Data Collection, Storage, Management, and Processing

(*centralized* and *distributed*)

GE461 - Introduction to Data Science

Spring 2025

Last update: Feb 16, 2025

Outline

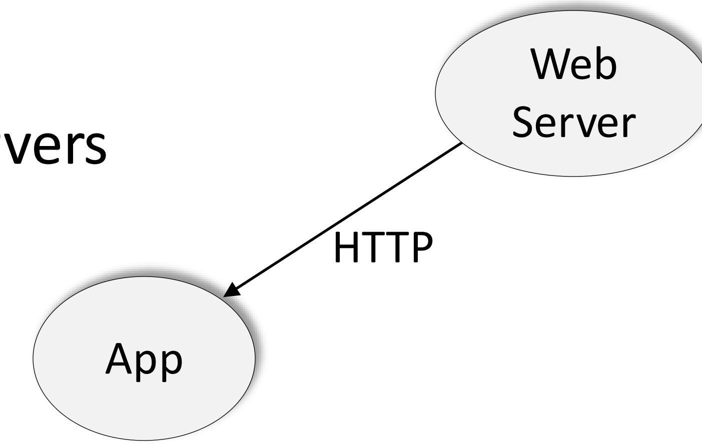
- Getting data
- Storing data
- Data management
- RDBMs and SQL
- Pandas
- Other data models
- Key-Value Stores and Column Stores
- Distributed Storage
- Parallel Processing frameworks
  - MapReduce
  - Spark

Mechanisms for Getting Data

Getting data

- We can download files manually (simply via a browser).
  - Various formats (txt, binary, CSV, JSON, XML, xls,…)
- We can write a program that scraps web.
  - Downloads pages and files reached via web links.
- A client program queries data from a database server (DB)
  - Program issues SQL requests to a DB server.
- A client program queries an API (usually web based API)
  - REST API is a common web-based API
  - SOA (service oriented architecture) is another alternative
  - Source of data can be a DB server or some other program

Web scraping: HTTP queries



- We can download pages from web servers
- Underlying protocol is HTTP

- Below is a python code

```
import requests
response = requests.get("http://w3.cs.bilkent.edu.tr")
# some relevant fields
print (response.status_code)
print (response.content)  # or response.text
print (response.headers)
print (response.headers['Content-Type'])
```

Page address (URL)

Page is downloaded to local disk

Web scraping: HTTP queries – Parameters

- Uses the GET method of the HTTP protocol
- A URL can have parameters
  - http://www.google.com/search?**q**=bilkent&**num**=5
  - **q** and **num** are parameters

- In python:

```
plist = {"q": “bilkent”, "num": “5”}    # parameter list
resp = requests.get("http://www.google.com/search” , params=plist)
print (resp.status_code)
print (resp.content)
```

Web API: HTTP commands

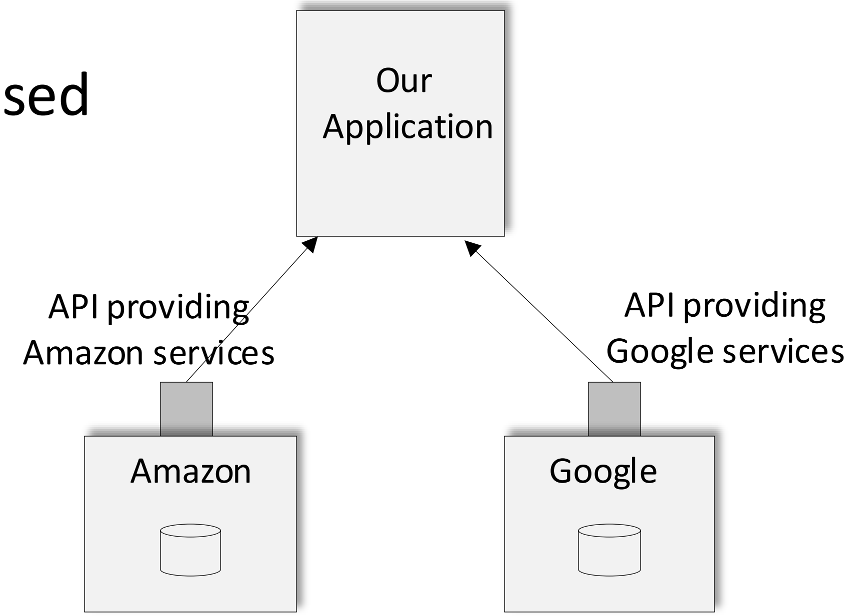
- We can query *web services* via **Web API** and get data.

- HTTP commands (methods) used
  - GET is the most common
    - URL specified
  - But there are other HTTP methods that can change some state on the server

    HTTP POST

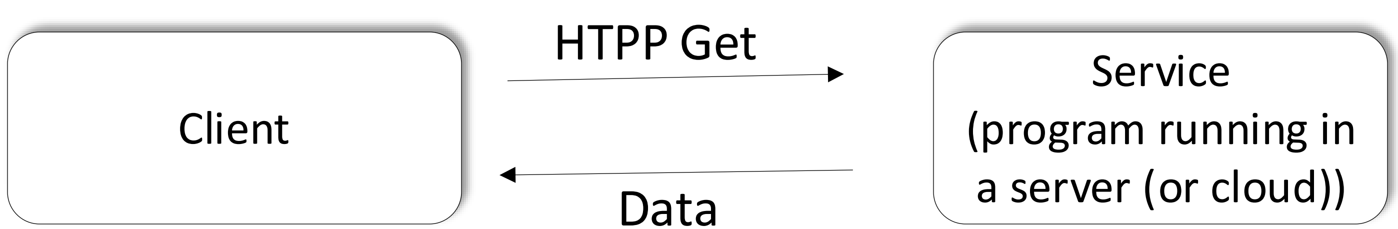
    HTTP PUT

    HTTP DELETE



Web API

- There are web APIs for a lot of web Services
- Web Services: applications running in remote servers (cloud) and accessed via web servers.
- The *service* should be *programmed to provide an API.*
- *REST* is one such *API standard*
    - *REST: representational state transfer*



REST

- REST is a commonly used API standard.
- Set of rules that developers follow when they create their APIs.
- It is a *simple* *architecture style* to transfer data (resources) over HTTP (offer services over web).
  - 1. Uses standard HTTP interface and methods (GET, PUT, POST, DELETE)
  - 2. Stateless – the server does not remember what is done previously (stores no state).

REST

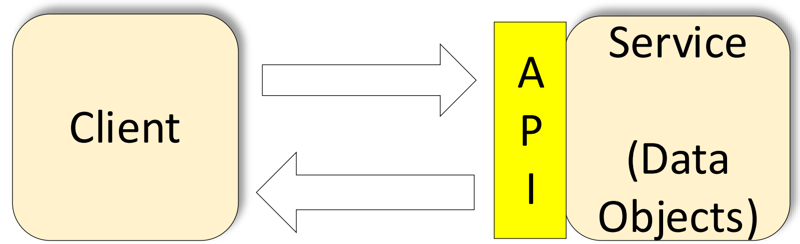
- You query a REST API with standard HTTP requests
  - You include parameters in the query.
- For example, GitHub API uses GET/PUT/DELETE to let you query or update elements in your GitHub account.
- A service that provides REST API: Restful service.

REST key elements

- *Resources* (and *URI*)
  - Data objects
- Request *Verbs*
  - What to do with data
- Request *Headers*
  - Additional instructions
- Request *Body*
  - Data
- Response *Body*
  - Data

We identify our **resources** with **URIs.**

We map them (URIs) to service endpoints (request handlers).



We write code to process GET, PUT, POST, DELETE (service handlers)

RESTful Service
an example

Request (GET, PUT, or Post)
Incudes URI

REST
Client

Response
(JSON, XML, or HTML)

REST request handler
Direct the GET/PUT request to correct service handler for the URI

REST service handler
Process request and prepare JSON response

Database handler
Access database if needed

Write in PHP for example

- Example: A server stores information about mobile phones. We can query the information.

Data Format: JSON

- JSON: JavaScript Object Notation
- Open-standard file and data format
  - For storing data
  - For exchanging data
- Uses human-readable text to transfer data objects
  A data object consists of attribute-value pairs or array data types

```json
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

https://en.wikipedia.org/wiki/JSON

XML

- XML: Extensible Markup Language
  - For representing data
  - For storing data
  - For exchanging data
- XML defines a set of rules for encoding documents and data in a format that is both human readable and machine-readable.
- Textual data format
- Allows to define your own custom tags

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Do not forget exercising</body>
</note>
```

https://en.wikiversity.org/wiki/XML

Structure of the data

- Structured data (has *schema* describing the structure)
  - Schema: defines the structure/organization of the data
  - Database (DB) Tables.

- Semi-structured data
  - Does not have a strict scheme, no rigid structure.
  - Documents
  - XML, JSON

- Unstructured data
  - Text files, plain text, media (images, videos).

Storing and Retrieving Data

Databases and Data Management Systems

- Database: A collection of data
- Database Management System
  - Software that stores, manages and facilitates access to data. (Oracle, MySql, Sqlite, …).
- Traditionally: relational database systems.
  - Supports transaction processing, concurrency, reliability, recovery….
  - Bank accounts, student records, customer records, inventory records, ….
- Modern needs and usage varies (NoSQL databases, etc.)
  - Hadoop, Spark.
  - Cloud databases.

File system

- We can store data in files.
- This may be good enough for a lot of applications.
  - But not all applications.
- File system is not a database
  - Two people (processes) accessing a file may cause inconsistency.
  - Sudden power off may cause loss of data.
  - No query support
  - No transaction (ACID) support.

Relational DBMSs and SQL

Relational Database

- Models a real world data environment
  - Entities (students, courses, instructors)
  - Relationships (taking the course, giving the course, is advisor of, etc.)
- RDMBs work with tables (relations)
  - Relation: a table (with rows and columns)
  - Schema: describes columns, fields.
- A *table (also called a relation)* stores information about objects or relations of the same kind (same set of attributes)
  - Rows are called tuples (records); must be unique
  - Columns are attributes

Table

attributes

Student

ID	Name	Dept	CGPA
1	Ali	CS	3,50
2	Veli	CS	3,20
3	Ahmet	CS	3,80

tuples

- Rows (tuples). A relation is a set of tuples.
- Columns (attributes)
- Relation (Table) name is Student.
- It has 4 attributes
- It has 3 tuples.
- These 3 tuples are an instance of the Student Relation.

Multiple Tables

- A database typically has multiple tables.
- Student table,
  Course table,
  Department table,
  Instructor Table,
  Offerings table,
  Enrollment table, ..

Course

ID	Name	Dept	Credits
CS342	Operating Systems	CS	4
GE461	Data Science	GE	3
EEE202	Circuit Theory	EEE	4
CS202	Data Structures	CS	3
IE202	Optimization	IE	3
ME101	Mechanical Systems	ME	4

Schema

- Schema for a database describes the tables and their attributes.
- It is fixed.
- It is the logical design.
- It is then populated with data (instances).
- Data + Schema = Database.

Schema

- Example Schema
  - Department (id, name, building)
  - Student (id, name, dept, CGPA)
  - Course (id, name, dept, credits)
- Some tables are for objects: Student table
- Some tables are for relations: Enrollment

Keys

- Primary Key: the attributes used to identify tuples in a table uniquely
- Foreign Key: an attribute in a table that is the primary key in another table.

Course

Foreign key

Primary key

ID	Name	Dept	Credits
CS342	Operating Systems	CS	4
GE461	Data Science	GE	3
EEE202	Ciruit Theory	EEE	4
CS202	Data Structures	CS	3
IE202	Optimization	IE	3
ME101	Mechanical Systems	ME	4

Primary key

Department

ID	Name	Building
CS	Computer Science	EA
EE	Electrical Engineering	EE
IE	Industrial Engineering	EA
ME	Mechanical Engineering	EA
MATH	Mathematics	SC

Query Language

- Query language is language to request information from a database
- *Procedural* or *declarative*
- SQL : structured query language (declarative)
  - Most common, but not the only one.

Query Language

- Can be used to
  - Create / delete a database (data definition)
  - Create / delete a table (data definition)
  - Insert, delete, update tuples (data manipulation)
  - Query table(s) (retrieve data) (data manipulation)
    - Select some set of tuples from a table
    - Join multiple tables

SQL

- SQL has two main parts:
  - DDL (data definition language);
  - DML (data manipulation language)
- Supported data types
  - char(n)
  - varchar(n)
  - int
  - real, float(n)
  - …

SQL

- CREATE TABLE Department (id varchar(20),
  name varchar(20),
  building varchar(20),
  primary key (id));

- CREATE TABLE Student (id int,
  name varchar(20),
  dept varchar(20),
  cgpa float,
  primary key (id),
  foreign key (dept) references Department;

- INSERT INTO Student VALUES (4, ‘Can’, ‘CS’, 3,75);

SQL

- To retrieve data from a table or from multiple tables, we can form and execute SQL queries.

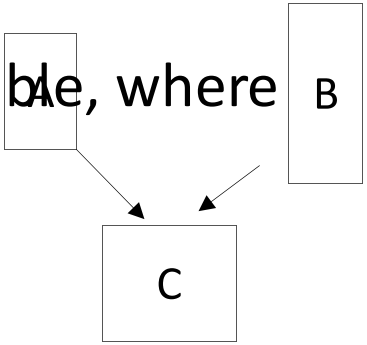
- Basic structure for SQL queries:

  SELECT <columns> FROM <tables> WHERE <predicate>

- SELECT name FROM Course
- SELECT dept FROM Course
- SELECT name, dept FROM Course
- SELECT name FROM Course WHERE dept == ‘CS’

Joins

- Merge information in multiple tables together.
- Join operation merges multiple tables into a single table/relation (can be then saved as a new table or just directly used)
- You join two tables on columns from each table, where these columns specify which rows are kept.
- There are different types of joins:
  - Inner
  - Left (outer)
  - Right (outer)
  - Full (outer)



Example: joining instructor and department

Instructor

ID	Name	Dept	Title
id101	Cem	CS	C
id102	Mustafa	CS	A
id103	Emre	EE	B
id103	Ayse	CS	A
id105	Ozgur	IE	C
id106	Dilek	ME	A
id107	Ahmet	POLS	B
id108	Atakan	IR	C
id109	Remzi	PSYC	A

Department

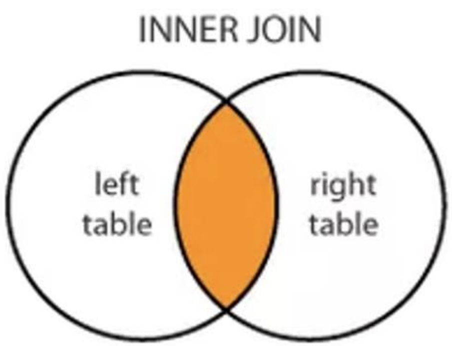
ID	Name	Building
CS	Computer Science	Building-X
EE	Electrical Engineering	Building-X
IE	Industrial Engineering	Building-X
ME	Mechanical Engineering	Building-X
MATH	Mathematics	Building-Y
PHYS	Physics	Building-Y
ECON	Economy	Building-Z

Example: joining instructor and department

SELECT * FROM Instructor INNER JOIN Department
ON *Instructor.dept* == *Department.id*;

Or

SELECT * FROM Instructor, Department
WHERE *Instructor.dept* == *Department.id*;



Resulting relation (can be used or can be saved)

ID	Name	Dept	Title	Name (Department)	Building
id101	Cem	CS	C	Computer Science	Building-X
id102	Mustafa	CS	A	Computer Science	Building-X
id103	Emre	EE	B	Electrical Engineering	Building-X
id103	Ayse	CS	A	Computer Science	Building-X
id105	Ozgur	IE	C	Industrial Engineering	Building-X
id106	Dilek	ME	A	Mechanical Engineering	Building-X

INNER JOIN: only matching rows included. Unmatched rows are not included.

SQL Lite

- SQLite: an actual relational database management system (RDBMS)
- Unlike most systems, it is a server-less model, applications directly connect to a file.
- Allows for simultaneous connections from many applications to the same database file (but not quite as much concurrency as client-server systems).
- All operations in SQLite use SQL (Structured Query Language) commands issued to the database object.

Client-Server DBMS vs Serverless DBMS

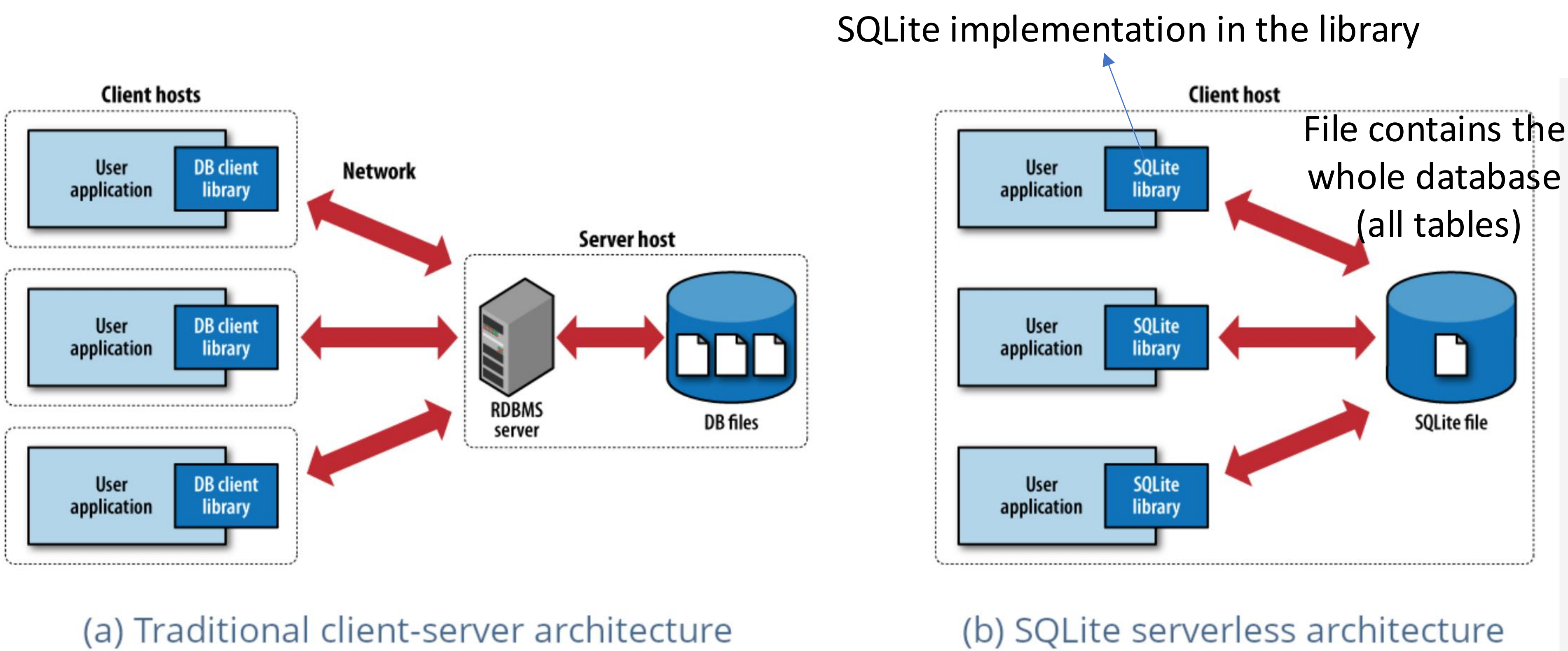


Figure from : developia.org/sqlite

Client – Server Architecture
For example: MySQL server

Serverless DBMS
For example: SQLite

Use of SQL in Python

```
import sqlite3

 conn = sqlite3.connect('ders.db') / # create or open db
 c = conn.cursor()  # obtain a handle to the connection

query = "CREATE TABLE Student (id varchar(10) \
 PRIMARY KEY, name varchar(20), dept varchar(10), \
 cgpa REAL NOT NULL);"

c.execute(query)
conn.commit()

query = "INSERT INTO Student VALUES (?, ?, ?, ?);”
c.execute(query, ‘2222’, ‘Ali’ , ‘CS’, ‘3.5’))
conn.commit()
```

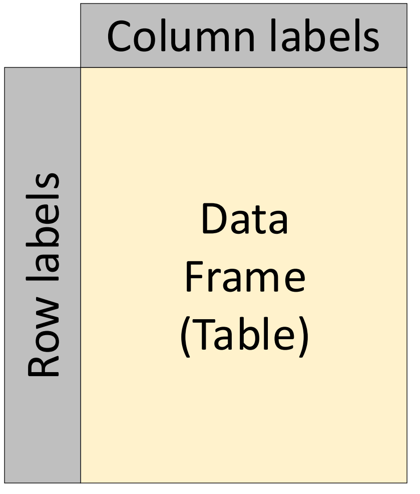
SQL in Python

```
query = "SELECT * FROM Student;”
c.execute(query)

rlist = c.fetchall()    # fetch the rows into a list
for i in range(len(rlist)):          # print the list
      print (rlist[i][1])            # one row at a time


query = "SELECT * FROM Student WHERE Student.dept == ‘CS’ ;”
c.execute(query)

query = "SELECT * FROM Instructor, Department WHERE   \
      Instructor.dept == Department.id;”          # JOIN
c.execute(query)
```

Pandas

- Pandas is a "Data Frame" library in Python, developed for manipulating in-memory data with row and column labels (as opposed to, e.g., matrices, that have no row or column labels)

- Pandas is not a relational database system, but it contains functions that mirror some functionality of relational databases. For example: merge mimics join.



Important data structures of Pandas

- Series:
  - Array (of objects of the same type) (1D)
  - Homogenous array that can be indexed.
- DataFrame:
  - Table structure (2D)
  - Columns
  - Column types can be different
  - For one column: all values are of the same type (a Series)

Pandas

- Fast and efficient DataFrame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.

From: https://www.tutorialspoint.com/python_pandas/

Pandas

- Label-based *slicing*, *indexing* and *subsetting* of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality.

Pandas

```
import pandas as pd

df = pd.DataFrame([('id1', 'Ali', 'CS', '3.4'),
('id2', 'Ahmet', 'EE', '3.3'),
('id3', 'Ayse', 'IE', '3.7'),
('id4', 'Begum', 'ME', '3.5'),
('id5', 'Mehmet', 'CS' '3.5'),
('id6', 'Ramazan', 'EE', '3.6')],
columns=["Stu ID", "Name", "Dept", "CGPA"])
```

// Column Labels

```
print (df)
```

Column index

Row index

	Stu ID	Name	Dept	CGPA
0	id1	Ali	CS	3.4
1	id2	Ahmet	EE	3.3
2	id3	Ayse	IE	3.7
3	id4	Begum	ME	3.5
4	id5	Mehmet	CS	3.5
5	id6	Ramazan	EE	3.6

Pandas

- Pandas is not RBMS, no primary key concept
- It has index concept.
- Operations in Pandas are typically not in place (that is, they return a new modified DataFrame, rather than modifying an existing one; by default)
- We can use the "inplace" flag to make them done in place
- If we select a single row or column in a Pandas DataFrame, it will return a "Series" object,
- A Series object is like a one-dimensional indexed array (sequence of values and their indices).

Pandas: some data frame methods

df.head(): some number of rows from beginning.

df.tail(): some number of rows from end.

df.iloc[i,j]: access the entry (value) at the ith row and jth column

x = df.iloc[0,1] // will access "Ali". [0,0] will access "id1".

df.loc[rowindexlabel, columnindexlabel]: access the entry at the specified row and column

x = df.loc[3, "Dept"]

will access "ME"

	Stu ID	Name	Dept	CGPA
0	id1	Ali	CS	3.4
1	id2	Ahmet	EE	3.3
2	id3	Ayse	IE	3.7
3	id4	Begum	ME	3.5
4	id5	Mehmet	CS	3.5
5	id6	Ramazan	EE	3.6

Other Data Models
and
Big Data

Other Data Models

- RDDMS is good for storing transactional and/or structured data.
  - Bank account data
  - Employee data
  - Student data
- New classes of data intensive applications
  - Search
  - Email
  - Browsing
  - Instant messaging
  - Social media
  - Online retail
- NoSQL databases (not only SQL)

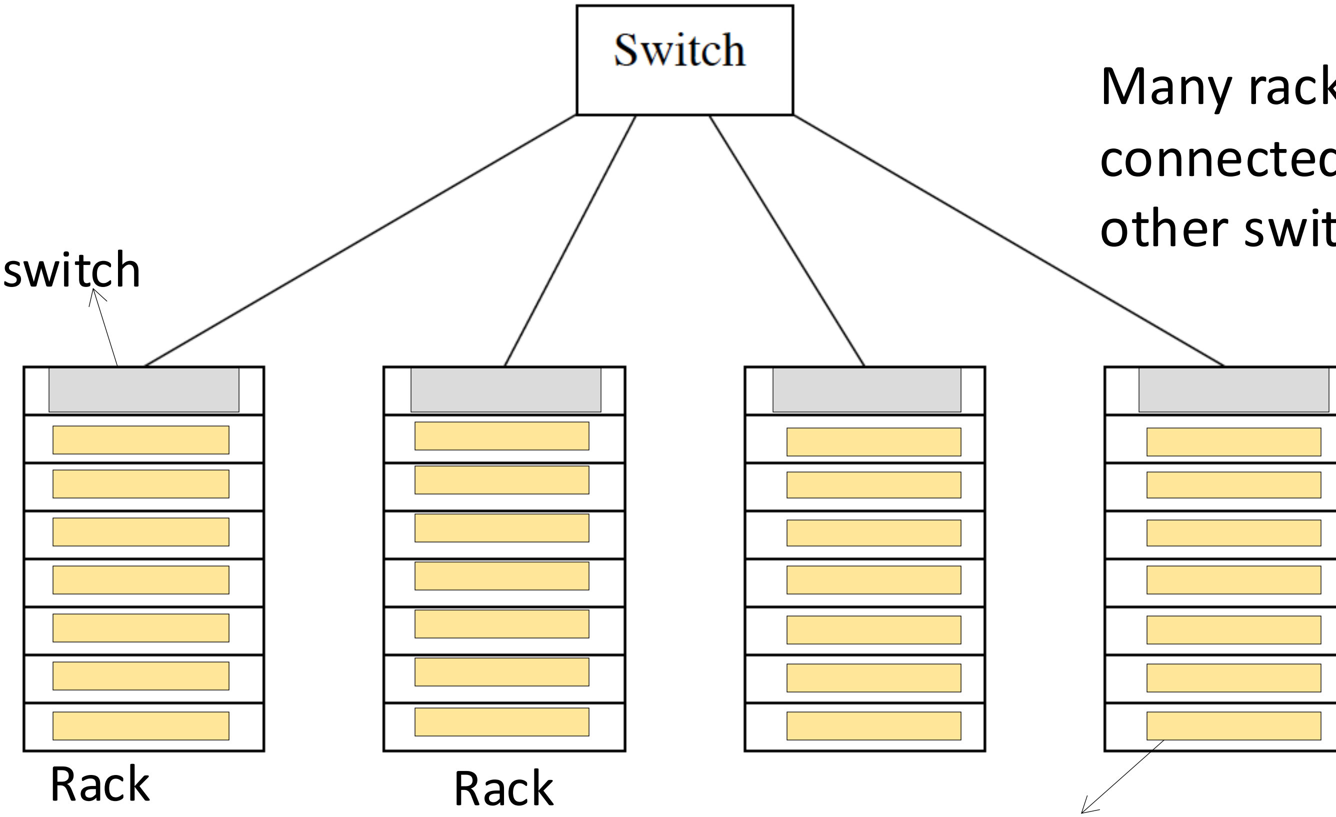
Big Data

- For non-big data:
  - Singe machine solutions are good.
- For big data (TeraBytes, PetaBytes of data), a single computer/server will not provide enough storage capacity, with acceptable reliability and performance.
- We need a cluster of machines to store and process big data.
- How can we store and process data in a cluster?

What is a Cluster?

Many racks connected by other switches

ToR switch



Rack

Rack

a Computer/Server
(Compute Node)
with local storage

Many servers in a rack.
Connected with a switch.

Compute node: processor(s), with its main memory, cache, and local disk (storage)

Distributed File System (DFS)

- To exploit cluster computing, files must look and behave somewhat differently from the conventional file systems found on single computers (Linux FS, NTFS, FAT32 are local file systems).
- This new file system, often called a distributed file system or DFS is typically used as follows.
  - Files can *be enormously big*, possibly terabytes in size.
  - Files are *rarely updated*. They are mostly read. New data is appended from time to time.
  - A single file's content is stored in multiple computers and is also replicated.
- Example: *HDFS* (Hadoop File System) or GFS (Google File System).

Data Stores
Key-Value Stores

- Key/Value Stores (NoSQL)
  - Can store very large data
  - Key-value sets stored
    - Example: customer id, purchased items, date.
  - Performance is critical
  - Eventual consistency is fine.
  - No fancy reports.
  - Data analysis and recommendation
  - Query set depends on the application
  - Just keys and values, no schema
- Example systems:
  - Amazon Dynamo DB.
  - Apache Cassandra.

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

From wikipedia

Other data stores:
Column Family Stores

- A big table of rows and columns (billions of rows, billions of columns possible): sparse
- Columns are grouped into Column Families
- Column Families:
  - Typically stored together (physically)
  - Can have *different columns for each row*
  - Can have duplicate items in any column
- No schema or type enforcement
  - All data treated as byte strings
- Indexed by row (***row key***)
  - Rows are grouped into tablets (chunks)
- Rows usually kept in *sorted order wrt row key*

Example: Google BigTable

Other data stores: Column Family Stores

Data Model: Column Family (2)

Column Families

Timestamps

Keys	Name		Address					
1	*First* Margo	*Last* Seltzer	*No* 3	*Street* Millstone Lane	City Lincoln	*State* MA	*Zip* 01773	2000
			No 394	*Street* East Riding Dr	*City* Carlisle	*State* MA	*Zip* 01741	1993
			PO 65		*City* Sonyea	*State* NY	*Zip* 14556	1961
3	*Title* Lady	*Last* Gaga	*City* Hollywood			*State* CA	*Zip* 90027	2008
4	*Last* Madonna		*New York*		*Los Angeles*		*London*	2000

versions

from: CS109 Harvard

How data internally stored

Logical View

CF1 CF2

	C1	C2	C3
R1	X (t3, t2, t1)		
R2		X	X
R3	X (t1)		X
R4	X (t2, t1)	X	
R5			X

Table

X denotes an existing value

Ri is a row key (string)

CFi: is a column family name

Ci is a <u>column name </u>(string) (also called <u>column key</u>)

Physical View

CF1

R1 CF1:C1 t3 X
R1 CF1:C1 t2 X
R1 CF1:C1 t1 X
R3 CF1:C1 t1 X
R4 CF1:C1 t2 X
R4 CF1:C1 t1 X

CF2

R2 CF2:C2 t1 X
R2 CF2:C3 t1 X
R3 CF2:C3 t1 X
R4 CF2:C2 t1 X
R5 CF2:C3 t1 X

This is how data can be stored internally in two files.

How data internally stored

- Bigtable cells which do not contain a value consume no disk space.
  - Sparse table.
- For each valid cell value, we store *both* the row key and the column name.
- For each cell, we can keep different versions of cell data (time stamped).
- To learn which column names are there in the table, we have to do a full scan of the table. Schema just gives created column families, not column keys.
- For each key-value pair, we keep the associated lengths as well.
  - *key length, value length* (both *variable size*).

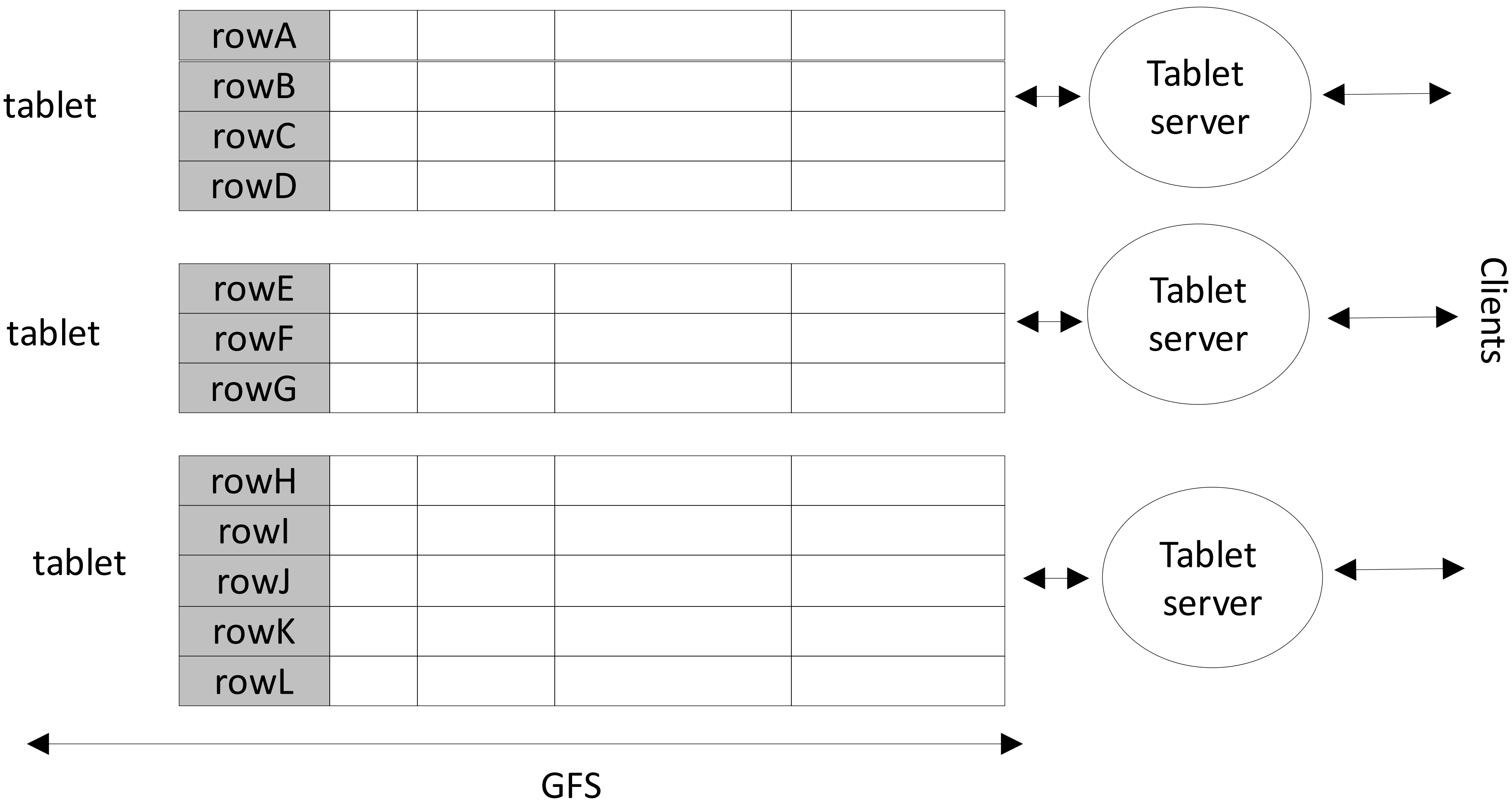
KeyLen	ValueLen	Key	Value

Table and Tablets

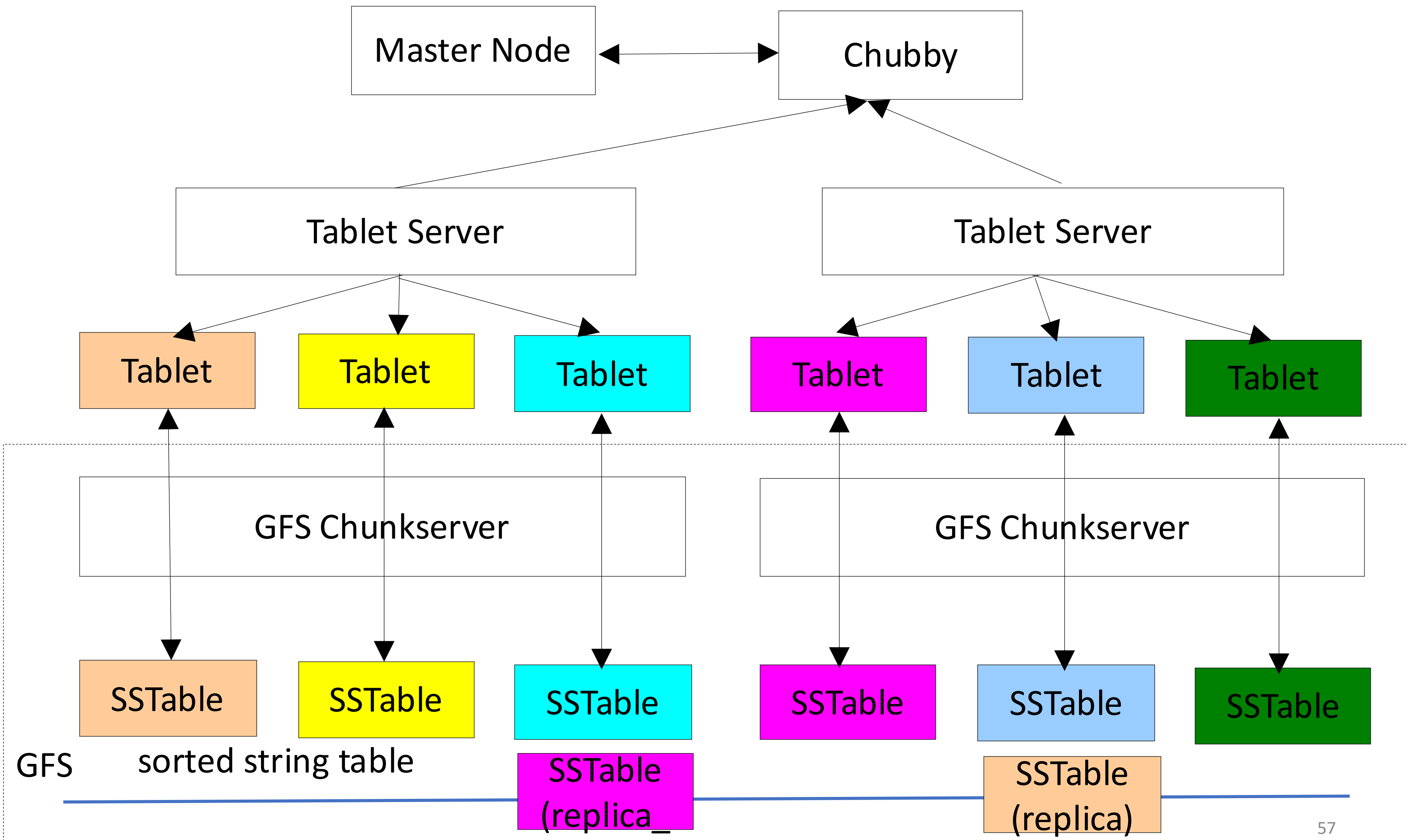
	rowA				
tablet	rowB				
	rowC				
	rowD				
	rowE				
tablet	rowF				
	rowG				
	rowH				
	rowI				
tablet	rowJ				
	rowK				
	rowL				

Rows are kept always in sorted order wrt row key

Table and Tablets

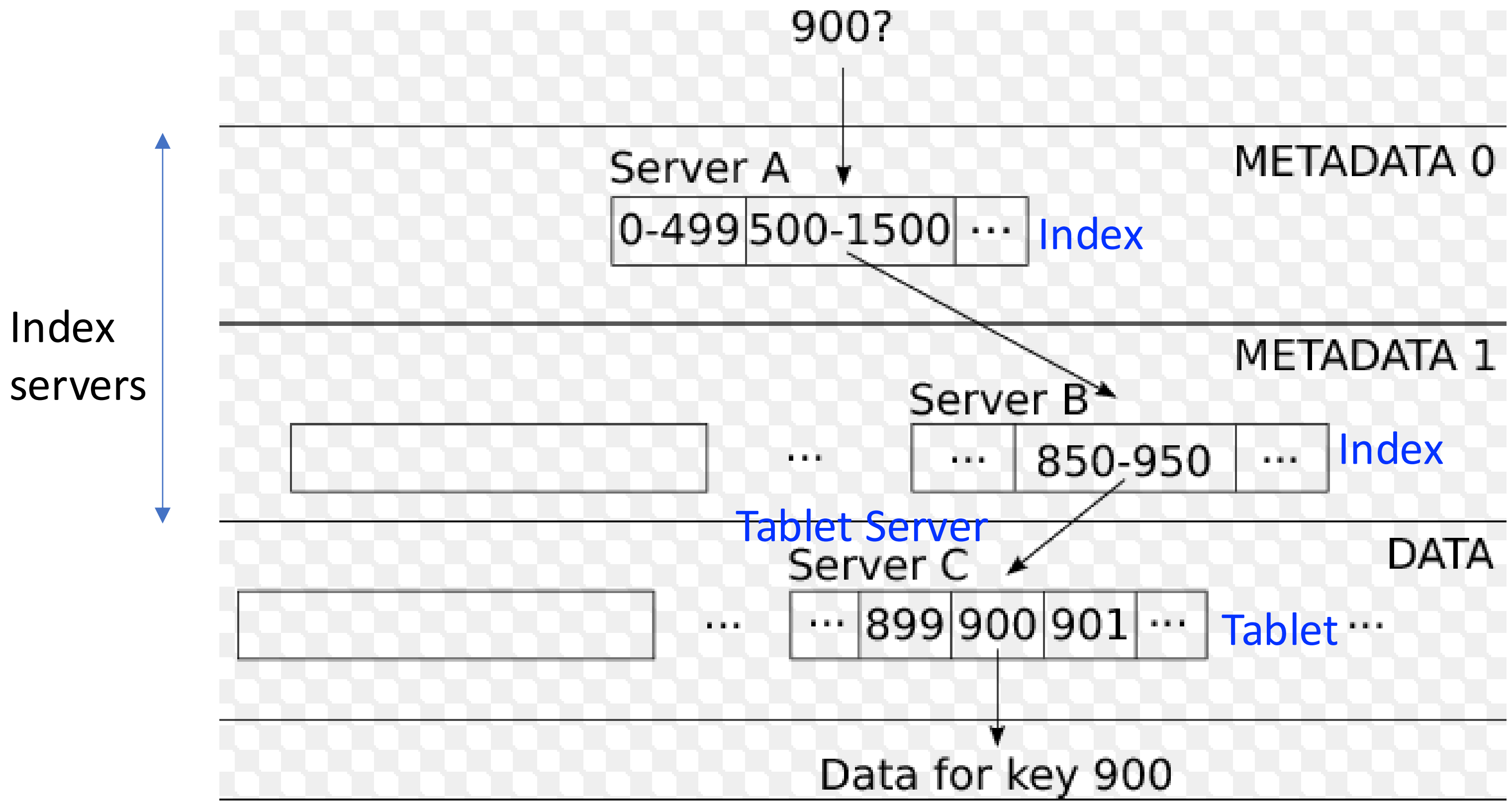


BigTable Architecture



Locating tablets and data

Example: locating data with row key = 900



Document Stores

- A Key/Value store where value is a document with structure
- Structures for documents:
  - JSON
  - XML
  - PDF
  - DOC
- Search for and within documents possible.

MapReduce

Distributed Big Data Processing

- Big Data is distributed on many machines
  - Local *processing* preferable, but not always sufficient and possible.
- MPI was used in the past
  - Explicit data handling.
- New frameworks are available to process data.
- MapReduce Framework (Google, Hadoop)
  - Distributed data storage file system (GFS or HDFS)
  - Distributed big data table (BigTable or HBASE)
  - Distributed processing language/framework (MapReduce)
- Spark Framework

MapReduce Framework

- **MapReduce**:
  - A programming model and associated implementation for processing and generating large datasets.
- Hadoop system has it as its programming model.
  - Hadoop system has also a file system (HDFS) and a NoSQL database system (Hbase).
- An application specifies a map() function and a reduce() function for a computation to be done.
- Many real-world tasks expressible with this model.
- A program written with this model is automatically parallelized and executed by the Framework <u>on a large cluster of machines.</u>

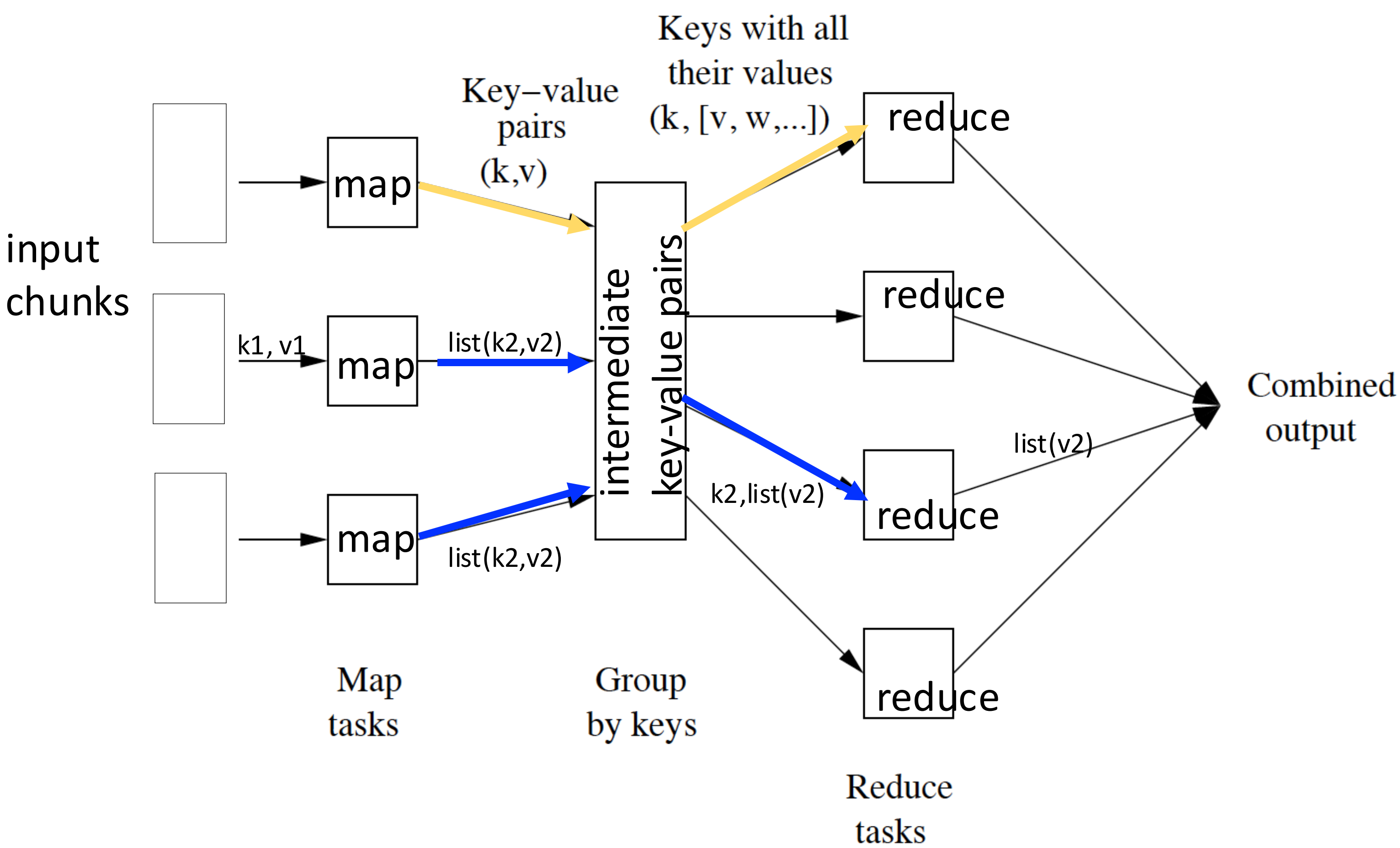
Programming Model

- Computation
  - Input: A set of input key/value pairs
  - Output: a set of output key/value pairs
- User of **MapReduce library** specifies
  - a map() function
  - a reduce() function

Programming Model

- Map function:
  - Takes: an input key/value pair (e.g., doc-name, doc-content)
  - Produces: a set of intermediate key/value pairs
  - All intermediate values with the same intermediate key are grouped.
- Reduce function:
  - Takes: an intermediate key and a set of values associated with that
  - Produces: a smaller set of values resulting from the merging of all the values associated with the key (for example, sum, count, etc.).

INPUT DATA



map `(k1,v1)` → `list(k2,v2)`
reduce `(k2,list(v2))` → `list(v2)`

Example: word-count
counting words in a set of documents

map() and reduce() functions below

```
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));
```

Programming

- We write an application program in which
  - We write map() and reduce() functions
  - Specify the input files
  - Specify the number of map workers (machines) (N)
  - Specify the number of reduce workers (machines) (R)
  - Specify output files
- Framework will do the rest (parallel processing)
  - Partition the input into M splits (for M map-tasks)
  - Handle each split via the map() as a task
  - Schedule tasks to machines (workers)
  - Sort at the reduce-workers before the reduce()
  - Reduce and write the results to output files (sorted order)

Application Examples

- Distributed Grep:
  - Map() function emit a line if it matches a supplied pattern
  - Framework sorts the lines at Reducer Machines.
  - The reduce() function is an *identity* function (does nothing)
- Count of URL access frequency
  - Logs of web page requests
  - Map() output is <URL, 1>.
  - Framework sorts the <URL, 1> pairs at Reducer Machines.
  - Reduce() adds together all values for the same URL and emits <URL, total-count> pair.
- Distributed Sort
  - Files containing records to be sorted
  - Map() extracts key from each record and emits <key, record>
  - Framework sorts the <key, record> pairs at Reducer Machines.
  - Reduce() emits all pairs unchanged.

Application Examples

- Reverse Web-Link Graph
  - Map() outputs <*target*, *source*> pairs for each link to a target URL found in a webpage that has name (also URL) as *source*
  - Framework sorts all <target,source> pairs at Reducer Machines.
  - Reduce() concatenates the list of all source URLs associated with a given target URL and emits the pair: <target, list(sources)>
- Inverted Index
  - Map() parses each document and emits a sequence of <word, document-ID> pairs.
  - Framework sorts the <word,document-ID> pairs (at Reducer Machines).
  - Reduce() accepts all pairs for a given word, sorts the corresponding document IDs and emits a <word, list(document ID)> pair.

Execution Overview

- 1) SPLIT: MapReduce library in **user program** splits the input files into M pieces (splits) of typically 16-64 MB each. Then it starts many copies of the user program on the machines of the cluster. Hence each machine runs a copy of the program.

- 2) SCHEDULE: One of the copies of the program is special – master. The rest are workers (N map and R reduce workers) that are assigned work by the master. There will be M map-tasks and R reduce-tasks to be assigned. Master picks up idle workers and assign each either map or reduce task.

- 3) MAP: A worker that is assigned map-task reads the content of the corresponding input-split, parses key-value pairs and passes each pair to the user-defined map() function. map() function produces intermediate key-value pairs and buffers them.

Execution Overview

- 4) INTERMEDIATE FILES: Periodically, buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The location of these files are passed to master, which forwards them later to the reduce workers.

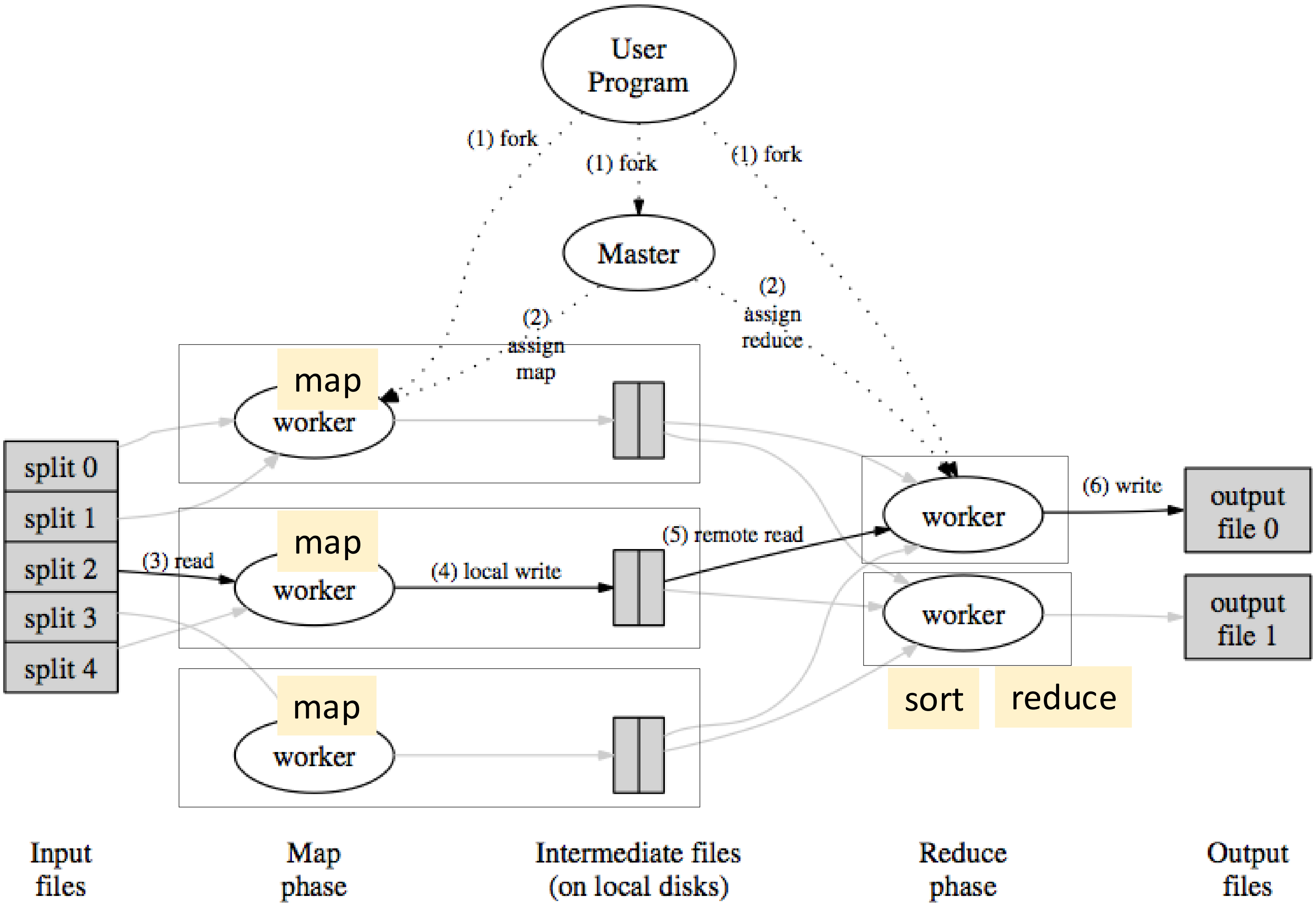
- 5) SORT AND GROUP: When a reduce worker is notified by the master about these locations (assigned a reduce task), it uses RPC to read the buffered regions (files) from map-worker local disks. When a reduce worker has read all data, it *sorts* by intermediate *key* so that all occurrences of the same key are *grouped* together. If memory is not enough, external sort can be used.

Execution Overview

- 6) REDUCE: The reduce worker iterates over the sorted intermediate key-value pairs and for each unique intermediate key encountered, it passes the key and the corresponding set of values to the user-defined reduce() function. The output of reduce() is appended to a final output file for this reduce partition.

- 7) FINISH: When all map and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce() call in the user program returns back to the user code.

At the end, R final output files are produced (one per reduce task).



User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
map
worker
map
worker
map
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
worker
worker
(6) write
output file 0
output file 1
sort
reduce
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Small Example: word count

Assume we have the following input data which is a sequence of lines of arbitrary words.

Assume M = 3, R = 2

this is a good school
cloud is nice today
sky and cloud nice school
the cloud computing blue
blue come true
sky is the limit
disk space the limit
nice output come today
hello cloud what nice is

Input Data

this is a good school
cloud is nice today
sky and cloud nice school

Split 0

the cloud computing blue
blue come true
sky is the limit

Split 1

disk space the limit
nice output come today
hello cloud what nice is

Split 2

Splitting the Input Data

Small Example

R1 R2

M = 3, **R = 2**

Machine M1

this is a good school
cloud is nice today
sky and cloud nice school

hash(key) mod R

map task 0

R1	R2
this 1 a 1 school 1 today 1 sky 1 and 1 school 1	is 1 good 1 cloud 1 is 1 nice cloud 1 nice 1

Machine M2

the cloud computing blue
blue come true
sky is the limit

map task 1

R1	R2
the 1 blue 1 blue 1 true 1 sky 1 the 1	cloud 1 computing 1 come 1 is 1 limit 1

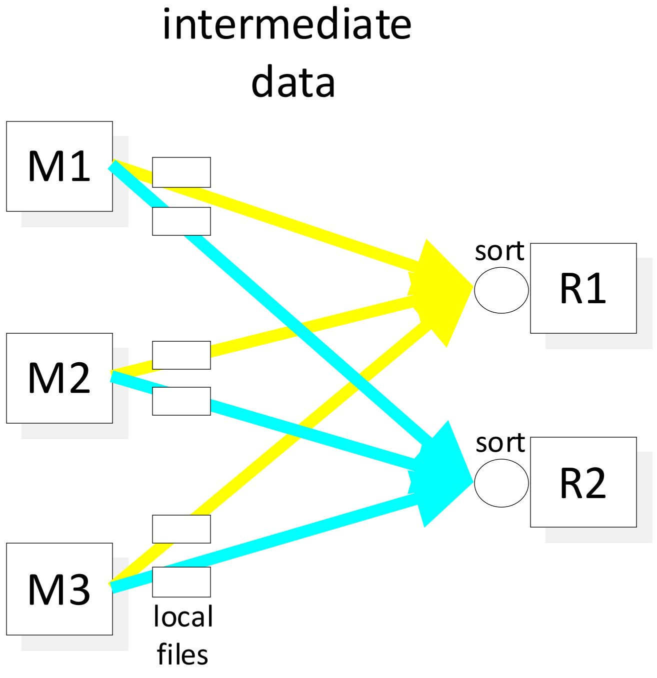
disk space the limit
nice output come today
hello cloud what nice is

map task 2

R1	R2
disk 1 space 1 the 1 today 1 what 1	limit 1 nice 1 output 1 come 1 hello 1 cloud 1 nice 1 is 1

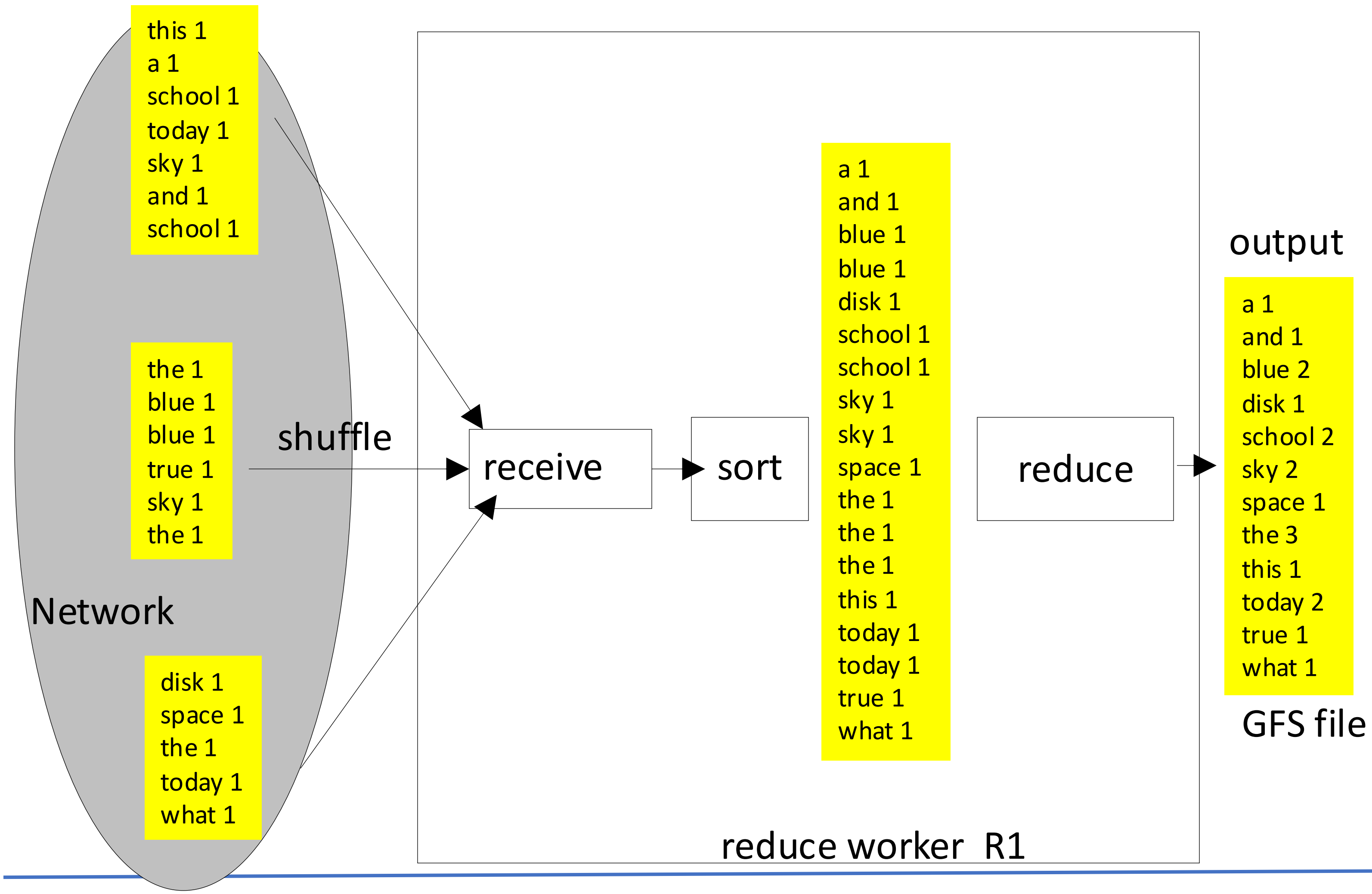
Machine M3

Small Example

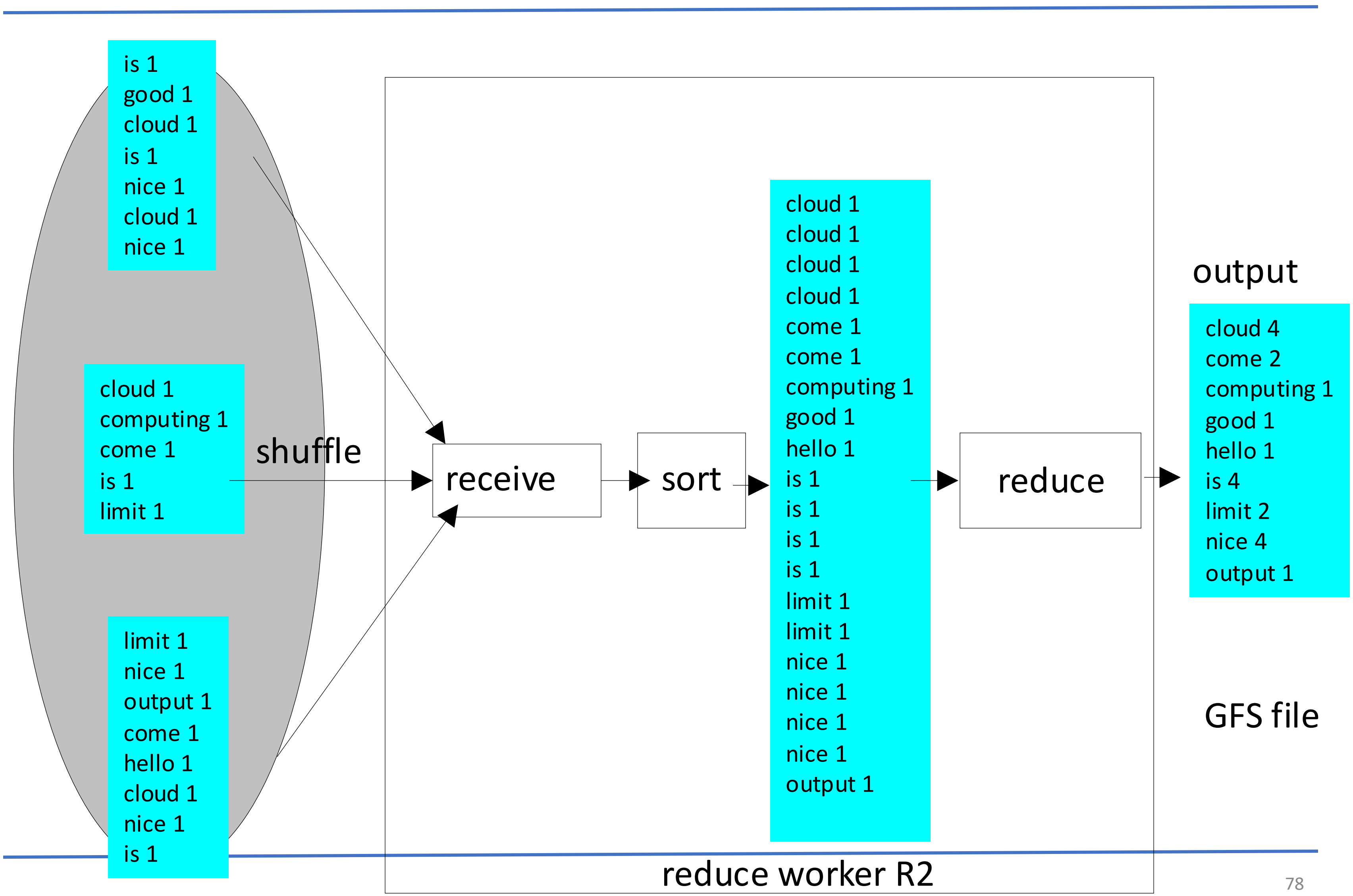


Shuffle over Network

Small Example



Small Example



Small Example

Result (Output) Files

a 1
and 1
blue 2
disk 1
school 2
sky 2
space 1
the 3
this 1
today 2
true 1
what 1

cloud 4
come 2
computing 1
good 1
hello 1
is 4
limit 2
nice 4
output 1

Sorted. Stored in GFS (a distributed file system).

Partitioning Function

- User specifies the number of reduce tasks (i.e., output files) that is desired: R.
- Data gets partitioned across these tasks using a partitioning function on the intermediate key
- Default function: hash(key) mod R
- User can specify a different function.
- Example:
  - hash(hostname(URL)) mod R
  - to have all entries belonging to a host in the same output file.

Additional Study Material (optional)

Spark

- MapReduce limitations (processing for big data)
  - Not good for <u>iterative</u> operations (Machine Learning algorithms): slow
  - Not good for <u>interactive</u> big data applications: slow
  - Difficulty in programming directly
  - Not good for every application
  - Good for batch applications working on big data
- Specialized systems built
  - Pregel, GraphLab, Storm.
- Spark’s goal was: to generalize MapReduce to support new apps with same engine
  - Still can work like map-reduce
  - But can do much more very efficiently (x10 or more)

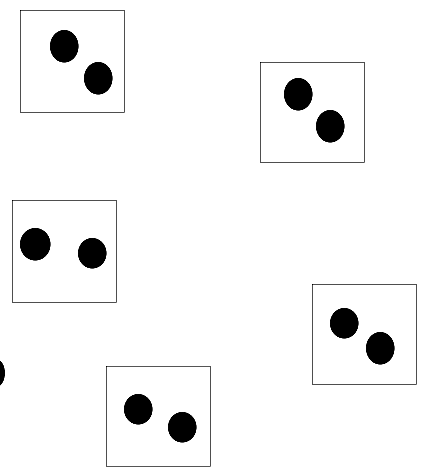
Spark features

- Handles batch, interactive and real-time jobs with a single framework
- Native integration with Java, Scala, Python
- Programming at a higher level of abstraction
- More general
  - Map/reduce is just one set of constructs

- It is a cluster computing framework. But can run on a single node (machine) as well.
  - Scalable (more nodes can be added to the cluster and Spark can utilize them)
  - Fault tolerant (node failures handled transparently)

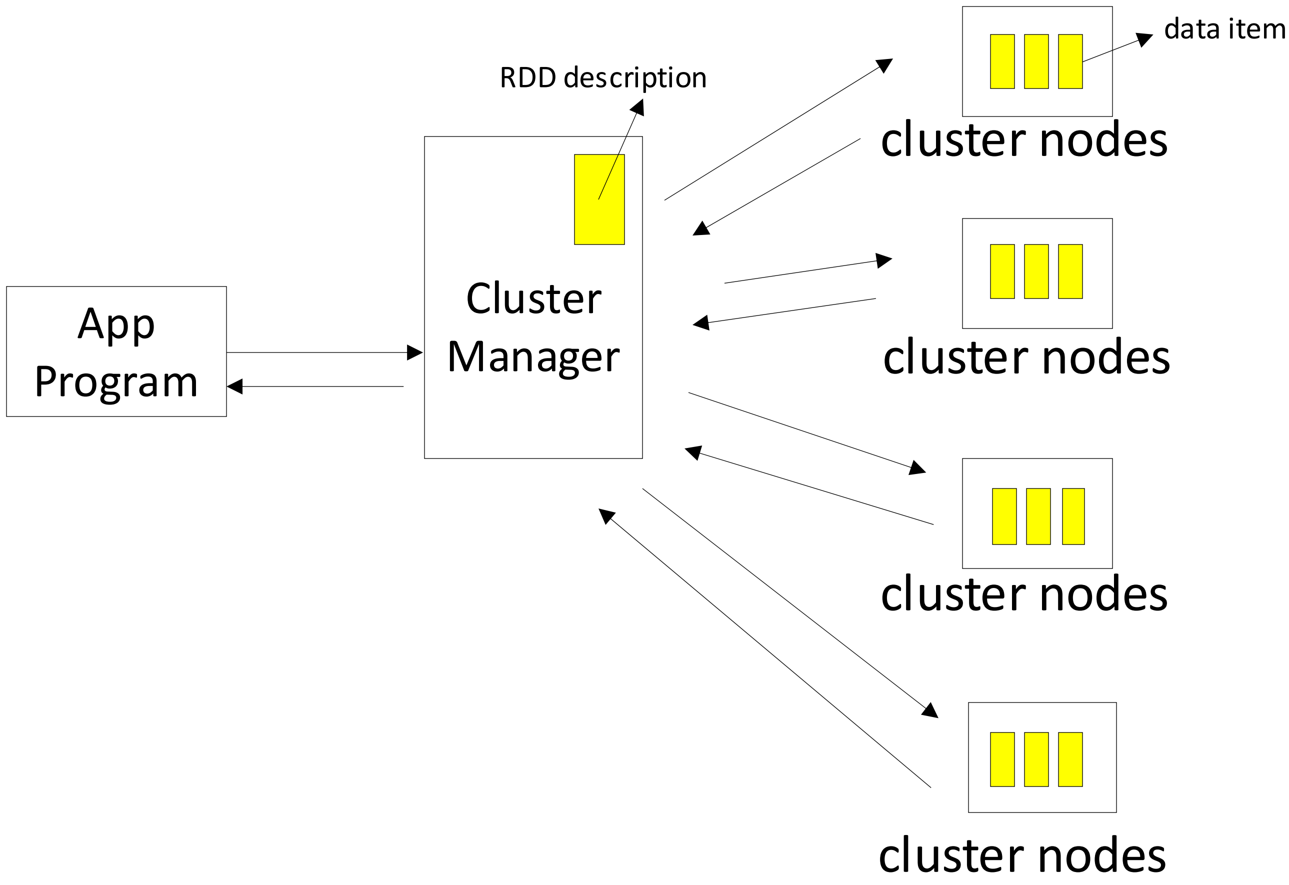
Spark

- Main abstraction in Spark is RDD (**resilient distributed dataset**)
- RDD represents a read-only collection of objects (data items) *partitioned* across a set of machines. Partition can be rebuilt if it is lost.
  - Data item (element) can be of various types.
- Users can explicitly cache an RDD across machines and reuse it in multiple MapReduce-like parallel operations.
- RDD has enough information about how it was derived from other RDDs (lineage) to be able to rebuild just that partition. Fault tolerance.
- There is a base RDD (on disk)



a machine (node)

Spark



data item
RDD description
cluster nodes
App
Program
Cluster
Manager
cluster nodes
cluster nodes
cluster nodes

RDD

- RDDs can only be created through deterministic *operations* (*transformations*) on either (1) data in stable storage or (2) other RDDs.
  - *map, flatmap, filter, join*
- RDDs do not need to be materialized at all times. RDD has enough information about how it was derived from other datasets (its lineage) to compute its partitions from data in stable storage.
- Users can control two other aspects of RDDs: *persistence* and *partitioning*.
  - Caching
  - Partitioning across machines on a key, etc.

Programming Interface

- For the programmer, each dataset (RDD) is represented as an object (language object) and transformations are invoked using methods on these objects.
  - Scala can be used.
  - Python can be used.
  - Java can be used
- Programmers start by defining one or more RDDs through *transformations* on data in stable storage
  - *map, fiter, …*
    - >>> linesRDD = sc.textFile ("world.txt")
- They can then use these RDDs in *actions*, which are operations that return a value to the application or export data to a storage system.
  - *count, collect, save, …*

RDDs can be stored or cached

- Programmers can call a *persist*() method to indicate which RDDs they want to reuse in future operations.
  - Spark keeps persistent RDDs in memory by default, but it can spill them to disk if there is not enough RAM.
  - Or can just put into the disk.
- The *cache*() method is similar, but default is Memory_Only.

Example: mining console logs

- Suppose that a web service is experiencing errors and an operator wants to search terabytes of logs in the Hadoop filesystem (HDFS), a distributed file system, to find the cause. Using Spark, the operator can load just the error messages from the logs into RAM across a set of nodes and query them interactively. The operator would first type the following Scala code:

Example: mining console logs

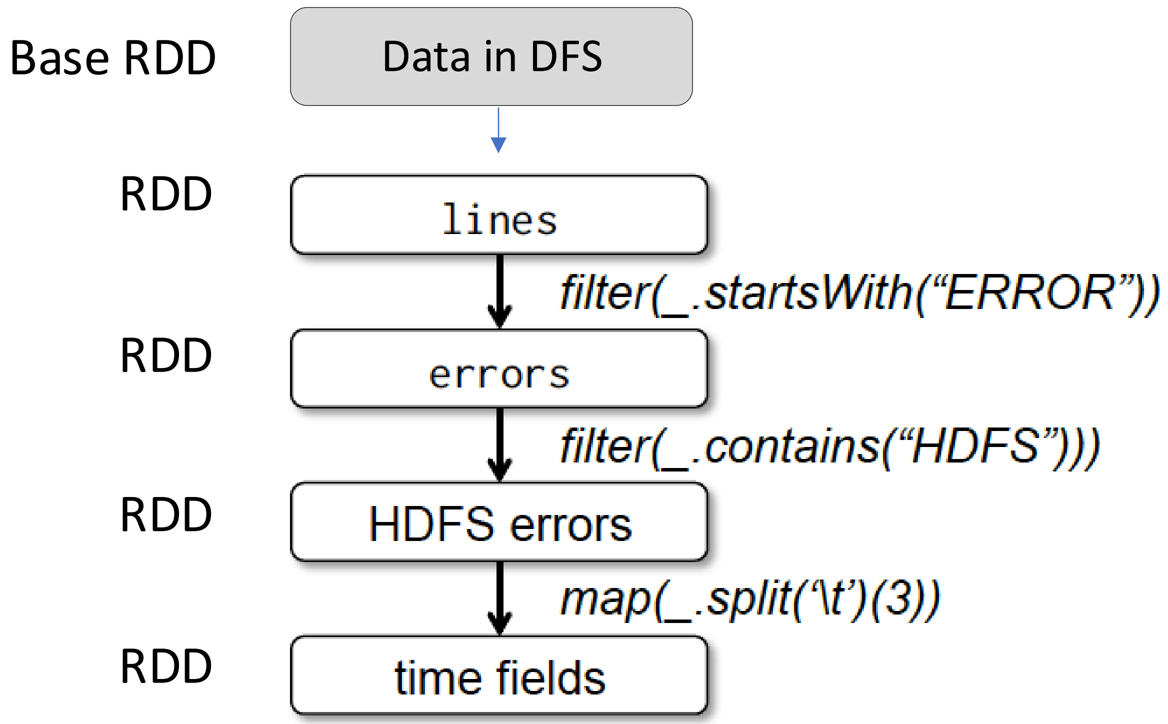
Extract and load error messages

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

querying

```
errors.count()

// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```

Lineage Graph



Extracting and querying error messages (illustrated)

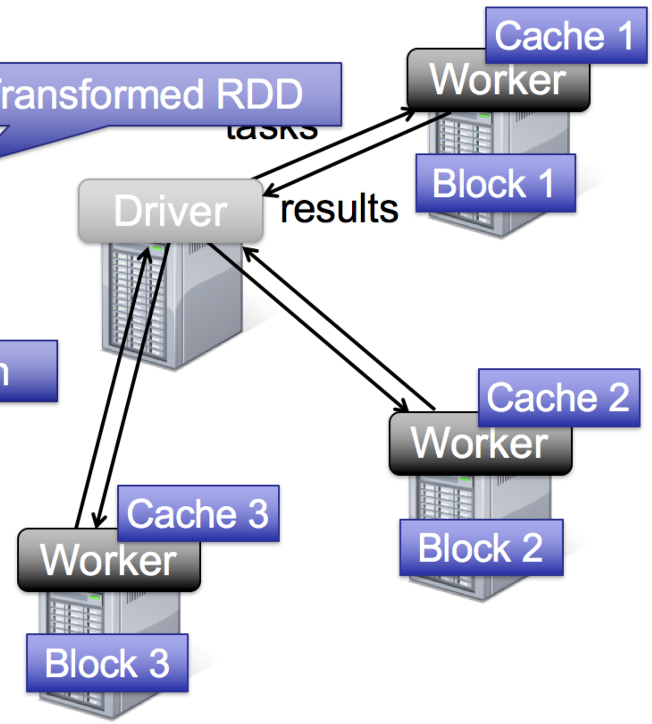
- Load error messages from a log into memory, then interactively search for patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split('\t')[2])
messages.cache()

messages.filter(lambda s: "foo" in s).count()
messages.filter(lambda s: "bar" in s).count()
. . .
```

Base RDD

Transformed RDD

Action

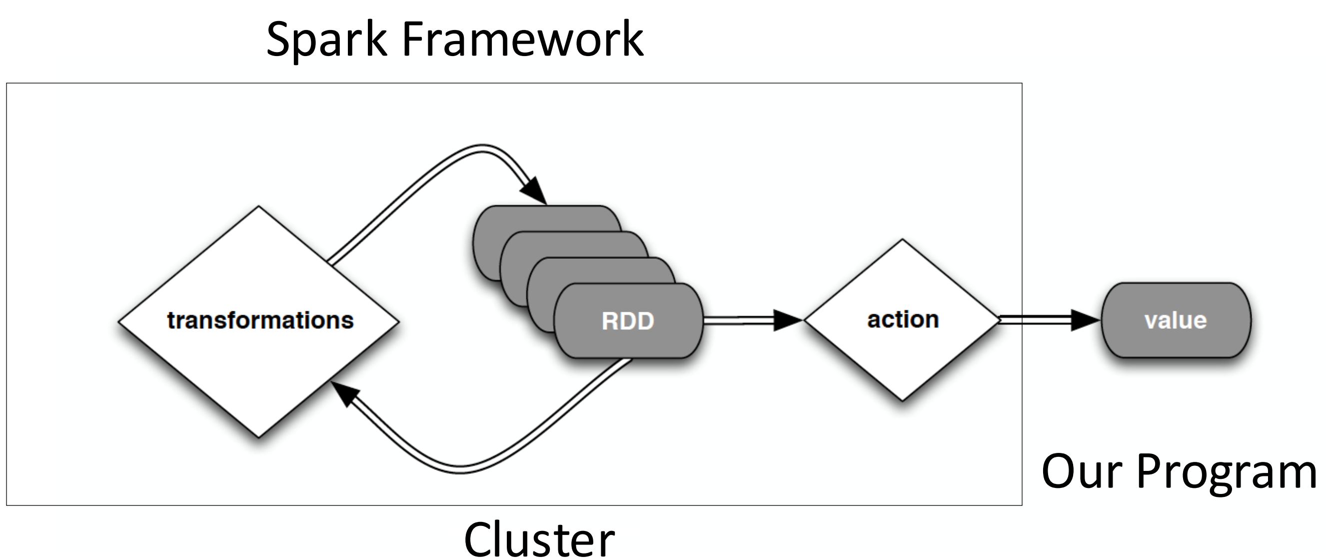


Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

Strata conference slides, 2013

RDD generation

- Spark can create RDDs from any file stored in HDFS or other storage systems supported by Hadoop, e.g., local file system, Amazon S3, Hypertable, HBase, etc.
- Spark supports text files, SequenceFiles, and any other Hadoop InputFormat, and can also take a directory or a glob (e.g. /data/201404*)



Generating RDDs in Python

```
# Turn a local collection into an RDD
sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8])

# Load text file from local FS, HDFS, or S3
sc.textFile(“file.txt”)
sc.textFile(“directory/*.txt”)
sc.textFile(“hdfs://namenode:9000/path/file”)

# Use any existing Hadoop InputFormat
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

Strata conference slides, 2013

RDD from another other RDD

- Transformations create a new dataset from an existing one
- All transformations in Spark are lazy: they do not compute their results right away – instead they remember the transformations
- applied to some base dataset
- optimize the required calculations
- recover from lost data partitions

Strata conference slides, 2013

```
nums = sc.parallelize([1, 2, 3])
# Pass each element through a function
squares = nums.map(lambda x: x*x)   # => {1, 4, 9}
# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0) # => {4}
# Map each element to zero or more others
nums.flatMap(lambda x: range(0, x))  # => {0, 0, 1, 0, 1, 2}
```

Operations: Transformations

transformation	*description*
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)
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
distinct([*numTasks***]))**	return a new dataset that contains the distinct elements of the source dataset

Operations: Transformations

transformation	*description*
groupByKey([*numTasks*]**)**	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Seq[V])` pairs
reduceByKey(*func*, [*numTasks*]**)**	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
sortByKey([*ascending*], [*numTasks*]**)**	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 ascending argument
join(*otherDataset*, [*numTasks*]**)**	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, (V, W))` pairs with all pairs of elements for each key
cogroup(*otherDataset*, [*numTasks*]**)**	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, Seq[V], Seq[W])` tuples – also called `groupWith`
cartesian(*otherDataset***)**	when called on datasets of types `T` and `U`, returns a dataset of `(T, U)` pairs (all pairs of elements)

Operations: Actions

action	*description*
reduce(*func*)	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also 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 – 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 – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement*, *fraction*, *seed*)	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

Operations: Actions

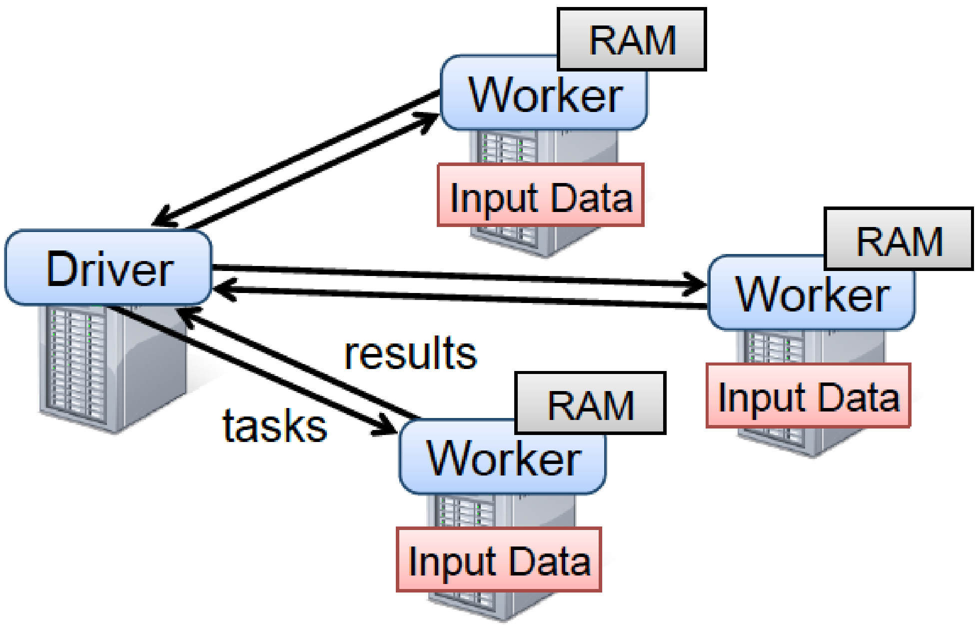
action	*description*
`saveAsTextFile(`*`path`*`)`	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
`saveAsSequenceFile(`*`path`*`)`	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. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
`countByKey()`	only available on RDDs of type `(K, V)`. Returns a \`Map\` of `(K, Int)` pairs with the count of each key
`foreach(`*`func`*`)`	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

RDD operations (Summary)

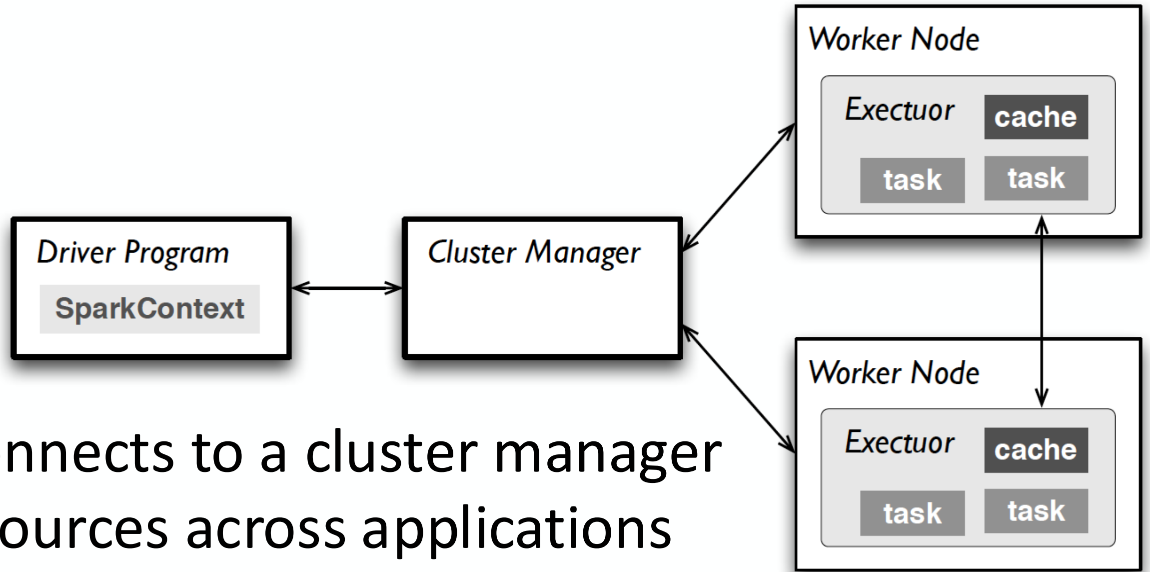
Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction : \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f : (\mathrm{V},\mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	$join()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
Actions	$count()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Long}$
	$collect()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Seq[T]}$
	$reduce(f : (\mathrm{T},\mathrm{T}) \Rightarrow \mathrm{T})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{T}$
	$lookup(k : \mathrm{K})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{Seq[V]}$ (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Spark Runtime

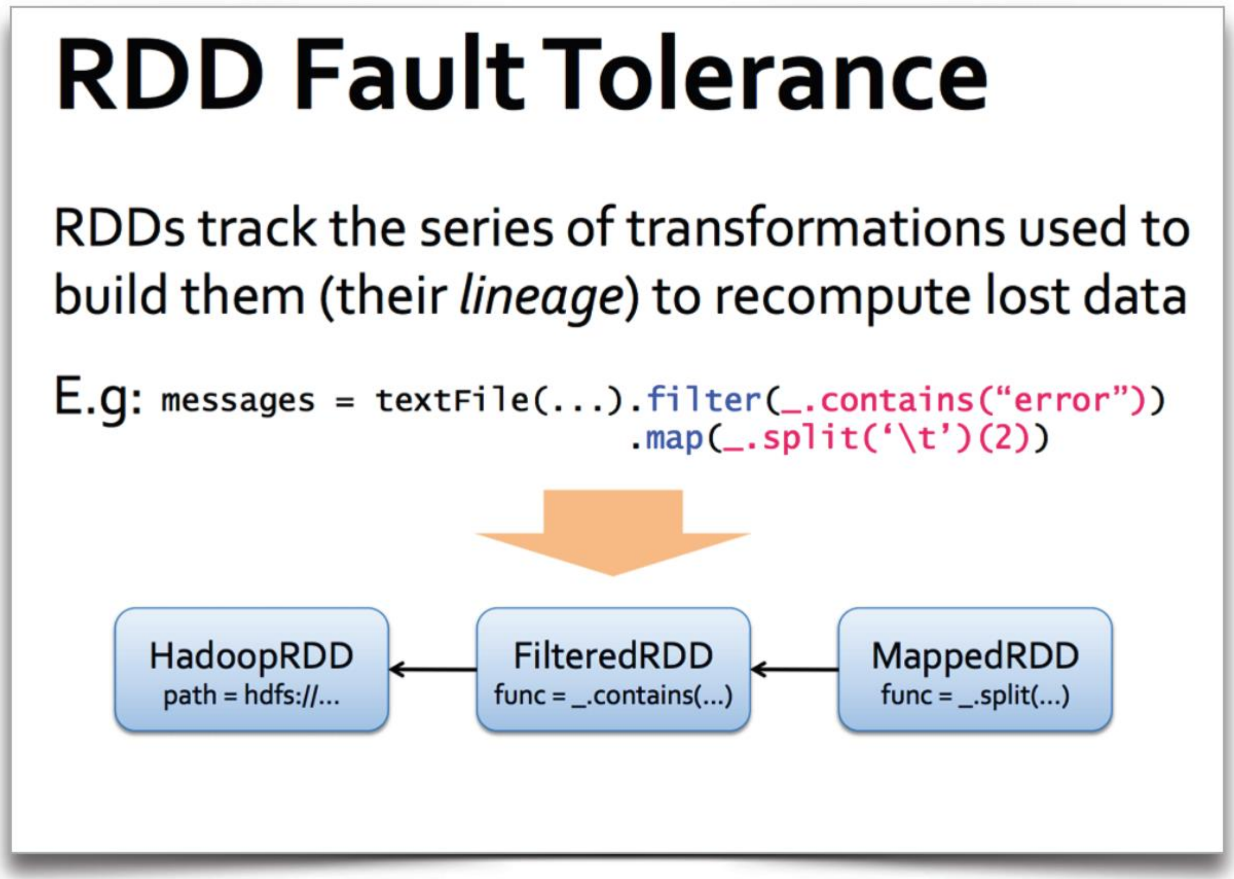


Spark Runtime



1. Our Program connects to a cluster manager which allocate resources across applications
2. acquires executors on cluster nodes – worker processes to run computations and store data
3. sends app code to the executors
4. sends tasks for the executors to run

How fault tolerance achieved



A text-file example to form RDD

- We can dowload a textfile from Internet
  - Ebook from Gutenberg project.
- Assume the downloaded ebook (Short History of the World) is put into a txt file world.txt

word.txt

world.txt

```
The Project Gutenberg EBook of A Short History of the World, by H. G. Wells

This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever.  You may copy it, give it away or
re-use it under the terms of the Project Gutenberg License included
with this eBook or online at www.gutenberg.net


Title: A Short History of the World

Author: H. G. Wells

Release Date: March 2, 2011 [EBook #35461]
[Last updated: November 3, 2011]

Language: English


*** START OF THIS PROJECT GUTENBERG EBOOK A SHORT HISTORY OF THE WORLD ***




Produced by Donald F. Behan
```

word.txt

world.txt

```
{v}

PREFACE

This SHORT HISTORY OF THE WORLD is meant to be read
straightforwardly almost as a novel is read.  It gives in the most
general way an account of our present knowledge of history, shorn
of elaborations and complications.  It has been amply illustrated
and everything has been done to make it vivid and clear.  From it
the reader should be able to get that general view of history
which is so necessary a framework for the study of it particular
period or the history of a particular country.  It may be found
useful as a preparatory excursion before the reading of the
author's much fuller and more explicit _Outline of History_ is
undertaken.  But its especial end is to meet the needs of the busy
general reader, too driven to study the maps and time charts of
that _Outline_ in detail, who wishes to refresh and repair his
faded or fragmentary conceptions of the great adventure of
mankind.  It is not an abstract or condensation of that former
work.  Within its aim the _Outline_ admits of no further
condensation.  This is a much more generalized History, planned
and written afresh.

{vii}



 CONTENTS
```

…

Process text file

- We can now process this file. For example, to obtain all words in the book into a list, or to count the words.

- To obtain words, in our Python program we write:
  - distFile = sc.textFile(“world.txt")!
  - distFile.map(lambda x: x.split(' ')).collect()

Word count

Python code:

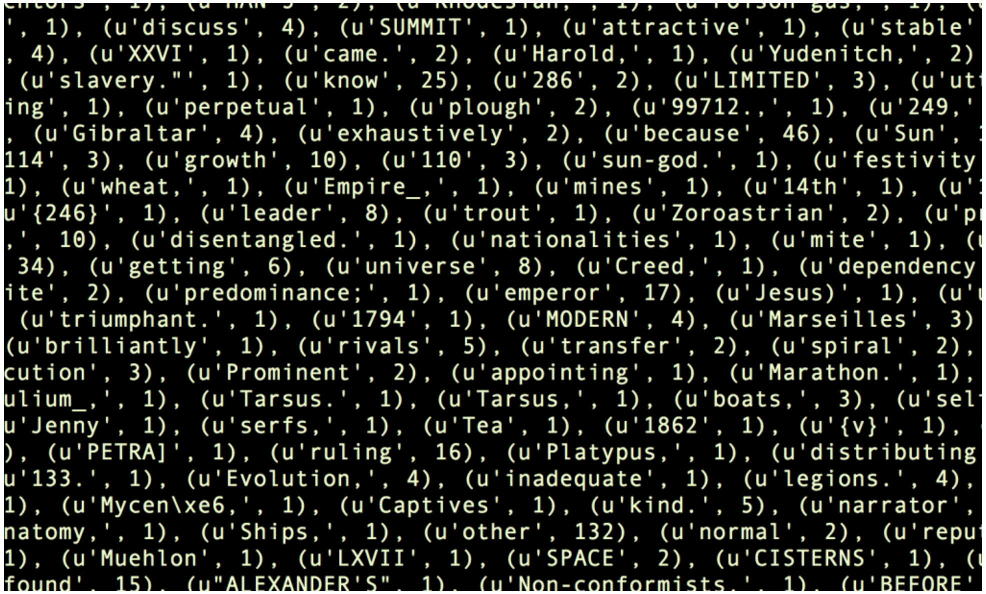
from operator import add

→ flatMap → Map

f = sc.textFile("world.txt")

words = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1))

words.reduceByKey(add).collect()



```
', 1), (u'discuss', 4), (u'SUMMIT', 1), (u'attractive', 1), (u'stable'
, 4), (u'XXVI', 1), (u'came.', 2), (u'Harold,', 1), (u'Yudenitch,', 2)
 (u'slavery."', 1), (u'know', 25), (u'286', 2), (u'LIMITED', 3), (u'ut
ing', 1), (u'perpetual', 1), (u'plough', 2), (u'99712.,', 1), (u'249,'
, (u'Gibraltar', 4), (u'exhaustively', 2), (u'because', 46), (u'Sun',
114', 3), (u'growth', 10), (u'110', 3), (u'sun-god.', 1), (u'festivity
1), (u'wheat,', 1), (u'Empire_,', 1), (u'mines', 1), (u'14th', 1), (u'
u'{246}', 1), (u'leader', 8), (u'trout', 1), (u'Zoroastrian', 2), (u'p
,', 10), (u'disentangled.', 1), (u'nationalities', 1), (u'mite', 1), (
 34), (u'getting', 6), (u'universe', 8), (u'Creed,', 1), (u'dependency
ite', 2), (u'predominance;', 1), (u'emperor', 17), (u'Jesus)', 1), (u'
 (u'triumphant.', 1), (u'1794', 1), (u'MODERN', 4), (u'Marseilles', 3)
(u'brilliantly', 1), (u'rivals', 5), (u'transfer', 2), (u'spiral', 2),
cution', 3), (u'Prominent', 2), (u'appointing', 1), (u'Marathon.', 1),
ulium_,', 1), (u'Tarsus.', 1), (u'Tarsus,', 1), (u'boats,', 3), (u'sel
u'Jenny', 1), (u'serfs,', 1), (u'Tea', 1), (u'1862', 1), (u'{v}', 1),
), (u'PETRA]', 1), (u'ruling', 16), (u'Platypus,', 1), (u'distributing
u'133.', 1), (u'Evolution,', 4), (u'inadequate', 1), (u'legions.', 4),
1), (u'Mycen\xe6,', 1), (u'Captives', 1), (u'kind.', 5), (u'narrator',
natomy,', 1), (u'Ships,', 1), (u'other', 132), (u'normal', 2), (u'repu
1), (u'Muehlon', 1), (u'LXVII', 1), (u'SPACE', 2), (u'CISTERNS', 1), (
```

Word count

- Spark can persist (or cache) a dataset in memory across operations
- Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than 10x faster
- The cache is fault-tolerant: if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it

```
from operator import add
f = sc.textFile("README.md")
w = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).cache()
w.reduceByKey(add).collect()
```

Accumulators

- Accumulators are variables that can only be "added" to through an associative operation
- Used to implement counters and sums, efficiently in parallel
- Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types
- Only the driver program can read an accumulator's value, not the tasks

Accumulators

- We can define and use an accumulator variable. All functions, no matter in which node they are executed, can add into the accumulator variable.

Python:

```
accum = sc.accumulator(0)
rdd = sc.parallelize([1, 2, 3, 4])
def f(x):
   global accum
   accum += x

rdd.foreach(f)

accum.value
```

Create the variable

There are 4 elements in the dataset

We define a function

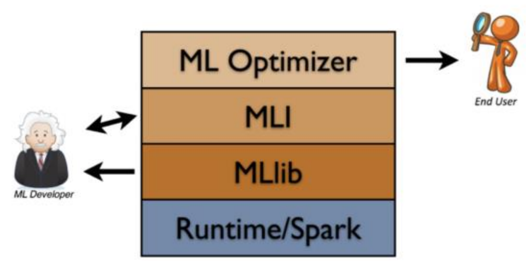
We are executing the function on each dataset element x

We are accessing to the accumulated value

Spark libraries/frameworks

- Spark Streaming
  - Stream analytics

- MLlib
  - Distributed machine learning framework



- GraphX
  - Distributed graph processing framework

References

1. Database System Concepts. Silberschatz et al. 6th edition. 2011.
2. CS109 Data Science, Harvard.
3. CMSC320 Introduction to Data Science, UMD.
4. 15-388/688 Practical Data Science, CMU.
5. CS194 Introduction to Data Science, UC Berkeley.
6. CSCI 1951A. Data Science, Brown.
7. Cloud Computing: Theory and Practice, D. Marinescu, Morgan Kaufmann, 2013.
8. MapReduce: Simplified Data Processing on Large Clusters, J. Dean and S. Ghamawat, OSDI, 2004.
9. Mining of Massive Datasets, J. Leskovec, A. Rajaraman, J. Ullman.
10. CS240A, UCSB.
11. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for
12. In-Memory Cluster Computing, Zaharia et al., NSDI 2012.

References

- https://phppot.com/php/php-restful-web-service/
- Sqlite3: https://www.sqlite.org/index.html
- RDMBs and Pandas: https://www.textbook.ds100.org/ch/09/sql_intro.html
- https://www.textbook.ds100.org/ch/03/pandas_intro.html
- https://medium.com/swlh/pyspark-on-macos-installation-and-use-31f84ca61400