Big Data Analysis with R

Anurag Nagar

CS 6301

Spark

Project Goals

Extend the MapReduce model to better support two common classes of analytics apps:

>> Iterative algorithms (machine learning, graph)

>> Interactive data mining

Enhance programmability:

>> Integrate into Scala programming language

>> Allow interactive use from Scala interpreter

Motivation

- MapReduce greatly simplified “big data” analysis on large, unreliable clusters
- But as soon as it got popular, users wanted more:
  - More **complex**, multi-stage applications (e.g. iterative machine learning & graph processing)
  - More **interactive** ad-hoc queries

Response: *specialized* frameworks for some of these apps (e.g. Pregel for graph processing)

Motivation

- Complex apps and interactive queries both need one thing that MapReduce lacks:
    - Efficient primitives for **data sharing**

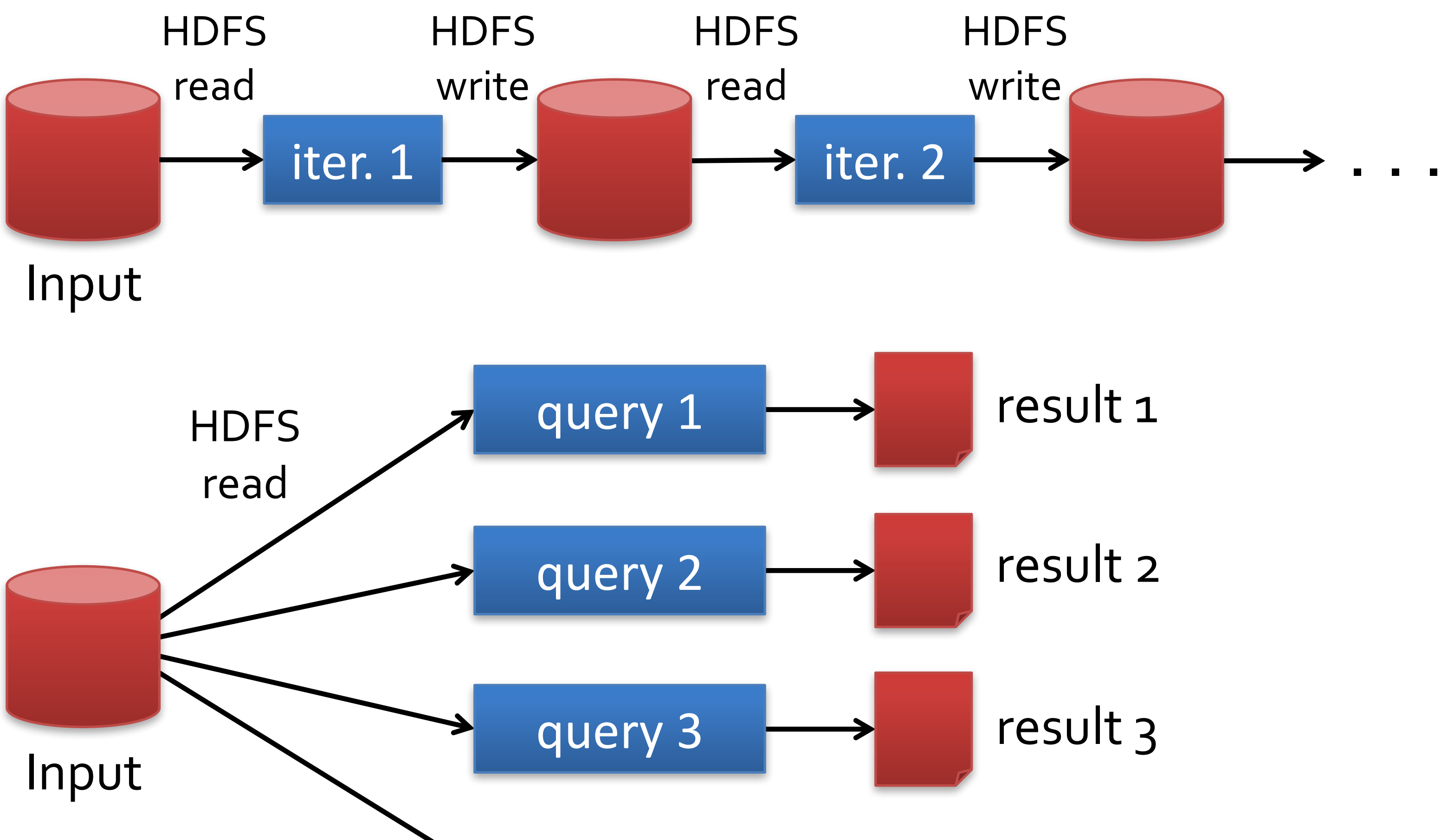
In MapReduce, the only way to share data across jobs is stable storage → slow!

Memory vs Disk

If Memory = **Minute**
Network = **Weeks**
Flash = **Months**
Disk = **Decades**

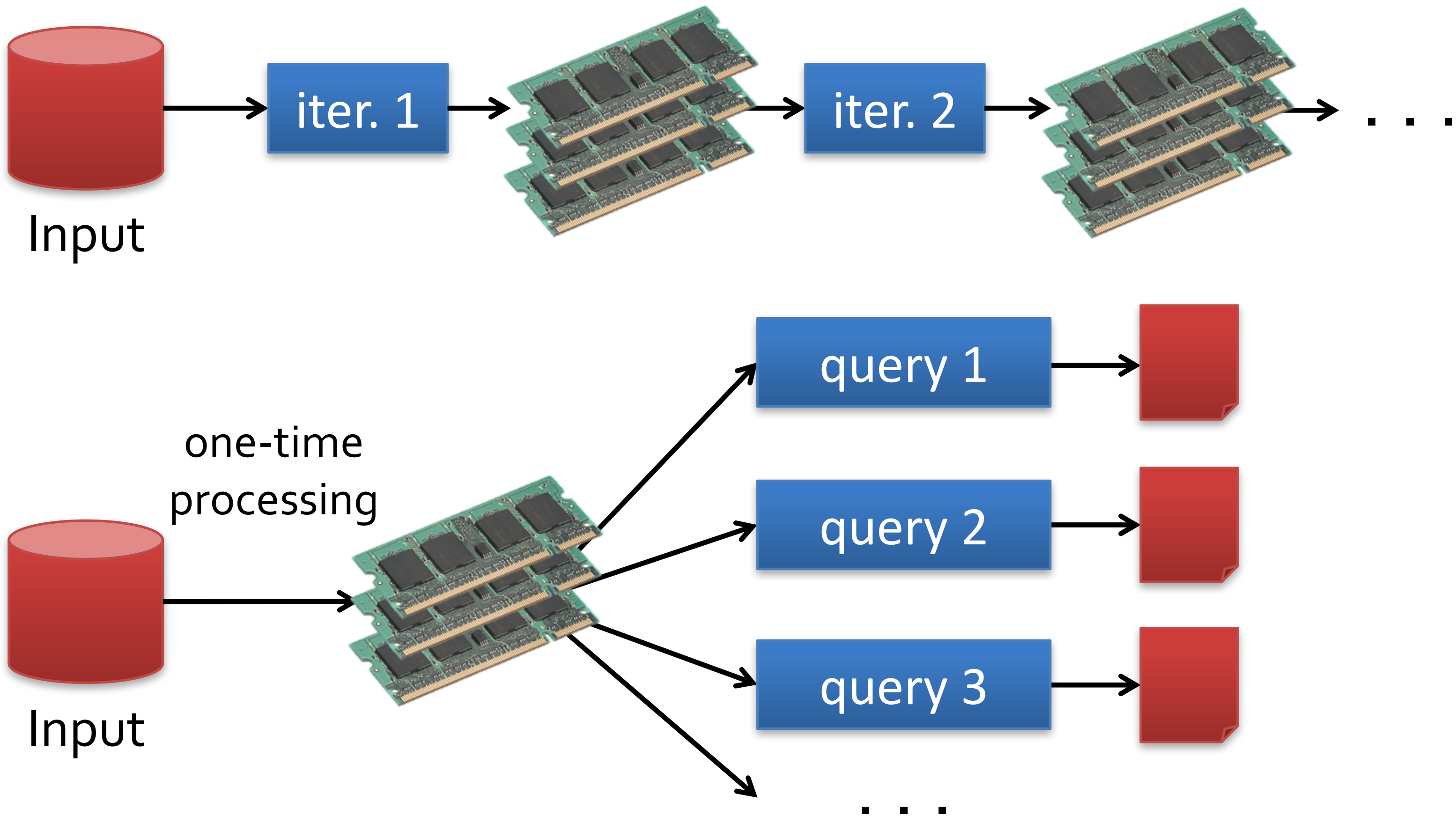
Source: http://blog.infinio.com/relative-speeds-from-ram-to-flash-to-d
http://blog.scoutapp.com/articles/2011/02/08/how-much-slower-is-di

Examples



Slow due to replication and disk I/O,
but necessary for fault tolerance

Goal: In-Memory Data Sharing



10-100× faster than network/disk, but how to get FT?

Challenge

How to design a distributed memory abstraction that is both **fault-tolerant** and **efficient**?

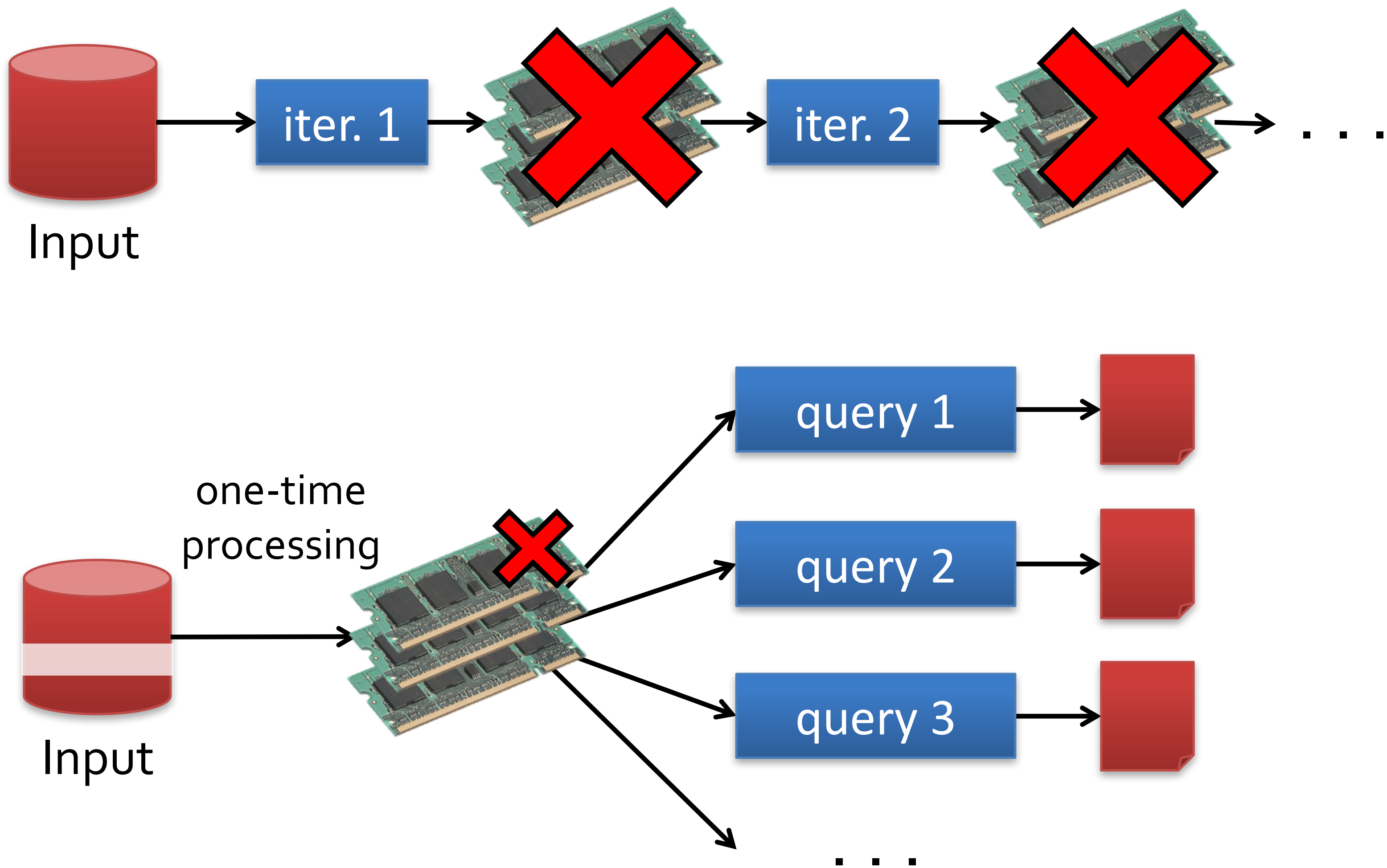
Challenge

- Existing storage abstractions have interfaces based on *fine-grained* updates to mutable state
  - RAMCloud, databases, distributed mem, Piccolo
- Requires replicating data or logs across nodes for fault tolerance
  - Costly for data-intensive apps
  - 10-100x slower than memory write

Solution: Resilient Distributed Datasets (RDDs)

- Restricted form of distributed shared memory
  - Immutable, partitioned collections of records
  - Can only be built through *coarse-grained* deterministic transformations (map, filter, join, …)
- Efficient fault recovery using *lineage*
  - Log one operation to apply to many elements
  - Recompute lost partitions on failure
  - No cost if nothing fails

RDD Recovery



Generality of RDDs

- Despite their restrictions, RDDs can express surprisingly many parallel algorithms
  - These naturally *apply the same operation to multiple items*
- Unify many current programming models
  - *Data flow models:* MapReduce, Dryad, SQL, …
  - *Specialized models* for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), bulk incremental, …
- Support *new apps* that these models don't

Basics of Spark

Go through earlier presentation on basics and architecture of Apache Spark

SparkR

Getting Started

- Sign up for an account at Databricks Community Edition: https://community.cloud.databricks.com/
- Go through basic tutorials:
- https://spark.apache.org/docs/latest/sparkr.html
- https://docs.databricks.com/spark/latest/sparkr/overview.html
- https://rpubs.com/wendyu/sparkr

Getting Started

SparkR library has to be loaded before proceeding further

```r
library(SparkR)
df <- createDataFrame(iris)
```

Note: Spark is already loaded in Databricks. If you are using any other resource, you would need to load it.

Dataframe Operations

SparkR lets you focus on the code, while taking care of query parallelization and distribution.

Let's load the old faithful dataset and run some operations:

```r
library(SparkR)
df <- createDataFrame(faithful)
head(df)
head(summary(df))
```

Dataframe Operations

Some more queries:

```r
library(SparkR)
head(select(df, "eruptions"))
head(filter(df, df$waiting < 50))
head(count(groupBy(df, df$waiting)))
```

Other examples

We will use R examples from the Spark github repository:
https://github.com/apache/spark

Make sure to upload the data folder to Databricks so you can run the queries.

Search Engine Creation

Search engine using tm package

The steps for creating a search engine are explained here:

https://rpubs.com/ftoresh/search-engine-Corpus

Project

Project Description

- For this project, you will work with a dataset of movie plot summaries that is available from the Carnegie Movie Summary Corpus site: http://www.cs.cmu.edu/~ark/personas/. Please download the version **"Dataset [46 M]"** and **not** the **"Stanford CoreNLP-processed summaries [628 M]"** version. Please upload this dataset to your UTD web account. Do not hardcode any local paths.
- We are interested in building a search engine for the plot summaries that are available in the file "plot summaries.txt" that is available under the Dataset link of the above page.

Project Description

- You will use the tf-idf technique studied in class to accomplish the above task.
- For more details on how to compute tf-idf using MapReduce, see the links below:

1. Good introduction from Coursera Distributed Programming course:

https://www.coursera.org/lecture/distributed-programming-in-java/1-4-tf-idf-example-4Sitg

2. Chapter 4 of the reference book Data-Intensive Text Processing using MapReduce:

https://lintool.github.io/MapReduceAlgorithms/

Project Steps

The project can be done using the following steps. You are free to make any reasonable changes:

1. First of all you would need to remove stop words, which can be done by searching for an appropriate package. Some suggestions are:

- *qdap* package:

  https://cran.r-project.org/web/packages/qdap/vignettes/qdap_vignette.html

- *tm* package:

  https://cran.r-project.org/web/packages/tm/vignettes/tm.pdf

Project Steps

2. The second step would be word stemming. This can be done using the *corpus* package or any other suitable package.
3. After the above pre-processing steps, you are ready to compute tf-idf values for each document-term pair. You can save this for faster processing next time.

Project Steps

4. Your program should read query phrases (which could be multiple terms such as "action movies with comedy scenes") from the command line and should return the top 10 documents matching the user's queries. The program should terminate when the user presses the "q" key.

Project Steps

5. The query could be of two types:

- **Single terms**: such as "Dallas". In such cases, you can simply return the top 10 documents with the highest tf-idf values for this term.
- **Multiple terms**: such as "movies starring Brad Pitt". In such a case, you would need to compute *cosine similarity* between the query and each of the documents, and return the top 10 most similar documents.

Project Steps

Some helpful hints for evaluating cosine similarity are:

```
- http://text2vec.org/similarity.html
See the section *Cosine similarity with Tf-Idf*

- https://courses.cs.washington.edu/courses/cse573/12sp/lectures/17-ir.pdf
```