Distributed Key-Value Pairs (Pair RDDs)

Big Data Analysis with Scala and Spark

Heather Miller

Distributed Key-Value Pairs

In single-node Scala, key-value pairs can be thought of as ***maps***.
(Or *associative arrays* or *dictionaries* in JavaScript or Python)

Distributed Key-Value Pairs

In single-node Scala, key-value pairs can be thought of as ***maps***.
(Or *associative arrays* or *dictionaries* in JavaScript or Python)

While maps/dictionaries/etc are available across most languages, they perhaps aren't the most commonly-used structure in single-node programs. List/Arrays probably more common.

Most common in world of big data processing:
Operating on data in the form of key-value pairs.

- Manipulating key-value pairs a key choice in design of MapReduce

Distributed Key-Value Pairs

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

(2004 research paper)

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution

Distributed Key-Value Pairs

(2004 research paper)

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined

Distributed Key-Value Pairs (Pair RDDs)

Large datasets are often made up of unfathomably large numbers of complex, nested data records.

To be able to work with such datasets, it's often desirable to *project down* these complex datatypes into **key-value pairs**.

Distributed Key-Value Pairs (Pair RDDs)

```
{
   "definitions":{
      "firstname":"string",
      "lastname":"string",
      "address":{
         "type":"object",
         "properties":{
            "street_address":{
               "type":"string"
            },
            "city":{
               "type":"string"
            },
            "state":{
               "type":"string"
            }
         },
         "required":[
            "street_address",
            "city",
            "state"
         ]
      }
   }
}
```

Large datasets are often made up of unfathomably large numbers of complex, nested data records.

To be able to work with such datasets, it's often desirable to *project down* these complex datatypes into **key-value pairs**.

Example:
In the JSON record to the left, it may be desirable to create an RDD of `properties` of type:

```
RDD[(String, Property)] // where ‘String‘ is a key representing a city,
                        // and ‘Property‘ is its corresponding value.

case class Property(street: String, city: String, state: String)
```

where instances of `Properties` can be grouped by their respective cities and represented in a RDD of key-value pairs.

Distributed Key-Value Pairs (Pair RDDs)

Often when working with distributed data, it's useful to organize data into **key-value pairs**.

In Spark, distributed key-value pairs are "Pair RDDs."

Useful because: Pair RDDs allow you to act on each key in parallel or regroup data across the network.

Distributed Key-Value Pairs (Pair RDDs)

Often when working with distributed data, it's useful to organize data into **key-value pairs**.

In Spark, distributed key-value pairs are "Pair RDDs."

Useful because: Pair RDDs allow you to act on each key in parallel or regroup data across the network.

Pair RDDs have additional, specialized methods for working with data associated with keys. RDDs are parameterized by a pair are Pair RDDs.

```
RDD[(K,V)]  // <== treated specially by Spark!
```

Pair RDDs (Key-Value Pairs)

Key-value pairs are known as Pair RDDs in Spark.

When an RDD is created with a pair as its element type, Spark automatically adds a number of extra useful additional methods (extension methods) for such pairs.

Some of the most important extension methods for RDDs containing pairs (*e.g.*, `RDD[(K, V)]`) are:

```
def groupByKey(): RDD[(K, Iterable[V])]
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]
```

Pair RDDs (Key-Value Pairs)

Creating a Pair RDD

Pair RDDs are most often created from already-existing non-pair RDDs, for example by using the `map` operation on RDDs:

```
val rdd: RDD[WikipediaPage] = ...


val pairRdd = ???
```

Pair RDDs (Key-Value Pairs)

Creating a Pair RDD

Pair RDDs are most often created from already-existing non-pair RDDs, for example by using the `map` operation on RDDs:

```
val rdd: RDD[WikipediaPage] = ...

// Has type: org.apache.spark.rdd.RDD[(String, String)]
val pairRdd = rdd.map(page => (page.title, page.text))
```

Once created, you can now use transformations specific to key-value pairs such as `reduceByKey`, `groupByKey`, and `join`