NoSQL and Big Data Systems

Parallel Systems

- A coarse-grain parallel machine → a small number of powerful processors
- A massively parallel or fine grain parallel machine → thousands of smaller processors.
- We see a variety of mixes of these today, especially with the rise of multi-core machines

- Two main performance measures:
  - throughput --- the number of tasks that can be completed in a given time interval
  - response time --- the amount of time it takes to complete a single task from the time it is submitted
Speed-Up and Scale-Up

- **Speedup**: a fixed-sized problem executing on a small system is given to a system which is N-times larger.
  - Measured by:
    \[ \text{speedup} = \frac{\text{small system elapsed time}}{\text{large system elapsed time}} \]
  - Speedup is linear if equation equals N.

- **Scaleup**: increase the size of both the problem and the system
  - N-times larger system used to perform N-times larger job
  - Measured by:
    \[ \text{scaleup} = \frac{\text{small system small problem elapsed time}}{\text{big system big problem elapsed time}} \]
  - Scale up is linear if equation equals 1.

Factors Limiting Speedup and Scaleup

- **Sequential computation**: Some parts may not be parallelizable
  - Amdahl's Law: If "p" is the fraction of the task that can be parallelized, then the best speedup you can get is:
    \[ \frac{1}{(1-p)+(p/n)} \]
  - If "p" is 0.9, the best speedup is 10

- **Startup costs**: Cost of starting up multiple processes may dominate computation time, if the degree of parallelism is high.

- **Interference**: Processes accessing shared resources (e.g., system bus, disks, or locks) compete with each other, thus spending time waiting on other processes, rather than performing useful work.

- **Skew**: Increasing the degree of parallelism increases the variance in service times of parallelly executing tasks. Overall execution time determined by slowest of parallely executing tasks.
What about “Distributed” Systems?

- Over a wide area network
- Typically not done for performance reasons
  - For that, use a parallel system
- Done because of necessity
  - Imagine a large corporation with offices all over the world
  - Or users distributed across the globe
  - Also, for redundancy and for disaster recovery reasons

![Distributed System Diagram](image)

**Figure 20.9** A distributed system.

Distributed Systems

- Over a wide area network
- Typically not done for performance reasons
  - For that, use a parallel system
- Done because of necessity
  - Imagine a large corporation with offices all over the world
  - Also, for redundancy and for disaster recovery reasons
- Lot of headaches
  - Especially if trying to execute transactions that involve data from multiple sites
    - Keeping the databases in sync
      - 2-phase commit for transactions uniformly hated
    - Autonomy issues
      - Even within an organization, people tend to be protective of their unit/department
  - Locks/Deadlock management
  - Works better for query processing
    - Since we are only reading the data
Parallel or Distributed Systems

- **Key Questions from Data Management Perspective:**
  - How to partition (or “shard”) data across a collection of storage devices/machines
  - How to execute an “operation” across a group of computers
    - In different configurations (shared-memory vs shared-disk vs shared-nothing vs NUMA)
    - Trade-offs and bottlenecks can be vastly different
  - How to execute an “update” across a group of computers
    - Need to ensure consistency
  - How to deal with “failures”

Data Partitioning (sharding)

- Partition a relation or a dataset across machines
  - Typically through “hashing”
- **Advantages:**
  - In-memory computation: data fits in memory across machines
  - Parallelism: simple read/write queries can be distributed across machines
- **Disadvantages:**
  - Complex queries: require combining data across all partitions, especially “joins” are tricky

Partitions of R (Not different relations)

- Machine 1 can directly read R1, S1
  - If it wants R2, Machine 2 must read it and send it to Machine 1
Data Replication

- A data item (file, relation, relation fragment, object, tuple) is replicated if it is stored redundantly in two or more sites.
- Advantages:
  - Availability: failures can be handled through replicas.
  - Parallelism: queries can be run on any replica.
  - Reduced latency: queries can go to the “closest” replica.
- Disadvantages:
  - Increased cost of updates: both computation as well as latency.
  - Increased complexity of concurrency control: need to update all copies of a data item/tuple.

![Diagram of three machines](image)

Read queries can go to any machine. Write queries must go to “all” machines (if we want consistency).

- e.g., what if Application 1 writes to Machine 1, and Application 2 sends its write to Machine 3?
  -- May result in an inconsistent state.

Data Sharding + Replication

- Many data management systems today combine both:
  - Shard a dataset/file/relation and distribute across machines.
  - Replicate each of the pieces multiple times.
- This may be done:
  - In a data center with very fast networks, or
  - In a wide-area setting with slower networks and higher latencies.
- So need to worry about:
  - Efficient execution of complex queries.
  - Consistency for updates.
  - Recovery from failures.
Failures

- Need to consider:
  - Disk failures: one of the disks (hard drives or SSDs) fails
    - Not uncommon with 10's of thousands of disks
  - Network failures: machines may not be able to talk to each other
  - Machine failure: a machine crashes during the execution of a query or a transaction

- Required guarantees:
  - Shouldn’t lose any data if a disk fails
  - Consistency (when making updates) shouldn’t be affected if one of the involved machines fails
    - Or if machines are not able to talk to each other
  - Shouldn’t have to restart a complex analytics task entirely if one of the involved machines fails

Two Primary Use Cases (again)

- **OLTP-like**
  - Simple queries, but lots of updates
  - Need to support distributed users
  - Need to support non-relational data (e.g., graphs, JSONs)
  - Need to scale fast (10 users to 10s of Millions of Users)
  - Need to work well in 3-tier Web Apps
  - Need to support fast schema changes

- **OLAP-like**
  - Complex analysis on large volumes of data
  - Often no “real-time” component, and no updates
  - Mostly non-relational data (images, webpages, text, etc)
  - Tasks often procedural in nature (analyse webpages for searching, data cleaning, ML)
Examples of Systems

- **Huge variety in current systems**
  - different types of data models supported
    - Files/Objects (HDFS, AWS S3), Document (MongoDB), Graph (Neo4j), Wide-table (Cassandra, DynamoDB), Multi-Model (Azure CosmosDB)
  - different types of query languages or frameworks or workloads
    - SQL (Snowflake, Redshift, …), MongoDB, Cassandra, DataFrames (Spark), MapReduce (Hadoop), TensorFlow for ML, …
  - different environmental assumptions
    - Distributed vs parallel, disks or in-memory only, single-machine or not, streaming or static, etc.
  - different performance focus and/or guarantees
    - e.g., consistency guarantees in a distributed setting differ quite a bit

- **Many of these systems work with each**
  - e.g., Spark can read data from most of the storage systems
  - Interoperability increasing a requirement

What We Will Cover

- **Map Reduce**
  - Grandfather of most current approaches

- **Apache Spark**
  - Current leader in big data (OLAP-style) frameworks
  - Supports many query/analysis models, including a light version of SQL
  - Used for Assignment 9

- **MongoDB**
  - Perhaps the most popular NoSQL system, uses a "document" (JSON) data model
  - Focus primarily on OLTP
  - Doesn't really support joins (some limited ability today) – have to do that in the app

- **How to “Parallelize” Operations**
  - Useful to understand how Spark and other systems actually work
  - Often times you have to build these in the application layer
  - The original MapReduce framework
    - Led to development of much work on large-scale data analysis (OLAP-style)
    - Basically a way to execute a group-by at scale on non-relational data

- **Hadoop Distributed File System (briefly)**
  - A key infrastructure piece, with no real alternative
  - Basic file system interface, with replication and redundancy built in for failures

- Quick overview of other NoSQL data models
MapReduce Framework

- Provides a fairly restricted, but still powerful abstraction for programming

- Programmers write a pipeline of functions, called map or reduce
  - map programs
    - inputs: a list of “records” (record defined arbitrarily – could be images, genomes etc…)
    - output: for each record, produce a set of “(key, value)” pairs
  - reduce programs
    - input: a list of “(key, {values})” grouped together from the mapper
    - output: whatever

- Both can do arbitrary computations on the input data as long as the basic structure is followed
Word Count Example

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));

MapReduce Framework: Word Count

input files  mappers  intermediate files  reducers  output files

abacdb  (a, 1)  (a, 1)  (a, 8)  (a, 8)
bcdaaa  (b, 1)  (a, 1)  (c, 5)  (c, 5)
ababab  (a, 1)  (c, 1)  (a, 1)  (a, 1)
ccccc  (c, 1)  (a, 1)  (a, 1)  (a, 1)
More Efficient Word Count

input files mappers intermediate files reducers output files

a b a c d b (a, 2) (a, 2)
(b, 2) (a, 3)
(c, 1) (c, 1)
(d, 1) (c, 5)

b c d a a a

(a, 8)
(c, 5)

a b a b a b

(b, 6)
(d, 2)

“mapper-side” combiner

Apache Spark

- Map-Reduce on steroids

- Book Chapters
  - 10.4 (7TH EDITION) covers this topic, but Spark programming guide is a better resource
  - Assignment will refer to the programming guide

- Key topics:
  - A Resilient Distributed Dataset (RDD)
  - Operations on RDDs
Spark

- Open-source, distributed cluster computing framework
- Much better performance than Hadoop MapReduce through in-memory caching and pipelining
- Originally provided a low-level RDD-centric API, but today, most of the use is through the “Dataframes” (i.e., relations) API
  - Dataframes support relational operations like Joins, Aggregates, etc.

Resilient Distributed Dataset (RDD)

- **RDD** = Collection of records stored across multiple machines in-memory

Drivers (apps)
- Come and go
- Not fault-tolerant

Worker Nodes
- Always running

 RDD Manipulation Commands

Results – typically at the end
Spark

- Why “Resilient”?
  - Can survive the failure of a worker node
  - Spark maintains a “lineage graph” of how each RDD partition was created
  - If a worker node fails, the partitions are recreated from its inputs
  - Only a small set of well-defined operations are permitted on the RDDs
    - But the operations usually take in arbitrary “map” and “reduce” functions

- Fault tolerance for the “driver” is trickier
  - Drivers have arbitrary logic (cf., the programs you are writing)
  - In some cases (e.g., Spark Streaming), you can do fault tolerance
  - But in general, driver failure requires a restart

Example Spark Program

```python
from pyspark import SparkContext

sc = SparkContext("local", "Simple App")
textFile = sc.textFile("README.md")

counts = textFile
  .flatMap(lambda line: line.split(" "))
  .map(lambda word: (word, 1))
  .reduceByKey(lambda a, b: a + b)

print(counts.take(100))
```

Initialize RDD by reading the textFile and partitioning
If textFile stored on HDFS, it is already partitioned – just read each partition as a separate RDD partition

Split each line into words, creating an RDD of words
For each word, output (word, 1), creating a new RDD
Do a group-by SUM aggregate to count the number of times each word appears

Retrieve 100 of the values in the final RDD
Spark

- Operations often take in a "function" as input
- Using the inline “lambda” functionality
  \[ \text{flatMap}(\lambda \text{line}: \text{line.split(" ")}) \]
- Or a more explicit function declaration
  \[
  \text{def split(line):
  \hspace{1em} \text{return line.split(" ")}
  \text{flatMap(split)}
  \]
- Similarly "reduce" functions essentially tell Spark how to do pairwise aggregation
  \[ \text{reduceByKey}(\lambda a, b: a + b) \]
  - Spark will apply this to the dataset pair of values at a time
  - Difficult to do something like “median”

Spark: Map

InputRDD: \[x_1, x_2, \ldots, x_n\]

map(lambda x: x + 1)

OutputRDD: \[x_1+1, x_2+1, \ldots, x_n+1\]

x_1, x_2, \ldots can be anything, including documents, images, text files, tuples, dicts, etc.

InputRDD: \[x_1, x_2, \ldots, x_n\]

map(fn)

OutputRDD: \[fn(x_1), fn(x_2), \ldots, fn(x_n)\]
Spark: flatMap

InputRDD: [(a1, b1), (a2, b2), …]  InputRDD: ['the little brown fox...', ...]

flatMap(lambda x: [x[0], x[1]])  flatMap(lambda x: x.split())

outputRDD: [a1, b1, a2, b2, ...]  outputRDD: ['the', 'little', 'brown', ...]

Spark: groupByKey

InputRDD: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]

InputRDD must be a collection of 2-tuples
Usually called (Key, Value) pairs
Value can be anything (e.g., dicts, tuples, bytes)

groupByKey()

outputRDD: [(a1, [b1, b3, b4, ...]), (a2, [b3, b5,...]), ...]
Spark: reduceByKey

InputRDD: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]

InputRDD must be a collection of 2-tuples
Usually called (Key, Value) pairs

def func(V1, V2):
    return V3

All of V1, V2, and V3 same type

"func" executed in parallel in a pairwise fashion

outputRDD: [(a1, …func(func(b1, b3), b4)...),
(a2, …func(func(b2, b5), ...)…),]

Spark: join

InputRDD1: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]
InputRDD2: [(a1, c1), (a2, c2), (a1, c3), (a1, c4), (a2, c5)…]

InputRDD1 and InputRDD2 both must be a collection of 2-tuples

inputRDD1.join(inputRDD2)

outputRDD: [(a1, (b1, c1)),
(a1, (b1, c3)),
(a1, (b1, c4)),
(a1, (b1, c4)),
…]
Spark: cogroup

InputRDD1: \([(a_1, b_1), (a_2, b_2), (a_1, b_3), (a_1, b_4), (a_2, b_5)\ldots]\)
InputRDD2: \([(a_1, c_1), (a_2, c_2), (a_1, c_3), (a_1, c_4), (a_2, c_5)\ldots]\)

InputRDD1 and InputRDD2 both must be a collection of 2-tuples

inputRDD1.cogroup(inputRDD2)

outputRDD: \[
(a_1, [(b_1, b_3, b_4, \ldots), (c_1, c_3, c_4, \ldots)],
(a_2, [(b_2, b_5, \ldots), (c_2, c_5, \ldots)]), \ldots
\]

RDD Operations
### Dataframes Example

```python
def basic_df_example(spark):
    # Example on:create_df
    # spark is an existing SparkSession
    df = spark.read.json("examples/src/main/resources/people.json")
    # Displays the context of the DataFrame to stdout
    df.show()
    # +----+-----+
    # | age | name |
    # +----+-----+
    # | null | Michael |
    # | 30   | Andy    |
    # | 19   | Justin  |
    # +----+-----+
    # Example off:create_df
    # Example on:untyped_ops
    # spark, df are from the previous example
    # Print the schema in a tree format
    df.printSchema()
    # root
    # |-- age: long (nullable = true)
    # |-- name: string (nullable = true)
    # Select only the "name" column
    df.select("name").show()
    # +---+
    # | name|
    # +---+
    # | Michael|
    # | Andy |
    # | Justin|
    # +---+
    # Select everybody, but increment the age by 1
    df.select(df["name"], df["age"] + 1).show()
    # +----+------+
    # | name | age + 1 |
    # +----+------+
    # | null | 31    |
    # | Andy | 31    |
    # | Justin| 20    |
    # +----+------+
    # Example off:untyped_ops
    sqlDF = spark.sql("SELECT * FROM people")
    sqlDF.show()
    # +----+-----+
    # | age | name |
    # +----+-----+
    # | null | Michael |
    # | 30   | Andy    |
    # | 19   | Justin  |
    # +----+-----+
    # Example off:run_sqls
    # Example on:global_temp_views
    # Register the DataFrame as a global temporary view
    df.createGlobalTempView("people")
    # Global temporary view is tied to a system preserved database
    "global_temp".
    spark.sql("SELECT * FROM global_temp.people").show()
    # +----+-----+
    # | age | name |
    # +----+-----+
    # | null| Michael |
    # | 30  | Andy    |
    # | 19  | Justin  |
    # +----+-----+
```

### Summary

- Spark is a popular and widely used framework for large-scale computing
- Simple programming interface
  - You don’t need to typically worry about the parallelization
  - That’s handled by Spark transparently
  - In practice, may need to fiddle with number of partitions etc.
- Managed services supported by several vendors including Databricks (started by the authors of Spark), Cloudera, etc.
- Many other concepts that we did not discuss
  - Shared accumulator and broadcast variables
  - Support for Machine Learning, Graph Analytics, Streaming, and other use cases
- Alternatives include: Apache Tez, Flink, and several others