Integrating Apache Spark and External Data Sources Using Hadoop Interfaces

Yong-Liang LI¹, a, Shu-Qiang YANG¹, b, *

¹School of computer, National University of Defense Technology, Changsha, China
a wildpassionlee@sina.com, b sqyang9999@126.com
*Corresponding author

Keywords: Integration, Apache Spark, Data sources, Hadoop interfaces.

Abstract. Requirements of data-processing are undergoing a profound transition with the dramatic increase of various application data. Along with this demands, sorts of storage systems with highly scalable are developed for large-scale data sources, and Apache Spark for immense amounts of data calculation has also captured attentions and excitement of the industry since release for its excellent performance. Technologies of combining Spark and external data sources have lots of potential to produce valuable data analysis platforms for solving a wide range of data-handling needs, accordingly it's very easy to shift from Hadoop to Spark application development since the industry pour a lot in these datastore system for Hadoop in the past and Spark can work well with Hadoop-supported systems. Paper delves into integration mechanism of spark and external data sources, such as NoSQL and relational databases (RDBs) and proposes a reference for tight and efficient integration of the two using Hadoop interfaces.

Introduction

Due to the explosive increase of raw data volume from various fields in recent years, new processing technologies are constantly evolving. MapReduce [1] is a distributed parallel computing frameworks devised to large-scale datasets over a cluster of commercial machines. The computing is divided into two periods: map and reduce. The former takes in an input pair and outputs a set of interim key/value pairs. The latter reduces the interim results from multiple nodes to produces the end result. The strength of MapReduce has spawned chains of systems that implement or extend it. Hadoop [2], composed of two core components: Hadoop Distributed File System (HDFS) and MapReduce, is a mature implementation of the model for massively parallel and distributed data processing in the industry. Hadoop exploits HDFS as the data management and distribution foundation, it automatically cuts data into chunks and spreads them on the whole cluster.

However, Hadoop forces writing data to disk at the end of each job execution and reading it again from disk for the next. The framework is limited to a very rigid data flow model and is unsuitable for many real time and iterative applications [3], for example, streaming processing, and graph processing. Spark develops the MapReduce framework to cater for multiple application requirements with more computational paradigms, and it contains widespread workflows which were originally done as technical systems based on Hadoop [4]. Spark adopts the strategy of in-memory caching intermediate results for later reuse to enhance efficiency and then, is suitable for complex analysis. For instance, applications such as iterative machine learning or interactive data analysis can benefit from reusing a data set cached in memory for multiple processing tasks.

Inadequate performance of traditional commercial databases in dealing with large-scale storage and high concurrent access, hastens the development of kinds of NoSQL databases such as HBase, MongoDB, Cassandra, that support large amounts distributed parallel access, and are widely used in social media, e-commerce and other fields as datastores [5]. Some NoSQL database and RDB have been Hadoop-supported systems all along, conveniently the storage systems which work with Hadoop fit Spark [6]. The compatibility with these systems supported by Hadoop appears extremely
significant, for interoperation between Spark and external data sources (HDFS, HBase, Cassandra, Oracle, MySQL, etc.) is necessary in several scenarios for advanced computation such as machine learning and graph analysis. Alternatively, this facilitates switching from Hadoop to Spark for better performance improvement. Therefore, integration of Spark and external data sources, can play their respective functional advantages to meet needs of data analysis and processing of a variety of applications.

**Apache Spark**

Spark [6] is a distributed, in-memory, computing framework for massive data processing using a cluster of multi-nodes. The Spark cluster consists of a single driver node which is in charge of task scheduling and dispatching, and multiple worker nodes which are responsible for running codes and storing data. Spark architecture enables users to write computation application up to 100x faster in memory, or 10x faster on disk than Hadoop. It has become one of the most popular computing engines actually and attracted many users and contributors.

**Resilient Distributed Datasets (RDDs)**

RDD is the basic abstraction and core concept in Spark. It represents an immutable, partitioned collection of elements and can be distributed parallel computing. Data flow of Spark RDDs is illustrated in Fig.1. A rich set of operations (transformations, such as map, filter, and actions, such as count, collect) are provided by RDD to manipulate its partitions. Transformations used to define new RDDs are lazy, such operations only recorded by Spark will not be executed until one of the terminal action can trigger submission of a job.

RDDs are typically created from parallelizing a collection object, or from referencing data stored in a data source, or from transformations applied on existing RDDs. With enough information, RDDs know well about how they were derived from other RDDs and are able to rebuild the lost partitions. RDD/partitions allows to be explicitly materialized in memory or external storage for subsequent reuse to speed up processing.

RDD dependencies fall into two types: narrow dependency, where each partition of the parent RDD is consumed by no more than one partition of the child RDD, and shuffle dependency, where multiple partitions of the child RDD may depend on the parent RDD. Spark scheduler builds DAG of stages at shuffle dependencies and pipelines narrow operation inside each stage. Also, the dependencies between stages determine the scheduling order of Spark. The operations exerted on the RDDs produce the entire Spark program [7].

![Figure 1. This is data flow of the Spark application model. Operation of RDD objects forms DAG in the left dotted frame. RDDs are boxes with solid outlines and partitions are shaded rectangles in the right dotted frame, Grey means the partition has been cached in memory.](image)

**Exposing External Data Source as Spark RDDs**

Spark is not limited to HDFS as underlying persistent storage. Instead, it can create distributed datasets from a wide range of data sources, for Spark core provides these storage platforms
compatible with Hadoop with built-in support [4]. In particular, Spark can access data through the InputFormat and OutputFormat interfaces used by Hadoop, which are available for many common file formats and datastore systems (e.g., Local file system, HDFS, MySQL, Oracle, Cassandra, and HBase). With the aid of the interfaces, Spark can encapsulate all elements of a dataset as RDDs, allowing them to be further processed. See sections below for more details on InputFormat and OutputFormat interfaces.

Data Locality

Data locality is one of key factors that influence the performance of Spark application. Typically, it is more effective to ship application codes across nodes than large blocks of data for code volume is much less than data. In order to reduce the network transmission of the data, the calculation task should be distributed to the place where the data is as close as possible. [4] Normally to leverage data locality, Spark tries to read data into an RDD from the nodes that are close to it, its scheduling may obtain good data locality by preferred location of RDD. For example, the preferred location of HadoopRDD partition is the node where HDFS blocks are on; If RDD or any partitions has been cached, tasks should be allocated to the nodes that cached them to perform. Otherwise, Spark traces back the lineage until it find an ancestor RDD which has the preferred position, on which the task should be executed. Spark establishes scheduling system based on the fundamental rules of data locality.

Hadoop Interfaces: InputFormat and OutputFormat

Hadoop supports processing of many different formats and types of external memory systems through its InputFormat and OutputFormat interfaces. InputFormat [2], an interface of Hadoop MapReduce framework, implements two important functions: data splitting and Record Reading. The former performs the logic that how input will be distributed between the map processes. The default InputFormat is TextInputFormat, which generates a single split per block and gives the location of the block to the map task. Then, Hadoop will execute a mapper for each split. The interface enables developer override to read multiple data formats. The latter is mainly responsible for read the divided data in certain way (e.g., such as reading by line) and submits splits to mapper processes in the form of key/value. Virtually, each split is a logical division of the processed data. The division just records length (determined by start and offset) and location (including input path and host lists) of one split that are useful for data locality. OutputFormat [2] defines the data storage format, location and the organization of the output data. It gives control of how to write the record efficiently to optimize output.

There are some standard implementations of InputFormat already available for users to facilitate the development and utilization, such as TextInputFormat, ColumnFamilyInputFormat, TableInputFormat, DBInputFormat, the same as OutputFormat corresponding to InputFormat. Certainly, these implemented interfaces for Hadoop interacting with HDFS, Cassandra, HBase and RDB respectively can be compatibly used to work well with Spark.

Implementation

We choose Cassandra, MySQL and HDFS as representatives of NoSQL, RDB and distributed file system respectively, and integrate Spark with them separately, as is displayed in Fig.2. For the consideration of data locality, The Spark cluster is overlapped on HDFS nodes and Cassandra nodes since both of them support distributed parallel access. In our deployment scheme, the Spark scheduler tends to intelligently allocate tasks to the worker nodes where data resides to concurrently process. Before using a customized or existing implementation of InputFormat, it is indispensable to create a Hadoop job, as there is no direct interface available of access to data sources for Spark which is compatible with any data sources supported by Hadoop. Also, it is important to specify initial contact address, port of the data source and other optional connection properties for the integration. The newAPIHadoopRDD method of SparkContext, that represents
the connection of the application to a Spark cluster, can expose external data source as Spark RDDs via certain implementation of InputFormat. After processed, RDDs or results can be saved to a persistent storage using saveAsNewAPIHadoopFile method of RDD with specifying implementation of OutputFormat for consumption by other clients.

For constrained by the limits of single-server architecture, MySQL server is proper to be deployed on a single node inside or outside a Spark cluster. In any case, there will be data movement on network inevitably, because Spark executors that take on tasks need to draw data from the server for parallel computing remotely. No magic for this.

Figure 2. The integration organization of Spark and different data sources (e.g., Cassandra, HDFS, MySQL) in client mode. In Spark client mode, driver program runs on the client and initializes a SparkContext to generate related job information which are submitted to cluster manager.

Evaluation

This section focuses on the process and the efficiency of Spark accessing HDFS, MySQL, Cassandra with Hadoop’s TextInputFormat, DBInputFormat, ColumnFamily-InputFormat respectively.

Experimental Setup

The experiment was conducted on a configuration of 2.6 GHz Intel Core i5 processor and 8GB RAM server. We deployed Spark 1.5.2, Hadoop2.6.0, and Cassandra 2.1.9 in one cluster on three virtual nodes of the machine, and another virtual nodes is employed to run MySQL Server 5.6.27. Cluster architecture refers to Fig.2.

We randomly generate four tables of different sizes: Teachers with 300,000 rows, 600,000 rows, 3,000,000 rows and 5,000,000 rows respectively, each row consists of three attributes: name, wage and and department.

We executed same calculation logic repeatedly to compare the efficiency of Spark working with HDFS, MySQL and Cassandra in two ways. One is that Spark application saves records of tables above into the three storages. The other is that Spark exposes records stored in the three storages as RDDs and computes number of records and total wages of all teachers. HDFS stored our records as a collection of lines in text files, and commas separated field values of a row on a single line.

Figure 3. Execution time of Spark writing HDFS, MySQL and Cassandra.

Figure 4. Execution time of Spark reading HDFS, MySQL and Cassandra.
**Result Analysis**

Fig.3 and Fig.4 show the experimental results. Spark can access text files in HDFS in second response time as the number of records varies from 300,000 to 6,000,000. It is very fast as well as the local file system.

The latency of Spark pulling data from MySQL is almost equivalent to that of Spark pulling data from Cassandra when the amount of records is less than 60 as illustrated in Fig.4. With the increase of data-scale, the performance of Spark accessing MySQL decline sharply. In comparison, Cassandra approximates linear variation. These stem from its capability of distributed data-parallel processing.

[8] Cassandra spreads the partitioned data across multiple nodes using token rings. A data partition is made of one or more contiguous token ranges, each is fetched in batches processed in parallel to reduce the number of roundtrips to Cassandra. In order not to break data locality, we kept Spark nodes collocated with Cassandra nodes, queries are always sent to the Cassandra process running on the same node as the Spark executor process, thus decreasing data transfer between nodes. However, it’s complicated to partition or shard a RDB, since it was designed for a single node rather than clusters model [4]. when pulling records from a RDB, Spark distributes the tasks on workers which retrieves sub-data to execute concurrently. In this case, database connection will become a bottleneck due to volumes of data migration between multiple Spark nodes and a single RDB server. Maybe a shared-nothing architecture parallel databases can overcome the shortcomings. HadoopDB [9] is such a compounded infrastructure of MapReduce and PostgreSQL system. Each Hadoop sever in the system works on an individual instance of RDB. When receiving a SQL query, HadoopDB translates the request into a MapReduce job which is then distributed into a set of subtasks on multiple single-node databases and gain good effects.

On the whole, when the three executes the same processing logic, database systems including Cassandra and MySQL, work with Spark have a longer delay compared to HDFS, because database systems build on file system and invest more time in the creation and maintenance of the connection, the query parsing and optimization etc.

**Limitations of DBInputFormat and DBOutputFormat**

In our experiments, Spark generates certain SQL queries by optimizing the application and sends the requests to the database. Yet, any incremental updates to the database could affect consistency of results of the multiple identical queries from Spark, and the set of results may vary. This will be handled by doing an incremental select to make sure data reference stay up to date with production databases. Another option would be to dump the data to a temporary table in the database before launching the Spark application.

The DBOutputFormat saves data into a database by generating a set of INSERT statements then executes them in a bulk transaction. Performing large volumes of these from several tasks concurrently against a database may swamp its server or overload network connection. Optionally, there are other approaches that would work better for bulk data migration, for instance Apache Sqoop [10] is a powerful open source tool that imports and exports data between RDBs and HDFS. It is very fast to store records as a text file in HDFS, then you can enjoy this tool to import data into a database from HDFS.

**Conclusion**

InputFormat and OutputFormat interfaces enable Spark applications to interact well with a variety of external data sources supported by Hadoop. In addition, Spark can support extra data types, since the interfaces allow developers to custom InputFormat and OutputFormat according to specific data sources. Base on this, the paper puts forward a reference for Spark to access various data sources. Consequently, Spark can extend the local or distributed file system, RDB, NoSQL database etc., as its underlying persistent storage system.

Out of the design concept of high concurrent access for Cassandra against large-scale datasets, Cassandra has better support for Spark compared to MySQL, especially when the data volume is
very large. RDBs turn out to be inadequate in distributed scenarios involving massive number of servers. Performance and limitations of the integration are identified, we present the corresponding proposals for improvement. However, each type of integration of Spark and a data source also has its own range of issues that need to be disposed by future research.

Acknowledgement

This research is partially supported by National Key Fundamental Research, Development Program (No.2013CB329601) and National Natural Science Foundation of China (No.61502517, No.61572492).

References


