Data Intensive Computing Frameworks

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
1394/2/25
Big Data

small data

big data

www.jolyen.co.uk
Big Data refers to datasets and flows large enough that has outpaced our capability to store, process, analyze, and understand.
The Four Dimensions of Big Data

- **Volume**: data size
- **Velocity**: data generation rate
- **Variety**: data heterogeneity
- **This 4th V is for Vacillation**: Veracity/Variability/Value
Where Does Big Data Come From?
Big Data Market Driving Factors

The number of web pages indexed by Google, which were around one million in 1998, have exceeded one trillion in 2008, and its expansion is accelerated by appearance of the social networks.*

*“Mining big data: current status, and forecast to the future” [Wei Fan et al., 2013]
Big Data Market Driving Factors

The amount of mobile data traffic is expected to grow to 10.8 Exabyte per month by 2016.*

*“Worldwide Big Data Technology and Services 2012-2015 Forecast” [Dan Vesset et al., 2013]
More than **65 billion devices** were connected to the Internet by 2010, and this number will go up to **230 billion** by 2020.*

*"The Internet of Things Is Coming" [John Mahoney et al., 2013]
Many companies are moving towards using Cloud services to access Big Data analytical tools.
Open source communities
How To Store and Process Big Data?
But First, The History
4000 B.C

- Manual recording
- From tablets to papyrus, to parchment, and then to paper
▶ Gutenberg’s printing press
1800’s - 1940’s

- Punched cards (no fault-tolerance)
- Binary data
- 1890: US census
- 1911: IBM appeared
1940’s - 1950’s

- Magnetic tapes
1950’s - 1960’s

- Large-scale mainframe computers
- Batch transaction processing
- File-oriented record processing model (e.g., COBOL)
1960’s - 1970’s

- Hierarchical DBMS (one-to-many)
- Network DBMS (many-to-many)
- VM OS by IBM → multiple VMs on a single physical node.
1970’s - 1980’s

- Relational DBMS (tables) and SQL
- ACID
- Client-server computing
- Parallel processing
1990’s - 2000’s

- Virtualized Private Network connections (VPN)
- The Internet...
2000’s - Now

- Cloud computing
- NoSQL: BASE instead of ACID
- Big Data
How To Store and Process Big Data?
Scale up or scale vertically: adding resources to a single node in a system.

Scale out or scale horizontally: adding more nodes to a system.
Scale Up vs. Scale Out (2/2)

- Scale up: more expensive than scaling out.
- Scale out: more challenging for fault tolerance and software development.
Taxonomy of Parallel Architectures

Two Main Types of Tools

- Data store
- Data processing
Data Store
Data Store

- How to store and access files? File System
What is Filesystem?

- Controls how data is stored in and retrieved from disk.
What is Filesystem?

- Controls how data is stored in and retrieved from disk.
Distributed Filesystems

- When data outgrows the storage capacity of a single machine.
Distributed Filesystems

- When data **outgrows** the storage capacity of a **single** machine.

- **Partition** data across a **number of separate** machines.
Distributed Filesystems

- When data outgrows the storage capacity of a single machine.
- Partition data across a number of separate machines.
- Distributed filesystems: manage the storage across a network of machines.
Distributed Filesystems

- When data \textit{outgrows} the storage capacity of a \textit{single} machine.

- \textbf{Partition} data across a \textit{number of separate} machines.

- \textbf{Distributed filesystems}: manage the storage across a network of machines.
HDFS (1/2)

- Hadoop Distributed FileSystem
- Appears as a single disk
- Runs on top of a native filesystem, e.g., ext3
HDFS (2/2)

Management node
Secondary Namenode
Namenode

Datanode
Node 1
Datanode
Node 2
...
Datanode
Node n
Files and Blocks (1/2)

- Files are split into blocks.

- Blocks, the single unit of storage.
  - Transparent to user.
  - 64MB or 128MB.
• Same block is **replicated** on multiple machines: default is 3

Test.txt file = chunk #1 (c1) + chunk #2 (c2)

Master

<table>
<thead>
<tr>
<th>Rack 1</th>
<th>Rack n</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChunkServer</td>
<td>ChunkServer</td>
</tr>
<tr>
<td>C1, C2</td>
<td>C1, C2</td>
</tr>
<tr>
<td>ChunkServer</td>
<td>ChunkServer</td>
</tr>
<tr>
<td>C1</td>
<td>C1</td>
</tr>
<tr>
<td>ChunkServer</td>
<td>ChunkServer</td>
</tr>
<tr>
<td>C2</td>
<td>C2</td>
</tr>
</tbody>
</table>
HDFS Write

- 1. Create a new file in the Namenode’s Namespace; calculate block topology.
- 2, 3, 4. Stream data to the first, second and third node.
- 5, 6, 7. Success/failure acknowledgment.
HDFS Read

- 1. Retrieve block locations.
- 2, 3. Read blocks to re-assemble the file.
What About Databases?
Database and Database Management System

- **Database**: an organized collection of data.
Database and Database Management System

- **Database**: an organized collection of data.

- **Database Management System (DBMS)**: a software that interacts with users, other applications, and the database itself to capture and analyze data.
Relational Databases Management Systems (RDMBSs)

- **RDMBSs**: the dominant technology for storing structured data in web and business applications.

- **SQL** is good
  - Rich language and toolset
  - Easy to use and integrate
  - Many vendors
Relational Databases Management Systems (RDMBSs)

- **RDMBSs**: the **dominant** technology for storing **structured** data in web and business applications.

- **SQL** is **good**
  - Rich language and toolset
  - Easy to use and integrate
  - Many **vendors**

- They promise: **ACID**
ACID Properties

- Atomicity
- Consistency
- Isolation
- Durability
Web-based applications caused spikes.

- Internet-scale data size
- High read-write rates
- Frequent schema changes
Scaling RDBMSs is **Expensive and Inefficient**

Not Only SQL
NoSQL

- **Avoidance** of unneeded **complexity**

- **High** **throughput**

- **Horizontal scalability** and running on **commodity** hardware
NoSQL Cost and Performance

Database Scales Out
Just add more commodity database servers

NoSQL Data Models: Key-Value

- Collection of key/value pairs.

- **Ordered** Key-Value: processing over key ranges.

- Dynamo, Scalaris, Voldemort, Riak, ...
NoSQL Data Models: Column-Oriented

- Similar to a key/value store, but the value can have multiple attributes (Columns).

- Column: a set of data values of a particular type.

- BigTable, Hbase, Cassandra, ...
NoSQL Data Models: Document-Based

- Similar to a column-oriented store, but values can have complex documents, e.g., XML, YAML, JSON, and BSON.

- CouchDB, MongoDB, ...

```json
{
    FirstName: "Bob",
    Address: "5 Oak St.",
    Hobby: "sailing"
}

{
    FirstName: "Jonathan",
    Address: "15 Wanamassa Point Road",
    Children: [
        {Name: "Michael", Age: 10},
        {Name: "Jennifer", Age: 8},
    ]
}
```
NoSQL Data Models: Graph-Based

- Uses **graph** structures with **nodes**, **edges**, and **properties** to represent and store data.

- Neo4J, InfoGrid, ...
Data Processing
Challenges

- How to distribute computation?
- How can we make it easy to write distributed programs?
- Machines failure.
Idea

- **Issue:**
  - Copying data over a network takes **time**.
Idea

- **Issue:**
  - Copying data over a network takes time.

- **Idea:**
  - Bring computation close to the data.
  - Store files multiple times for reliability.
MapReduce

▶ A shared nothing architecture for processing large data sets with a parallel/distributed algorithm on clusters.
▶ Don’t worry about parallelization, fault tolerance, data distribution, and load balancing (MapReduce takes care of these).

▶ Hide system-level details from programmers.
Warm-up Task (1/2)

- We have a huge text document.

- Count the number of times each distinct word appears in the file.
File too large for memory, but all \langle word, count \rangle pairs fit in memory.
Warm-up Task (2/2)

- File too large for memory, but all \(\langle \text{word}, \text{count} \rangle\) pairs fit in memory.

- \texttt{words(doc.txt) | sort | uniq -c}
  - where \texttt{words} takes a file and outputs the words in it, one per a line
Warm-up Task (2/2)

- File too large for memory, but all \(\langle \text{word, count} \rangle\) pairs fit in memory.

- `words(doc.txt) | sort | uniq -c`
  - where `words` takes a file and outputs the words in it, one per a line

- It captures the essence of MapReduce: great thing is that it is naturally parallelizable.
MapReduce Overview

- words(doc.txt) | sort | uniq -c
MapReduce Overview

- words(doc.txt) | sort | uniq -c

- Sequentially read a lot of data.
MapReduce Overview

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- **Map**: extract something you care about.

![MapReduce Diagram]
MapReduce Overview

- `words(doc.txt) | sort | uniq -c`

- Sequentially **read** a lot of data.

- **Map**: extract something you care about.

- **Group by key**: sort and shuffle.
MapReduce Overview

- `words(doc.txt) | sort | uniq -c`

- Sequentially **read** a lot of data.

- **Map**: **extract** something you care about.

- **Group by key**: **sort** and **shuffle**.

- **Reduce**: aggregate, summarize, filter or transform.
MapReduce Overview

- `words(doc.txt) | sort | uniq -c`

- Sequentially read a lot of data.

- **Map**: extract something you care about.

- **Group by key**: sort and shuffle.

- **Reduce**: aggregate, summarize, filter or transform.

- Write the result.
Consider doing a word count of the following file using MapReduce:

Hello World Bye World
Hello Hadoop Goodbye Hadoop
The map function reads in words one a time and outputs \((word, 1)\) for each parsed input word.

The map function output is:

\[
\begin{align*}
(Hello, 1) \\
(World, 1) \\
(Bye, 1) \\
(World, 1) \\
(Hello, 1) \\
(Hadoop, 1) \\
(Goodbye, 1) \\
(Hadoop, 1)
\end{align*}
\]
The **shuffle** phase between **map** and **reduce** phase creates a list of values associated with each key.

The **reduce** function **input** is:

- (Bye, (1))
- (Goodbye, (1))
- (Hadoop, (1, 1))
- (Hello, (1, 1))
- (World, (1, 1))
The **reduce** function sums the numbers in the list for each key and outputs \((\text{word}, \text{count})\) pairs.

The output of the reduce function is the output of the MapReduce job:

\[
\begin{align*}
\text{(Bye, 1)} \\
\text{(Goodbye, 1)} \\
\text{(Hadoop, 2)} \\
\text{(Hello, 2)} \\
\text{(World, 2)}
\end{align*}
\]
public static class MyMap extends Mapper<...> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
}
public static class MyReduce extends Reducer<> {
    public void reduce(Text key, Iterator<?> values, Context context) throws IOException, InterruptedException {
        int sum = 0;

        while (values.hasNext())
            sum += values.next().get();

        context.write(key, new IntWritable(sum));
    }
}
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(MyMap.class);
    job.setReducerClass(MyReduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
}
MapReduce Execution

MapReduce Weaknesses

- MapReduce programming model has not been designed for complex operations, e.g., data mining.
MapReduce Weaknesses

- **MapReduce programming model** has not been designed for **complex** operations, e.g., data mining.

![MapReduce Diagram]

- Very **expensive**, i.e., always goes to disk and HDFS.
Solution?

Spark
Spark

- Extends MapReduce with more operators.
- Support for advanced data flow graphs.
- In-memory and out-of-core processing.

![Data Flow Diagram]

Hadoop  Spark
Spark vs. Hadoop

![Diagram showing the process flow between HDFS Read and Write for Iter. 1 and Iter. 2, with a continuation symbol indicating more iterations.](image-url)
Spark vs. Hadoop

![Diagram comparing Spark and Hadoop processes](image-url)
Spark vs. Hadoop
Spark vs. Hadoop
Resilient Distributed Datasets (RDD)

- Immutable collections of objects spread across a cluster.
- An RDD is divided into a number of partitions.
- Partitions of an RDD can be stored on different nodes of a cluster.
What About Streaming Data?
Many applications must process large streams of live data and provide results in real-time.
Motivation

- Many applications must process large streams of live data and provide results in real-time.

- Processing information as it flows, without storing them persistently.
Motivation

- Many applications must process large streams of live data and provide results in real-time.

- Processing information as it flows, without storing them persistently.

- Traditional DBMSs:
  - Store and index data before processing it.
  - Process data only when explicitly asked by the users.
DBMS vs. DSMS (1/3)

- **DBMS:** persistent data where updates are relatively infrequent.

- **DSMS:** transient data that is continuously updated.
DBMS vs. DSMS (2/3)

- **DBMS**: runs queries just once to return a complete answer.

- **DSMS**: executes standing queries, which run continuously and provide updated answers as new data arrives.
Despite these differences, DSMSs resemble DBMSs: both process incoming data through a sequence of transformations based on SQL operators, e.g., selections, aggregates, joins.
DSMS

- **Source**: produces the incoming information flows
- **Sink**: consumes the results of processing
- **IFP engine**: processes incoming flows
- **Processing rules**: how to process the incoming flows
- **Rule manager**: adds/removes processing rules
What About Graph Data?
Large graphs need large-scale processing.

A large graph either cannot fit into memory of single computer or it fits with huge cost.
The platforms that have worked well for developing parallel applications are not necessarily effective for large-scale graph problems.

Why?
Graph Algorithms Characteristics

- Unstructured problems: difficult to partition the data
- Data-driven computations: difficult to partition computation
- Poor data locality
- High data access to computation ratio
Proposed Solution

融合发展

Graph-Parallel Processing

- Computation typically depends on the neighbors.
Graph-Parallel Processing

- Restricts the types of computation.
- New techniques to partition and distribute graphs.
- Exploit graph structure.
- Executes graph algorithms orders-of-magnitude faster than more general data-parallel systems.
Data-Parallel vs. Graph-Parallel Computation

Data-Parallel

Graph-Parallel

Table
Row
Row
Row
Row
Result

Property Graph
Think as a vertex.

Each vertex computes individually its value: in parallel.

Each vertex can see its local context, and updates its value accordingly.
Summary
Scale-out vs. Scale-up
Summary

- Scale-out vs. Scale-up

- How to store data?
  - Distributed file systems: HDFS
  - NoSQL databases: HBase, Cassandra, ...
Summary

▶ Scale-out vs. Scale-up

▶ How to store data?
  • Distributed file systems: HDFS
  • NoSQL databases: HBase, Cassandra, ...

▶ How to process data?
  • Batch data: MapReduce, Spark
  • Streaming data: Spark stream, Flink, Storm, S4
  • Graph data: Giraph, GraphLab, GraphX, Flink
Questions?