the input

Where they use for storing ...

- input to a MapReduce job comes from data stored on the distributed file system and output is written back to it
- Google's MapReduce
- **BigTable**
 - a sparse, distributed, persistent multidimensional sorted map
- Hadoop
 - **Hbase** is an open-source BigTable clone
 - **MPP** relational databases

BigTable

a **sparse, distributed,**
persistent **multidimensional**
sorted map

- ✓ **Fundamentally Distributed**
- ✓ **Column Oriented**
- ✓ **Variable num of Columns**

Unlike traditional **RDBMS**
where each "**row**" is **stored**
contiguous on disk, **BigTable**,
store each **column**
contiguously on disk

Row Oriented
(RDBMS Model)

id	Name	Age	Interests
1	Ricky		Soccer, Movies, Baseball
2	Ankur	20	
3	Sam	25	Music

Multi-valued

null

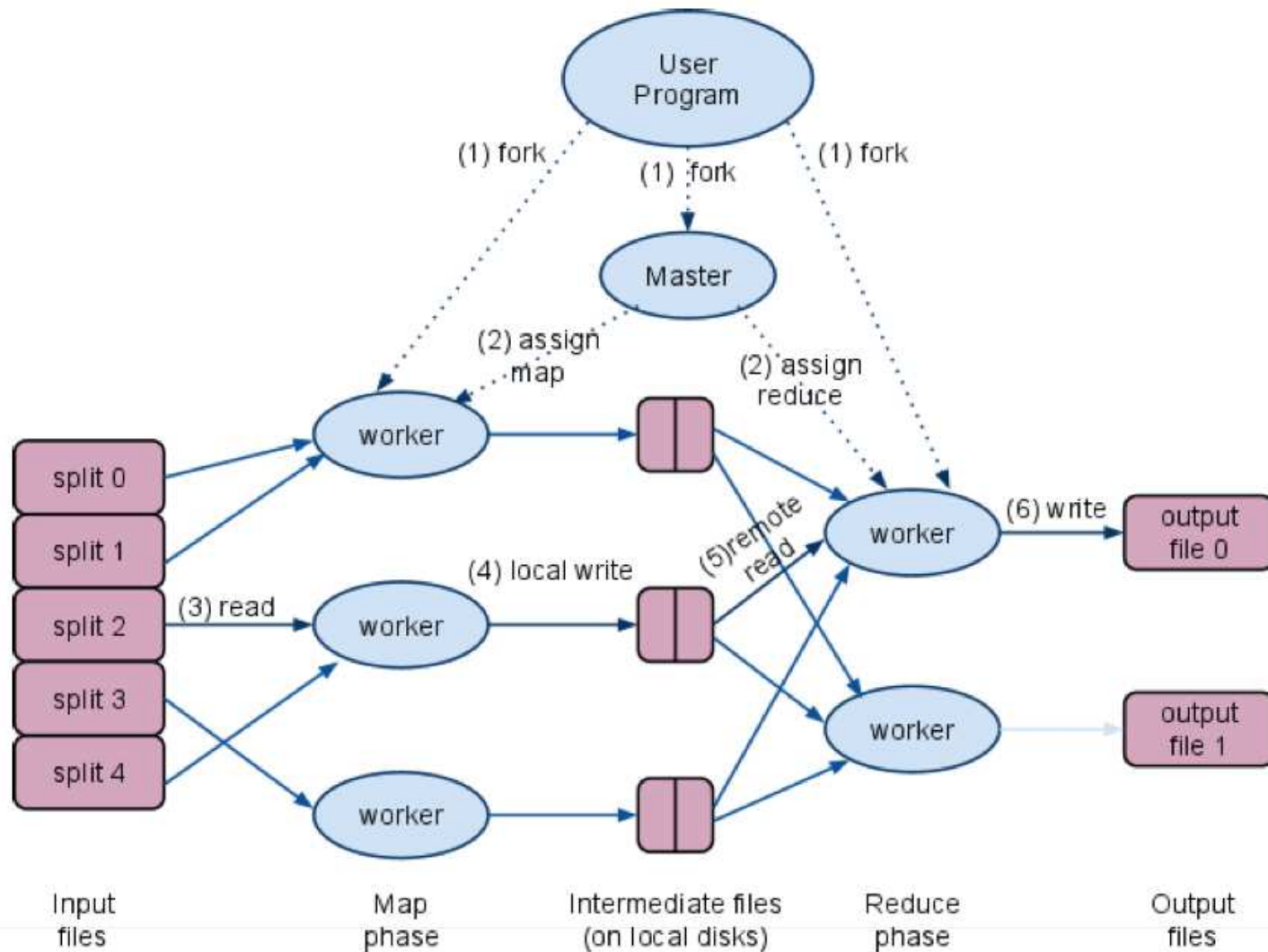
Column Oriented
(Multi-value sorted map)

id	Name
1	Ricky
2	Ankur
3	Sam

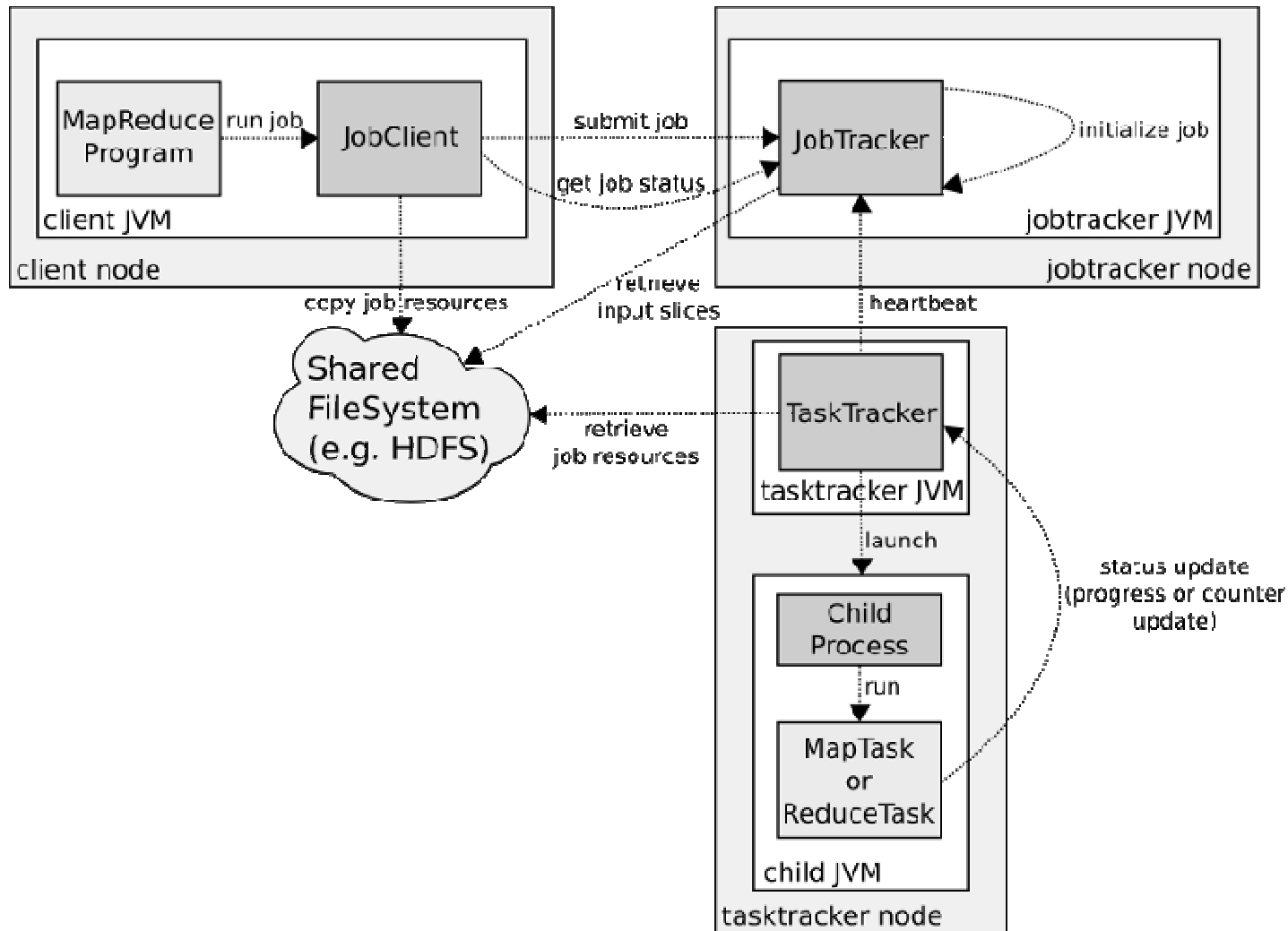
id	Age
2	20
3	25

id	Interests
1	Soccer
1	Movies
1	Baseball
3	Music

MapReduce execution



Hadoop[®] MapReduce execution



THE EXECUTION FRAMEWORK

- Scheduling
 - Speculative execution and backup tasks
 - an identical copy of the same task is executed on a different machine for solving some stragglers
 - skew in the distribution of values associated with intermediate keys (leading to reduce stragglers).
- Data/code co-location
- Synchronization
- Error and fault handling

THE EXECUTION FRAMEWORK

- Scheduling
- Data/code co-location
 - key ideas behind MapReduce is to **move the code, not the data**
 - scheduler starts tasks on the node that holds a particular block of data needed by the task
- Synchronization
- Error and fault handling

THE EXECUTION FRAMEWORK

- Scheduling
- Data/code co-location
- Synchronization
 - accomplished by a barrier between the map and reduce phases of processing
 - key-value pairs must be grouped by key, which is accomplished by **shuffle and sort**
- Error and fault handling

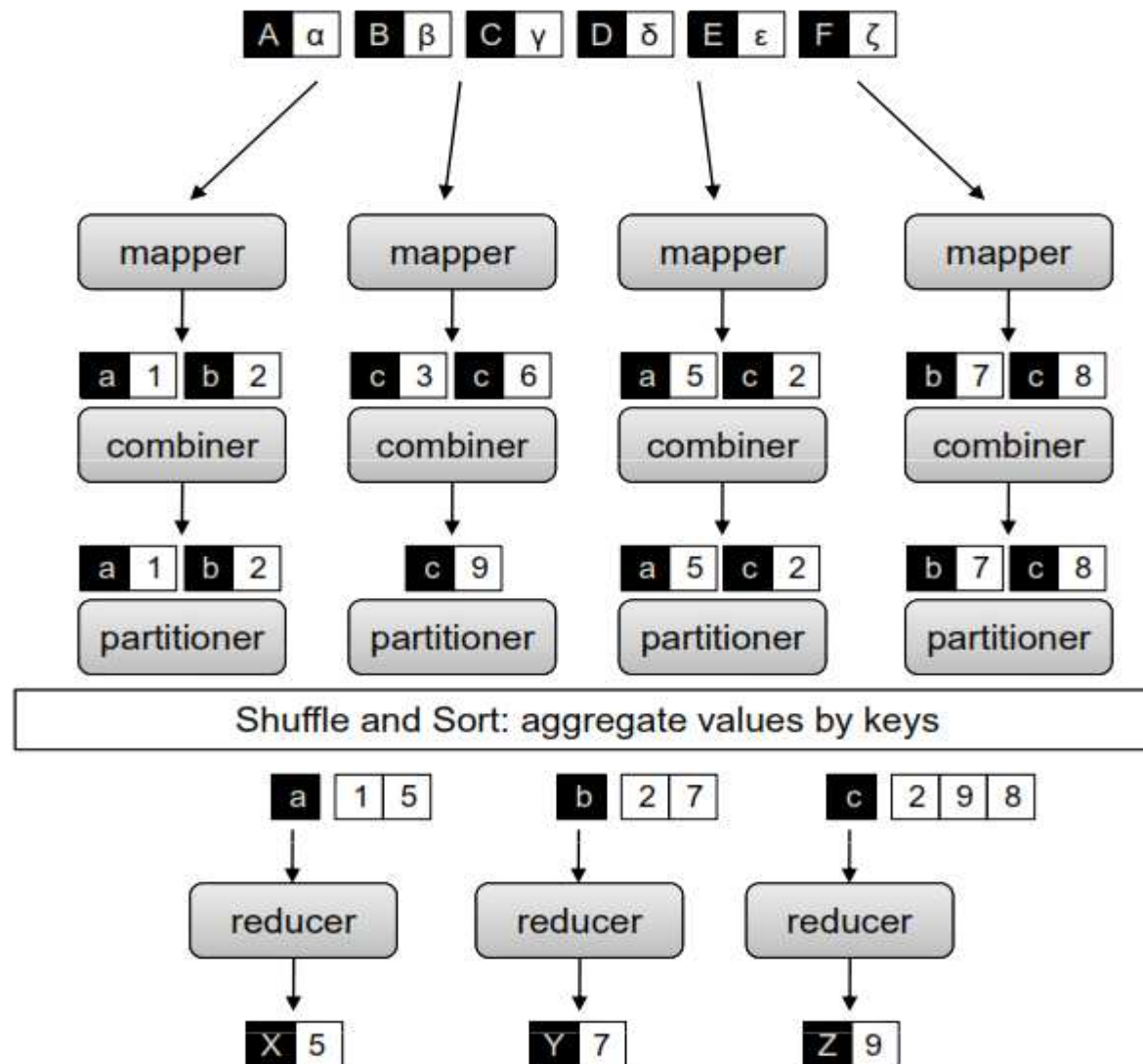
THE EXECUTION FRAMEWORK

- Scheduling
- Data/code co-location
- Synchronization
- Error and fault handling
 - disk failures | system maintenance and hardware upgrades | power failure, connectivity loss | No software is bug free | dataset will contain corrupted data

PARTITIONERS AND COMBINERS

- **Partitioners**
 - responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers
- **Combiners**
 - are an optimization in MapReduce that allow for local aggregation before the shuffle and sort phase
 - “mini-reducers” that take place on the output of the mappers prior to the shuffle and sort phase

Complete view of MapReduce



Word Frequency Example

- **Input:** Large number of text documents
- **Task:** Compute word frequency across all the document
 - Frequency is calculated using the total word count
- A naive solution with basic MapReduce model requires two MapReduces
 - MR1: count number of all words in these documents
 - **Use combiners**
 - MR2: count number of each word and divide it by the total count from MR1

Word Frequency Example

- **A nice trick:** To compute the total number of words in all documents
 - Every map task sends its total word count with key "" to ALL reducer splits
 - **Key ""** will be the first key processed by reducer
 - Sum of its values → total number of words!

Word Frequency Solution: Mapper with Combiner

```
map(String key, String value):  
    // key: document name, value: document contents  
    int word_count = 0;  
    for each word w in value:  
        EmitIntermediate(w, "1");  
        word_count++;  
    EmitIntermediateToAllReducers("", AsString(word_count));  
  
combine(String key, Iterator values):  
    // Combiner for map output  
    // key: a word, values: a list of counts  
    int partial_word_count = 0;  
    for each v in values:  
        partial_word_count += ParseInt(v);  
    Emit(key, AsString(partial_word_count));
```

Word Frequency Solution: Reducer

```
reduce(String key, Iterator values):  
    // Actual reducer  
    // key: a word  
    // values: a list of counts  
    if (is_first_key):  
        assert("" == key); // sanity check  
        total_word_count_ = 0;  
        for each v in values:  
            total_word_count_ += ParseInt(v)  
    else:  
        assert("" != key); // sanity check  
        int word_count = 0;  
        for each v in values:  
            word_count += ParseInt(v);  
        Emit(key, AsString(word_count / total_word_count_));
```

THE DISTRIBUTED FILE SYSTEM

- As **compute capacity grows**, the **link** between the compute nodes and the storage **becomes a bottleneck**
 - 10 gigabit Ethernet | InfiniBand
 - **abandon** the **separation** of **computation** and **storage** as distinct components
- distributed file system (**DFS**) that underlies MapReduce **adopts exactly second approach**

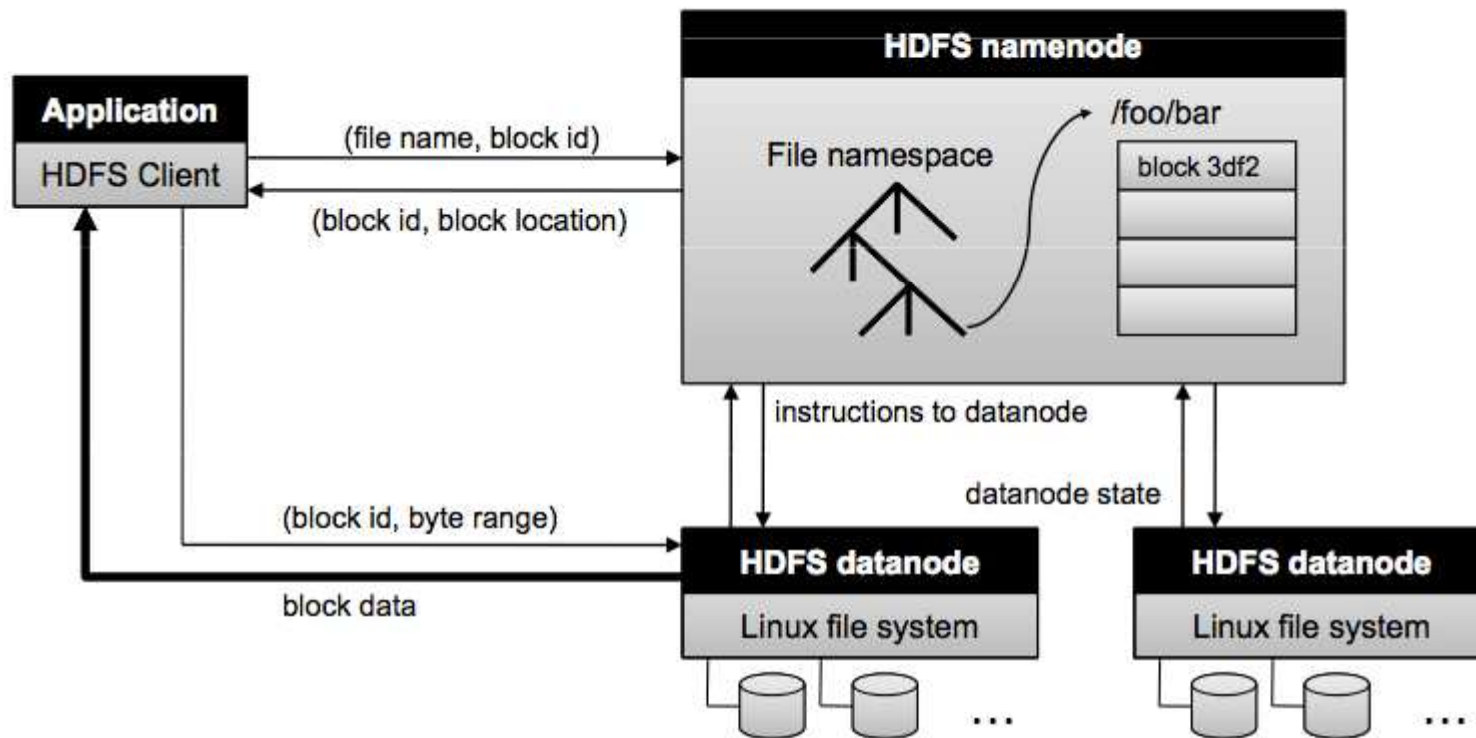
THE DISTRIBUTED FILE SYSTEM

- The **main idea** is to **divide user data** into **blocks** and **replicate** those blocks across the **local disks** of nodes in the cluster
- DFS adopts a **master- slave architecture**
 - Master
 - file namespace
 - metadata, directory structure, le to block mapping, location of blocks, and access permissions
 - Slave
 - manage the actual data blocks

Hadoop VS Google's DFS

implementation	DFS	Master	Slave
Google	Google File System	GFS master	GFS chunkservers
Hadoop	HDFS	namenode	datanode

architecture of HDFS



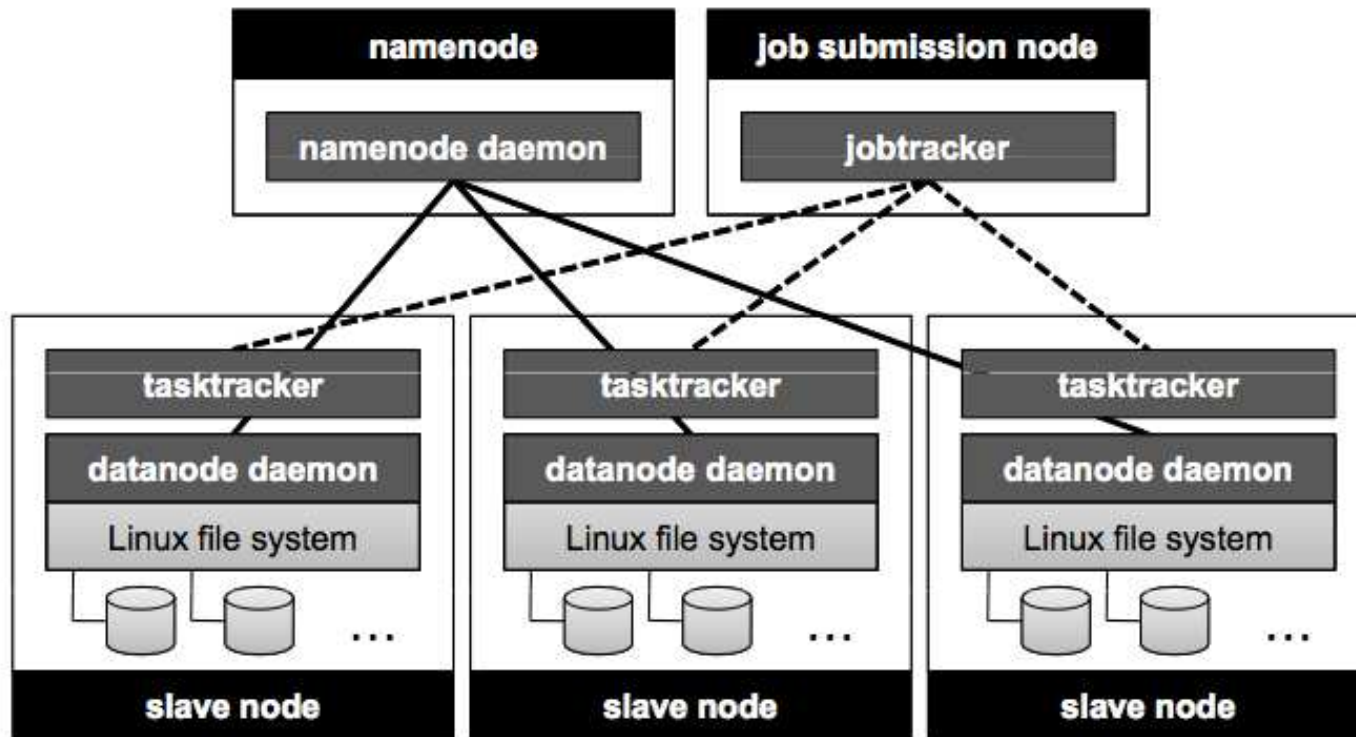
the HDFS namenode responsibilities

- Namespace management
- Coordinating file operations
- Maintaining overall health of the file system
- rebalancing the file system

Some critical point

- file system stores a relatively modest number of large files
- Workloads are batch oriented
- deployed in an environment of cooperative users.
- system is built from unreliable but inexpensive commodity components

HADOOP CLUSTER ARCHITECTURE



Average Income In a City

APPLICATION EXAMPLES

Average Income In a City

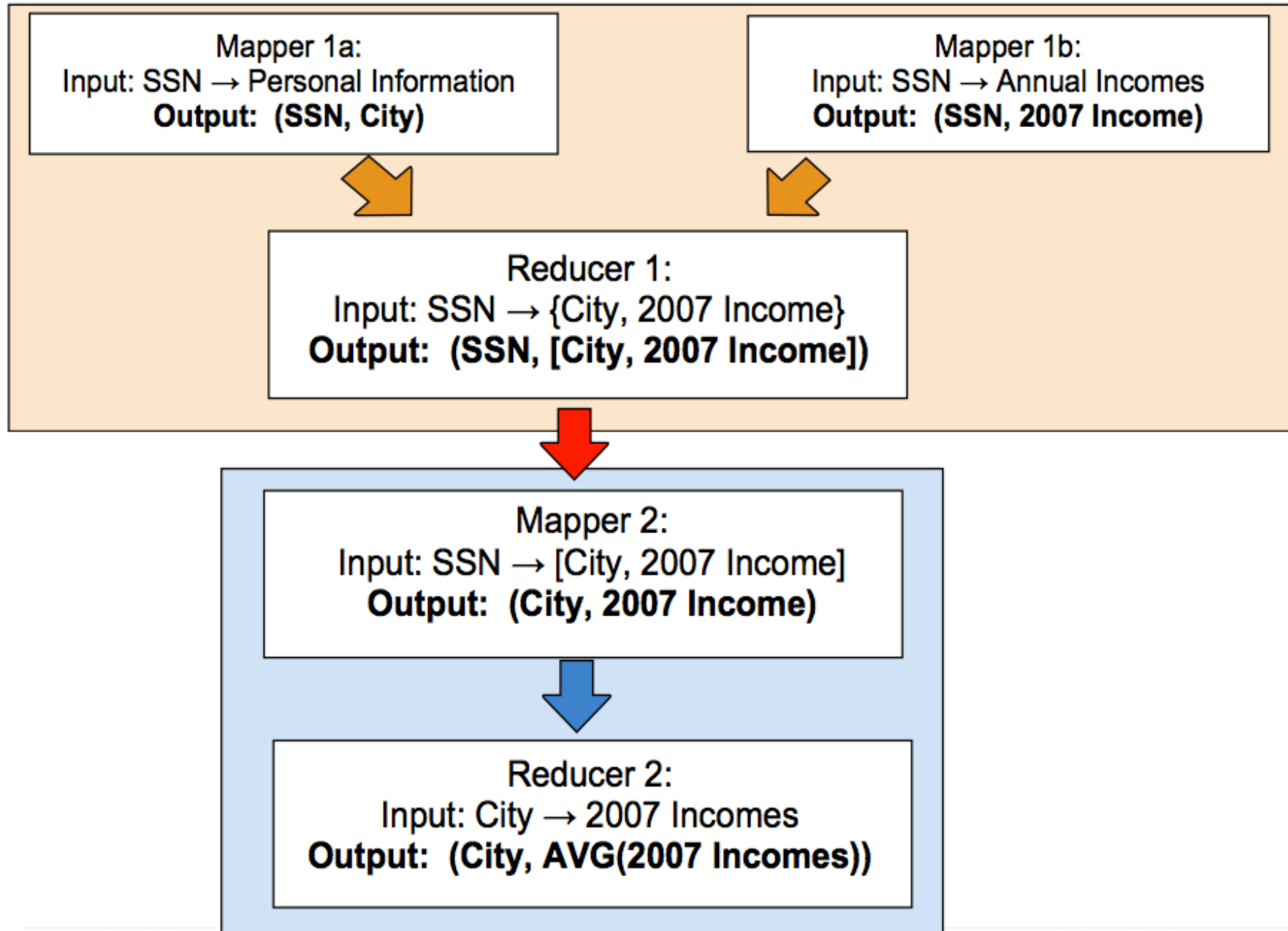
- SSTable 1: (SSN, {Personal Information})
- 123456:(John Smith;Sunnyvale, CA)
- 123457:(Jane Brown;Mountain View, CA)
- 123458:(Tom Little;Mountain View, CA)

- SSTable 2: (SSN, {year, income})
- 123456:(2007,\$70000),(2006,\$65000),(2005,\$6000),...
- 123457:(2007,\$72000),(2006,\$70000),(2005,\$6000),...
- 123458:(2007,\$80000),(2006,\$85000),(2005,\$7500),...

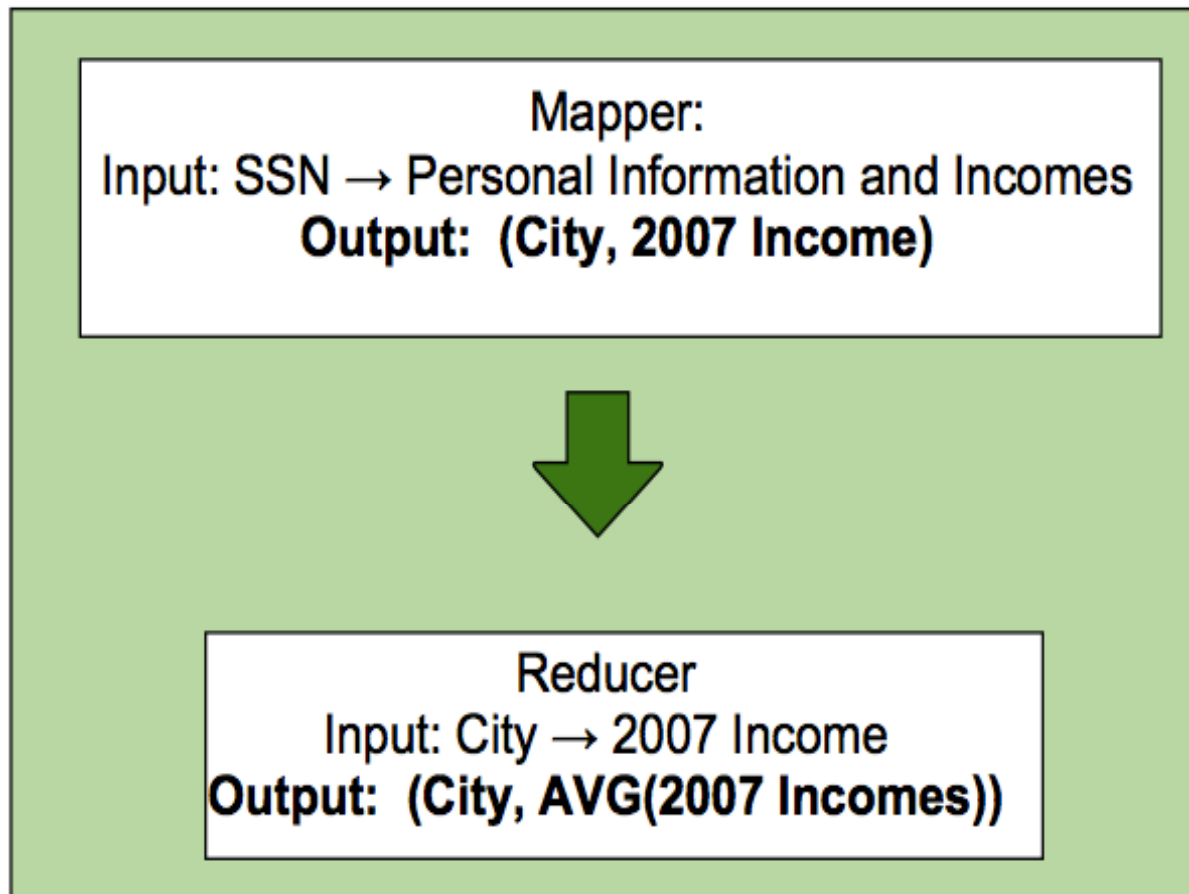
- Task: Compute average income in each city in 2007

- Note: **Both inputs sorted by SSN**

Average Income in a City Basic Solution



Average Income in a Joined Solution



Application Examples

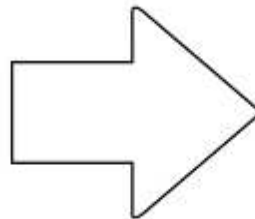
OVERLAYING SATELLITE IMAGES

Stitch Imagery Data for Google Maps



A simplified version could be:

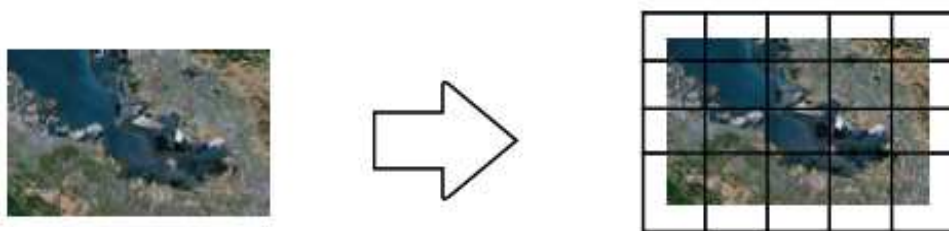
- Imagery data from different content providers
 - Different formats
 - Different coverages
 - Different timestamps
 - Different resolutions
 - Different exposures/tones
- Large amount to data to be processed
- Goal: produce data to serve a "satellite" view to users



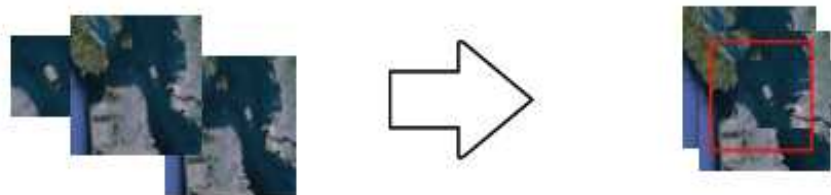
Stitch Imagery Data Algorithm



1. Split the whole territory into "tiles" with fixed location IDs
2. Split each source image according to the tiles it covers



3. For a given tile, stitch contributions from different sources, based on its freshness and resolution, or other preference



4. Serve the merged imagery data for each tile, so they can be loaded into and served from a image server farm.

Other application

- Inverted Indexing for Text Retrieval
- Graph Algorithms
- EM Algorithms for Text Processing
- Closing Remarks