CS143 Non-Relational Database (MongoDB)

Professor Junghoo “John” Cho

JSON (JavaScript Object Notation)

- Syntax to represent objects in JavaScript
  - [{ “x”: 3, “y”: “Good”}, { “x”: 4, “y”: “Bad” }]
- One of the most popular data-exchange formats over Internet
  - As JavaScript gained popularity, JSON’s popularity grew
  - Simple and easy to learn
  - Others popular formats include XML, CSV, …

Basic JSON Syntax

- Supports basic data types like numbers and strings, as well as arrays and “objects”
- Double quotes for string: “Best”, “UCLA”, “Worst”, “USC”
- Square brackets for array: [1, 2, 3, “four”, 5]
- Objects: (attribute, name) pairs. Use curly braces
  - { “sid”: 301, “name”: “James Dean” }
- Things can be nested
  - { “sid” : 301,
“name”: { “first”: “James”, “last”: “Dean” },
“classes”: [“CS143”, “CS144”] }

RDBMS for JavaScript Object Persistence

- JavaScript applications need a “persistence layer” to store and retrieve JavaScript object
- Traditionally (until mid 2010) this was done with RDBMS
  - RDBMS as massive, safe, efficient, multi-user storage engine
- Q: How can we store JavaScript object in RDB?
- “Impedance mismatch”: Two choices
  1. Store object’s JSON as a string in a column
  2. “Normalize” the object into set of relations
- Q: Pros and cons of each approach?
- Q: Can we just create “native database” for JSON?

MongoDB

- Database for JSON objects
  - Perfect as a simple persistence layer for JavaScript objects
  - “NoSQL database”
- Data is stored as a collection of documents
  - Document: (almost) JSON object
  - Collection: group of “similar” documents
- Analogy
  - Document in MongoDB ~ row in RDB
  - Collection in MongoDB ~ table in RDB

MongoDB “Document”

```
{
  "_id": ObjectId(8df38ad8902c),
  "title": "MongoDB",
  "description": "MongoDB is NoSQL database",
  "tags": ["mongodb", "database", "NoSQL"],
  "likes": 100,
  "comments": [
    { "user":"lover", "comment": "Perfect!" },
    { "user":"hater", "comment": "Worst!" }
  ]
}
```

- _id field: primary key
  - May be of any type other than array
  - If not provided, automatically added with a unique ObjectId value
- Stored as BSON (Binary representation of JSON)
  - Supports more data types than JSON
  - Does not require double quotes for field names

MongoDB “Philosophy”

- Adopts JavaScript “laissez faire” philosophy
  - Don’t be too strict! Be accommodating! Handle user request in a “reasonable” way
- Schema-less: no predefined schema
  - Give me anything. I will store it anywhere you want
  - One collection will store documents of *any* kind with no complaint
- No need to “plan ahead”
  - A “database” is created when a first collection is created
  - A “collection” is created when a first document is inserted
- Both blessing and curse

MongoDB Demo

```
show dbs;
use demo;
show collections;
db.books.insertOne({title: "MongoDB", likes: 100});
db.books.find();
show collections;
show dbs;
db.books.insertMany([{title: "a"}, {name: "b"}]);
db.books.find();
db.books.find({likes: 100});
db.books.find({likes: {$gt: 10}});
db.books.updateOne({title: "MongoDB"}, {$set: { likes: 200 }});
db.books.find();
db.books.deleteOne({title: "a"});
db.books.drop();
show collections;
show dbs;
```

Basic MongoDB Commands (1)

- mongo: start MongoDB shell
- use <dbName>: use the database
- show dbs: show list of databases
- show collections: show list of collections
- db.colName.drop(): delete `colName` collection
- db.dropDatabase(): delete current database

Basic MongoDB Commands (2)

- CRUD operations
  - insertOne(), insertMany()
  - findOne(), find()
  - updateOne(), updateMany()
  - deleteOne(), deleteMany()
- Insertion: insertX(doc(s))
  - db.books.insertOne({title: "MongoDB", likes: 100})
  - db.books.insertMany([{title: "a"}, {title: "b"}])

Basic MongoDB Commands (3)

- Retrieval: findX(condition)
  - db.books.findOne({likes: 100})
  - db.books.find({$and: [{likes: {$gte: 10}}, {likes: {$lt: 20}}]})
    - Other Boolean/comaprision operators: $or, $not, $gt, $ne, …
- Update: updateX(condition, update_operation)
  - db.books.updateOne({title: "MongoDB"}, {$set: {title: "MongoDB II"}})
  - db.books.updateMany({title: "MongoDB"}, {$inc: {likes: 1}})
    - Other update operators: $mul (multiply), $unset (remove field), …
- Deletion: deleteX(condition)
  - db.books.deleteOne({title: "MongoDB"})
  - db.books.deleteMany({likes: {$lt: 100}})

MongoDB Aggregates

- MongoDB supports complex queries through “aggregates”
- MongoDB aggregates are very much like SQL SELECT queries
  - stages – SQL SELECT clause
  - pipeline – SQL SELECT statement

MongoDB Aggregates: Example

- { _id: 1, cust_id: "a", status: "A", amount: 50 }
  { _id: 2, cust_id: "a", status: "A", amount: 100 }
  { _id: 3, cust_id: "c", status: "D", amount: 25 }
  { _id: 4, cust_id: "d", status: "C", amount: 125 }
  { _id: 5, cust_id: "d", status: "A", amount: 25 }

```
db.orders.aggregate([
  { $match: { status: "A" } },
  { $group: {
       _id: "$cust_id",
       total: { $sum: "$amount" },
       count: { $sum: 1 }
     }
  },
  { $sort: { total: -1 } }
])
```

- Equivalent to SQL SELECT
  - Just $match is fine, for example
  - In $group stage, _id is “group by attributes”

Common Aggregate Stages

- \$match ≈ WHERE
- \$group ≈ GROUP BY
- \$sort ≈ ORDER BY
- \$limit ≈ FETCH FIRST
- \$project ≈ SELECT
- \$unwind: replicate document per every element in the array
  - {\$unwind: "y" }: {"x": 1, "y": [1, 2] } -> {"x": 1, "y": 1}, {"x": 1, "y": 2 }
- \$lookup: "look up and join" another document based on the attribute value
  - {\$lookup: { from: <collection to join>, localField: <local join attr>, foreignField: <remote join attr>, as: <output field name> }}
  - Matching documents are returned as an array in <output field name>

More on MongoDB aggregates

- Short tutorial: https://studio3t.com/knowledge-base/articles/mongodb-aggregation-framework/
- Reference: https://docs.mongodb.com/manual/reference/method/db.collection.aggregate/

MongoDB vs RDB

- MongoDB document
  - Preserves structure
    - Nested objects
  - Potential redundancy
  - Restructuring or combining data is complex and inefficient
- MongoDB: “laissez faire”
  - No explicit db/collection creation
  - No schema. Anything is fine

- RDB tuple
  - “Flattens” data
    - Set of flat rows
  - Removes redundancy
  - Data can be easily “combined” using relational operators
- RDB: “Straight-jacket”
  - Declare everything before use
  - Reject if not compliant

More on MongoDB

- We learned just the basic
- MongoDB has many more features
  - Transactions
  - Replication
  - (Auto)sharding
  - …
- Read MongoDB documentation and online tutorials to learn more

CS143
Map Reduce (Spark)

Professor Junghoo "John" Cho

Distributed Analytics using Cluster

- Often, our data is non-relational (e.g., flat file) and huge
  - Billions of query logs
  - Billions of web pages
  - ...
- Q: Can we perform analytics on large data quickly using thousands of machines?

Example 1: Search Log Analysis

- Log of billions of queries. Count frequency of each query
  - Input query log:

    cat,time,userid1,ip1,referrer1

    dog,time,userid2,ip2,referrer2

    ...
  - Output query frequency:

    cat 200000

    dog 120000

    ...
- Q: How can we perform this task? How can we parallelize it?

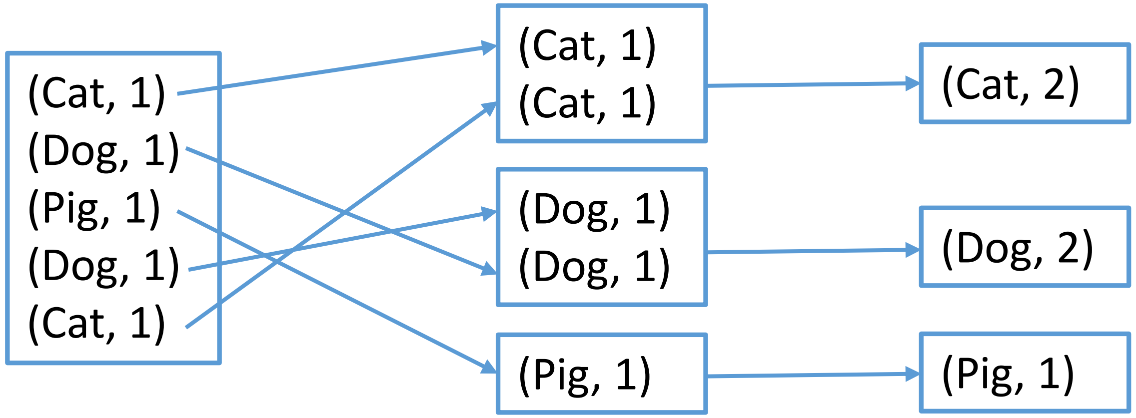
Example 1: Search Log Analysis (1)

- Step 1: “Transform” each line of query log into (query, 1)

Cat, 12:30, 1.34.24.6, … Dog, 12:30, 193.42.34.5, … Pig, 12:31, 213.12.6.26, … Dog, 12:32, 31.63.34.23, … Cat, 12:33, 1.46.23.642, ..	→	(Cat, 1) (Dog, 1) (Pig, 1) (Dog, 1) (Cat, 1)

Example 1: Search Log Analysis (2)

- Step 2: Collect all tuples with the same query and "aggregate" them



- Q: How can we parallelize the two steps?

Example 1: Search Log Analysis (3)

- Step 1: “Transform” each line of query log into (query, 1)

Cat, 12:30, 1.34.24.6, …
Dog, 12:30, 193.42.34.5, …
Pig, 12:31, 213.12.6.26, …
Dog, 12:32, 31.63.34.23, …
Cat, 12:33, 1.46.23.642, ..

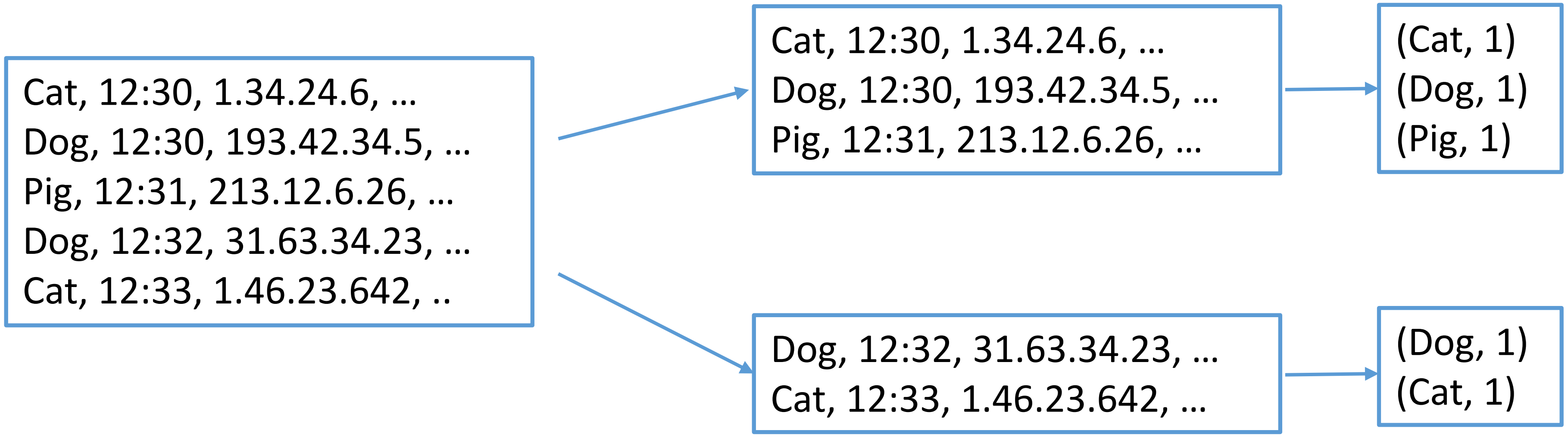
→

(Cat, 1)
(Dog, 1)
(Pig, 1)
(Dog, 1)
(Cat, 1)

- Q: Can the transformation of each line done independently of each other?

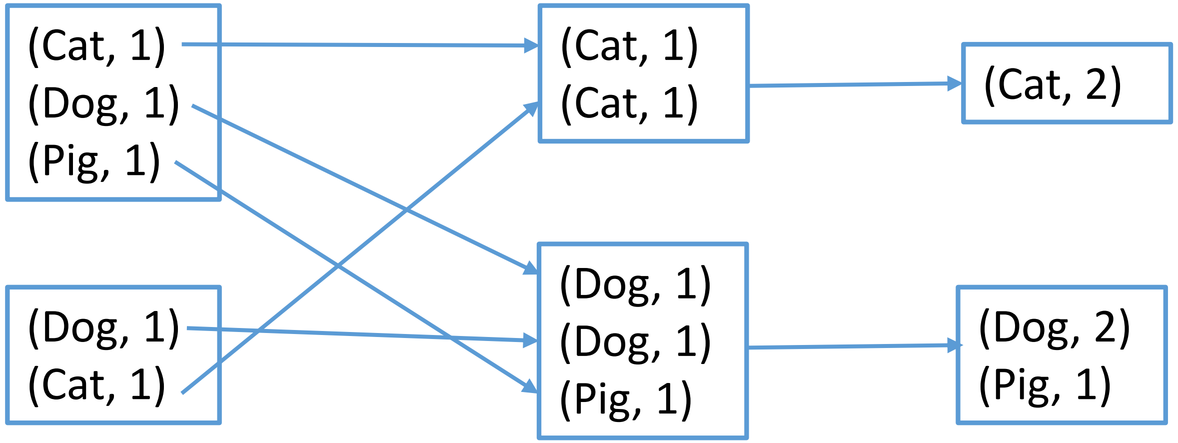
Example 1: Search Log Analysis (4)

- Step 1: For parallel processing
  - Split input data into multiple independent chunks
  - Move each chunk to separate machine
  - Perform “transformation” on multiple machines in parallel



Example 1: Search Log Analysis (5)

- Q: How do we parallelize the second "aggregation" step?
- Step 2: For parallel processing
  - Move the tuples with the same query to the same machine
  - Perform aggregation on multiple machine in parallel



Example 2: Web Indexing

- 1 billion pages. Build “inverted index”
  - Input documents:

    1: cat chases dog

    2: dog loves cat

    ...
  - Output index:

    cat 1,2,5,10,20

    dog 1,2,3,8,9
- Q: How can we do this?

Example 2: Web Indexing (1)

- Step 1: “Transform” every document into (word, doc_id) tuples

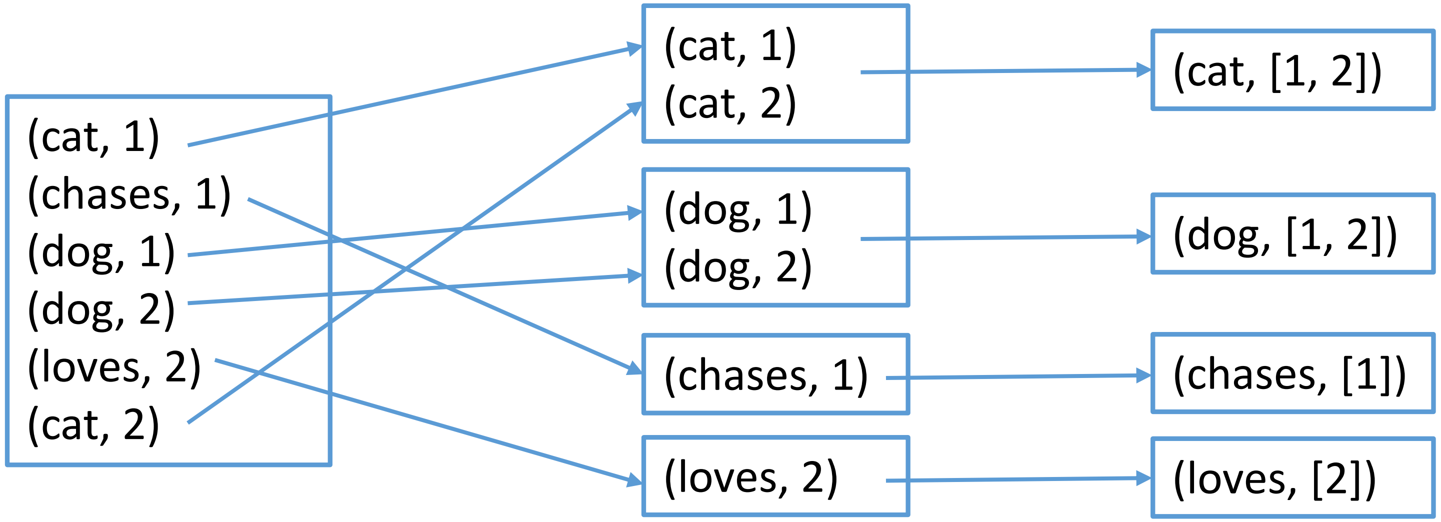
1: cat chases dog
2: dog loves cat

→

(cat, 1)
(chases, 1)
(dog, 1)
(dog, 2)
(loves, 2)
(cat, 2)

Example 2: Web Indexing (2)

- Step 2: Collect all tuples with the same word and "aggregate" (or concatenate) the doc_id's



- Q: How can we parallelize the two steps on multiple machines?

Example 2: Web Indexing (3)

- Step 1: “Transform” every document into (word, doc_id) tuples

1: cat chases dog
2: dog loves cat

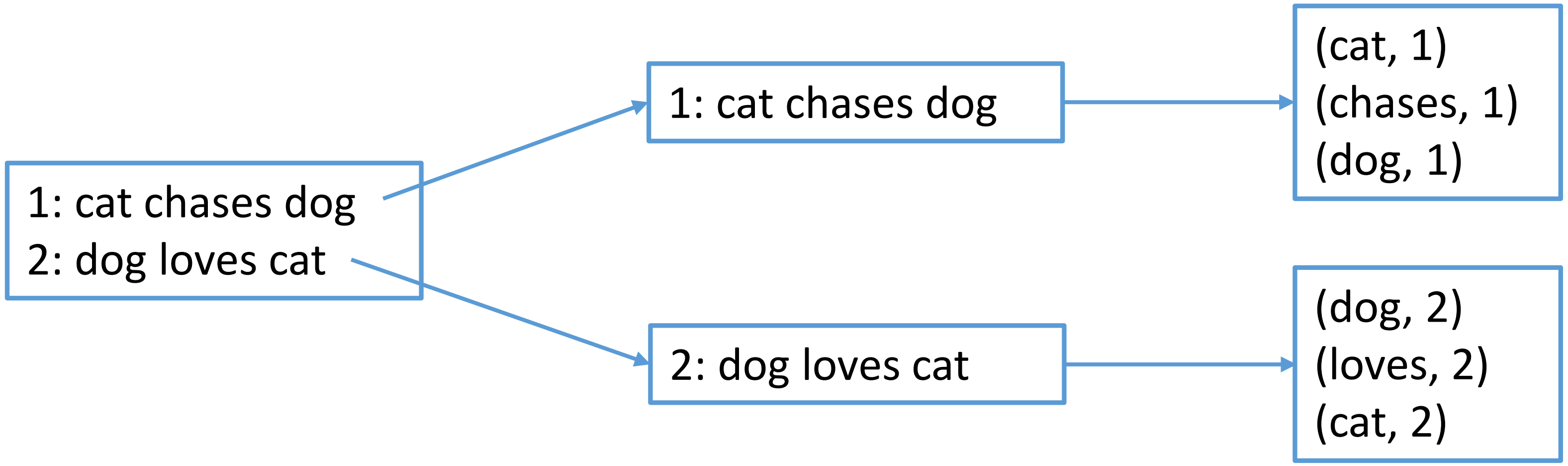
→

(cat, 1)
(chases, 1)
(dog, 1)
(dog, 2)
(loves, 2)
(cat, 2)

- Q: Can the transformation of each document be done independently of each other?

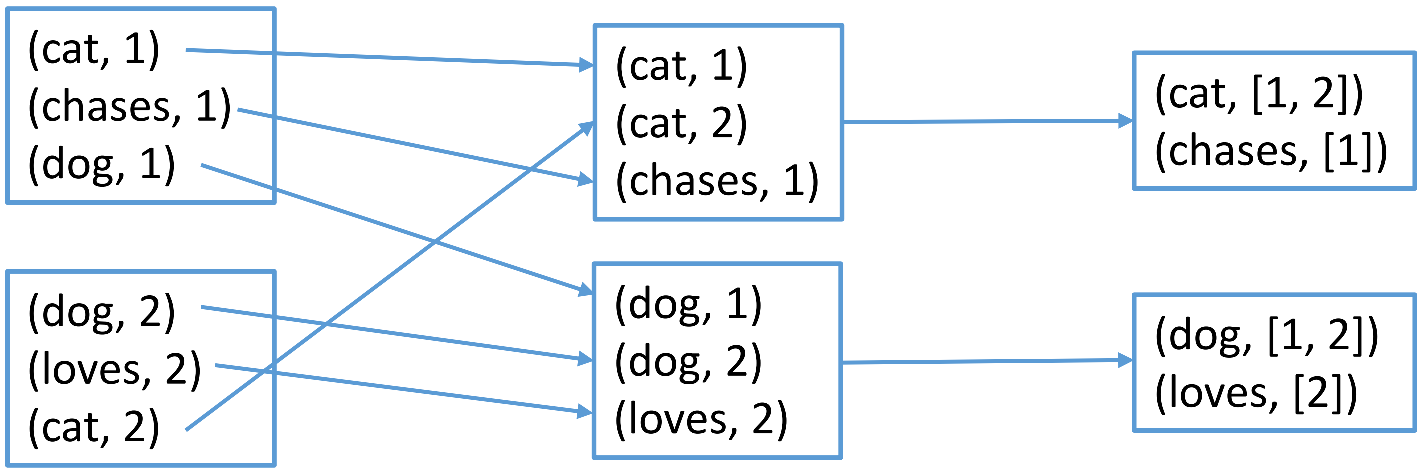
Example 2: Web Indexing (4)

- Step 1: For parallel processing
  - Split input data into multiple independent chunks
  - Move each chunk to separate machine
  - Perform “transformation” on multiple machines in parallel



Example 2: Web Indexing (5)

- Q: How can we parallelize second “concatenation step”?
- Step 2: For parallel processing
  - Move the tuples with the same word to the same machine
  - Perform aggregation on multiple machine in parallel



Generalization (1)

- Input data consists of multiple independent units
  - Each line of query log
  - Each web page
- Partition input data into multiple “chunks” and distribute them to multiple machines
- Transformation/map input into (key, value) tuples
  - Query log: query_log_line → (query, 1)
  - Indexing: web_page → ($word_1$, page_id), ($word_2$, page_id), …
- Reshuffle tuples of the same key to the same machine
- Aggregate/reduce the tuples of same keys
  - Query log: (query, 1), (query, 1), … → (query, count)
  - Indexing: (word, 1), (word, 3), … → (word, [1, 3, …])
- Collect and output the aggregation results

Generalization (2)

- The two examples are almost the same except
  - "The mapping function"
    - Query log: query_log_line → (query, 1)
    - Indexing: web_page → ($word_1$, page_id), ($word_2$, page_id), …
  - "The reduction function"
    - Query log: (query, 1), (query, 1), … → (query, count)
    - Indexing: (word, 1), (word, 3), … → (word, [1, 3, …])

MapReduce Model

- Programmer provides
  1. Map function: "unit data" $\rightarrow (k', v'), (k'', v''), \ldots$
  2. Reduce function: $(k, v_1), (k, v_2), \ldots \rightarrow (k, \mathrm{aggr}(v_1, v_2, \ldots))$
- MapReduce handles the rest
  - Automatic data partition, distribution, and collection
  - Failure and speed-disparity handling
- Many systems exist supporting MapReduce model

Hadoop

- First open-source implementation of MapReduce and GFS (Google File System)
  - Implemented in Java
- User implements map and reduce functions as:
  - Mapper.map(key, value, output, reporter)
  - Reducer.reduce(key, value, output, reporter)

Spark

- Open-source cluster computing infrastructure
- Supports MapReduce and SQL
  - Supports data flow more general than simple MapReduce
- Input data is converted into RDD (resilient distributed dataset)
  - A collection of independent tuples
  - The tuples are automatically distributed and shuffled by Spark
- Supports multiple programming languages
  - Scala, Java, Python, …
  - Scala and Java are much more performant than others

Spark Example: Count words

```
lines = sc.textFile("input.txt")
words = lines.flatMap(lambda line: line.split(" "))
word1s = words.map(lambda word: (word, 1))
wordCounts = word1s.reduceByKey(lambda a,b: a+b)
wordCounts.saveAsTextFile("output")
```

Key Spark Functions

- Transformation: Convert RDD tuple into RDD tuple(s)
  - map(): convert one input tuple into one output tuple
  - flatMap(): convert one input into multiple output tuples
  - reduceByKey(): specify how two input “values” should be aggregated
  - filter(): filter out tuples based on condition
- Action: Perform “actions” on RDD
  - saveAsTextFile(): save RDD in a directory as text file(s)
  - collect(): create Python tuples from Spark RDD
  - textFile(): create RDD from text (each line becomes an RDD tuple)

What We Learned

- Large-scale data analytics on distributed cluster
- MapReduce model
- Spark