Resilient Distributed Datasets vs Spark SQL

Ottar Gudmundsson Xin Ren

I. Motivation

A. RDD

RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude.

1) Current frameworks lack abstractions for leveraging distributed memory which makes them inefficient for an important class of emerging applications that reuse intermediate results across multiple computations.
2) Data reuse is common in many iterative machine learning and graph algorithms, as well as interactive data mining, where a user runs multiple adhoc queries on the same subset of the data.
3) In most current frameworks, the only way to reuse data between computations is to write it to an external stable storage system, which can dominate application execution times.
4) Specialized frameworks developed for some applications that require data reuse only support specific computation patterns and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse.

B. Spark SQL

Spark SQL is highly motivated by industry requirements of big data applications. These requirements are associated with processing techniques, data sources and storage formats.

1) The earliest systems designed for these workloads gave users low-level, procedural programming interface. To achieve high performance, manual optimization was needed.
2) The relational approach of multiple new systems taking advantage of declarative queries to provide richer automatic optimizations is insufficient for many big data applications: users want to perform ETL to and from various data sources that might be semi-or unstructured, requiring custom code; users want to perform advanced analytics, such as machine learning and graph processing, that are challenging to express in relational systems.
3) It is observed that most data pipelines would ideally be expressed with a combination of both relational queries and complex procedural algorithms. Unfortunately, these two classes of systemsrelational and proceduralhave until now remained largely disjoint, forcing users to choose one paradigm or the other.

Therefore a new model that combines both procedural and relational models was required, catering to multiple needs.

II. Contributions

A. RDD

The resilient distributed datasets (RDDs) is an efficient, general-purpose and fault-tolerant abstraction for sharing data in cluster applications.

1) RDDs can express a wide range of parallel applications, including many specialized programming models that have been proposed for iterative computation, and new applications that these models do not capture.
2) Unlike existing storage abstractions for clusters, which require data replication for fault tolerance, RDDs offer an API based on coarse-grained transformations that lets them recover data efficiently using lineage.
3) RDDs is implemented in a system called Spark that outperforms Hadoop by up to 20 in iterative applications and can be used interactively to query hundreds of gigabytes of data.

B. Spark SQL

Spark SQL is a new module in Apache Spark that integrates relational processing with Sparks functional programming API. It lets Spark programmers leverage the benefits of relational processing, and lets SQL users call complex analytics libraries in Spark. It's an evolution of both SQL-on-Spark and of Spark itself, offering richer APIs and optimizations while keeping the benefits of the Spark programming model. This is achieved by two major components:

1) The DataFrame API, a tight integration between relational and procedural processing that allows operations on multiple data sources, both external and in built. It evaluates operations lazily so that it can perform relational optimizations.
2) Catalyst, a highly extensible query optimizer that allows simple methods to add new data sources, optimization rules, and data types for domains such as machine learning.

III. Solution

A. RDD

An RDD is a read-only, partitioned collection of records that can only be created through transformations on either data in stable storage or other RDDs. Each RDD is represented through a common interface that exposes following information: a set of partitions; a set of dependencies on parent RDDs;

a function for computing the dataset based on its parents; and metadata about its partitioning scheme and data placement. Dependencies between RDDs are classified into two types: narrow dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD, wide dependencies, where multiple child partitions may depend on it. Narrow dependencies allow for pipelined execution on one cluster node and efficient recovery after a node failure.

Spark provides the RDD abstraction through a language integrated API in Scala. To use Spark, Users write a driver program that connects to a cluster of workers. The driver defines one or more RDDs (transformations) and invokes actions on them. Transformations are lazy operations, while actions launch a computation to return a value to the program or write data to external storage. Spark code on the driver also tracks the RDDs lineage. The workers are long-lived processes that can store RDD partitions in RAM across operations. Users can call a persist method to indicate which RDDs they want to reuse in future operations and are in control of partitioning of RDDs.

Spark runs over the Mesos cluster manager, allowing it to share resources with Hadoop, MPI and other applications. Sparks scheduler uses the representation of RDDs. Whenever a user runs an action on an RDD, the scheduler examines that RDDs lineage graph to build a DAG of stages to execute. Each stage contains as many pipelined transformations with narrow dependencies as possible. The boundaries of the stages are the shuffle operations required for wide dependencies, or any already computed partitions that can short-circuit the computation of a parent RDD. The scheduler then launches tasks to compute missing partitions from each stage until it has computed the target RDD. The scheduler assigns tasks to machines based on data locality using delay scheduling. For wide dependencies, currently intermediate records are materialized on the nodes holding parent partitions to simplify fault recovery. If a task fails, it's re-run on another node as long as its stages parents are still available. If some stages have become unavailable, tasks are resubmitted to compute the missing partitions in parallel.

To let users run Spark interactively from the interpreter to query big datasets, two changes are made to the interpreter in Spark: the interpreter is made to serve these classes over HTTP and the code generation logic is modified to reference the instance of each line object directly.

Spark provides three options for storage of persistent RDDs: in-memory storage as deserialized Java objects, in-memory storage as serialized data, and on-disk storage. Spark keeps persistent RDDs in memory by default. When a new RDD partition is computed but there is not enough space to store it, a partition from the least recently accessed RDD is evicted, unless it is the same RDD as the one with the new partition. In that case, the old partition is kept in memory to prevent cycling partitions from the same RDD in and out. Users can also request other persistence strategies, such as storing the RDD only on disk or replicating it across machines or setting a persistence priority on each RDD to specify which in-memory data should spill to disk first.

RDDs provide fault tolerance with its lineage and checkpointing is used to shorten the recovery time for RDDs with long lineage chains especially those containing wide dependencies. If a node fails, lost partitions from these RDDs can be recomputed in parallel on other nodes, at a fraction of the cost of replicating the whole RDD.

B. Spark SQL

Spark SQL runs as a library on top of Spark. It exposes SQL interfaces, which can be accessed through JDBC/ODBC or through a command-line console, as well as the DataFrame API integrated into Sparks supported programming languages.

The main abstraction in Spark SQLs API is a DataFrame, which is a collection of distributed rows with homogeneous schema that can be constructed from tables in a system catalog (based on external data sources) or from existing RDDs of native Java/Python objects. Once constructed, they can be manipulated with all common relational operators, for example, select, join and where. These operations all take expression objects in a limited DSL that lets Spark capture the structure of the expression and construct an AST (abstract syntax tree) of the expression which is later optimized by the Catalyst. DataFrames can also be registered as temporary tables in the system catalog and queried using SQL. The Dataframe presents a logical plan to compute a dataset and will only become physical when any output is required.This enables rich optimization across all operations that were used to build the DataFrame.

Spark SQL can automatically infer the its schema using reflection, when DataFrame is constructed from native RDD. it can also cache data in memory using columnar cache which applies columnar compression schemes such as dictionary encoding and run-length encoding. UDFs are supported as well, which can be registered inline by passing Scala, Java or Python functions. UDFs can operate on scalar values, as well as an entire table by taking its name, so that one can use the distributed Spark API within them, in which way they help to expose advanced analytics functions to SQL users.

To implement Spark SQL, a new extensible optimizer, Catalyst, is designed based on functional programming constructs in Scala. Catalyst uses standard features of the Scala programming language, such as pattern-matching, to let developers use the full programming language while still making rules easy to specify. The main data type in Catalyst is a tree composed of node objects. Trees offer a transform method defined by rules that applies a pattern matching function recursively on all nodes of the tree, transforming the ones that match each pattern to a result. Catalyst groups rules into batches, and executes each batch until it reaches a fixed point. Trees are immutable, allowing parallel optimization on the tree.

Catalysts general tree transformation framework is used in four phases:

1) Logical Plan Analysis: Resolves attribute references that have unknown type or no match to the input. Tracks all tables in all data sources to resolve references.

2) Logical Optimization: Apply standard rule-based optimizations to the logical plan such as boolean expressions simplification and constant folding.
3) Physical Planning: One or more physical plans are generated from a logical plan, using physical operators that match the Spark execution engine, and the one with the lowest cost will be selected.
4) Code Generation: Scala provides quassiquotes to construct an AST that is fed to the Scala compiler at runtime to generate bytecode.

Catalysts design around composable rules makes it easy for users and third-party libraries to extend. Developers can add batches of rules to each phase of query optimization at runtime, as long as they adhere to the contract of each phase. To make it even simpler to add some types of extensions without understanding Catalyst rules, two narrower public extension points are defined: data sources(new ones can be defined using several APIs) and user-defined types (mapped to structures composed of Catalysts built-in types).

To handle challenges in "big data" environments, three features were added to Spark SQL:

1) A schema inference algorithm for JSON and other semistructured data: The algorithm attempts to infer a tree of STRUCT types, each of which may contain atoms, arrays, or other STRUCTs. For each field defined by a distinct path from the root JSON object, the algorithm finds the most specific Spark SQL data type that matches observed instances of the field. It's implemented using a single reduce operation over the data, which starts with schemata from each individual record and merges them using an associative most specific supertype" function that generalizes the types of each field.
2) Integration with Sparks Machine Learning Library: MLlib, Sparks machine learning library, introduced a new high-level API based on the concept of machine learning pipelines. A pipeline is a graph of transformations on data, each of which exchange datasets. To represent datasets, the new API uses DataFrames, where each column represents a feature of the data. All algorithms that can be called in pipelines take a name for the input column(s) and output column(s), and can thus be called on any subset of the fields and produce new ones. MLlib creates a user-defined type for vectors, which stores both sparse and dense vectors, and represents them as four primitive fields: a boolean for the type, a size for the vector, an array of indices, and an array of double values.
3) Query federation allowing a single program to efficiently query disparate sources: Spark SQL data sources leverage Catalyst to push predicates down into the data sources whenever possible.

IV. Your Opinions

1) In RDD, transformations are lazy operations, while actions launch a computation to generate output, similarly in Spark SQL, the Dataframe presents a logical plan to compute a dataset and will only become physical when any output is required.
2) Both RDD and Tree in Spark SQL are immutable.
3) A DataFrame is equivalent to a table in a relational database, and can also be manipulated in similar ways to the RDDs, but Unlike RDDs, DataFrames keep track of their schema and support various relational operations that lead to more optimized execution.
4) Both RDD and DataFrame can be cached in memory, RDD as deserialized Java objects or serialized data, while DataFrame applies columnar compression schemes such as dictionary encoding and run-length encoding, which reduce memory footprint by an order of magnitude.
5) DataFrame can be constructed from RDD.
6) In both paper, only runtime is benchmarked, other metrics such as memory/CPU usage is not displayed which might indicate other interesting trade offs in performance.
7) Both papers list references which is good, but can be better if there are hyperlinks.
8) It is good that in RDD paper, the motivation is explicitly stated; Spark SQL paper can be improved by doing this.
9) It is good that in Spark SQL paper, the contribution is explicitly stated; RDD paper can be improved by doing this.
10) In both papers, the structure of the paper is described in the end of the introduction.
11) Both paper point out some drawbacks of old systems and compare with related works.
12) Both paper use examples to demonstrate ideas, which makes it easier to understand.