Part II: Data Center Software Architecture:

Topic 3: Programming Models

Shark: Fast Data Analysis Using Coarse-grained Distributed Models

Shark: SQL and Rich Analytics at Scale

Presented by Ye Li
Outline

Introduction
  • Background
  • Related Works
  • Motivations
  • Contributions

Methodology
  • Architecture
  • Engine Extensions
  • Machine Learning Support
  • Implementation
  • Discussion

Conclusions
References
Shark is a system that leverages a modern MapReduce engine and techniques from databases, which supports both SQL and complex analytics efficiently, while maintaining fault-tolerance.

Characteristics:
1. MapReduce engine - using Spark
2. SQL query - interactive data ETL
3. complex analytics - machine learning supports
4. fault-tolerance - fine-grained mid-query recovery
Spark

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split(\t')(1))
messages.cache()
messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
```

Shark:

```sql
CREATE TABLE log(header string, message string) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LOCATION "hdfs://...";
CREATE TABLE errors_cached AS SELECT message FROM log WHERE header = "ERROR";
SELECT count(*) FROM errors_cached WHERE message LIKE "%foo%";
SELECT count(*) FROM errors_cached WHERE message LIKE "%bar%";
```
Background

Why combining MapReduce engines and features from databases

MapReduce’s pros:
1. a fine-grained fault tolerance model for tasks in large clusters;
2. being able to express large-scale statistical and learning algorithms;
3. supporting unstructured data and “schema-on-read”.

MapReduce’s cons:
1. high latencies;
2. coarser-grained recovery for (complex) SQL queries.
Background

Why combining MapReduce engines and features from databases?

Databases’ pros:
1. Data representations:
   A. schema-aware, column-oriented;
   B. co-partition & co-location of data.
2. Execution strategies:
   A. low scheduling/task launching overhead
   B. cost-based optimization
   C. indexing

Databases’ cons:
1. lack of mid-query fault tolerance
Background

```
SELECT page_name, SUM(page_views) as views FROM wikistats
GROUP BY page_name ORDER BY views DESC LIMIT 10;
```

```
Stage 1: Map-Shuffle
Mapper(row) {
    emit(page_views, page_name);
}

... shuffle sorts the data

Stage 2: Local
data = open("stage1.out")
for (i in 0 to 10) {
    print(data.getNxt());
}
```

Page 0: Map-Shuffle-Reduce

```
Group By

Reducer(key, values) {
    SUM page_views = 0;
    for (page_views in values) {
        sum += value;
    }
    emit(key, page_views);
}
```
Large-scale data analytics systems: three classes

System that compiles declarative queries into MapReduce-style jobs:

- ASTREIX, Tenzing, SCOPE, Cheetah and Hive
- execution engine to transform SQL queries into MapReduce jobs
- high latencies, hard to achieve interactive query response times

Low-latency engines using architectures resembling shared-nothing parallel databases:

- PowerDrill, Cloudera Impala and Google Dremel;
- have to re-execute the entire query for mid-query failures;
- do not support complex shuffle DAGs required for large joins or distributed machine learning.
Large-scale data analytics systems: three classes

Hybrid approach by combining a MapReduce-like engine with relational databases:

- HadoopDB, Ospery;
- connects multiple single-node database systems using Hadoop communication layer;
- breaking SQL query into multiple small queries.
- complicate system considering fault tolerance

Shark can be categorized into the first class of systems while it supports low latency query (second class) and all of the properties of the third class of systems.
Motivations & Contributions

Motivations

- combine SQL and complex analytics in large-scale datasets
- high latency in MR-style system (Hive)
- coarse-grained fault tolerance in databases system
- lack the rich analytics functions in databases system

Contributions

- SQL on Spark
- built on Apache Spark to avoid disk I/O and support fine-grained fault tolerance
- in-memory columnar storage and columnar compression
- Partial DAG Execution (PDE) in optimizer
Outline

Introduction
  • Background
  • Related Works
  • Motivations
  • Contributions

Methodology
  • Architecture
  • Engine Extensions
  • Machine Learning Support
  • Implementation
  • Discussion

Conclusions

References
Apache Hive
1. Metastore
2. Driver
3. Compiler
4. Execution Engine
Architecture - Hive

Metastore:
1. stores the system catalog and metadata about tables, columns, partitions etc;
2. this information should be served fast to the compiler for type checking, semantic analysis and query optimization;
3. stored in a traditional RDBMS for low latency;
4. replicated by a backup RDBMS server

Tables - A table is stored in a directory in hdfs;

Partitions - A partition of the table is stored in a subdirectory within the table’s directory

Buckets - A bucket is stored in a file within the partition’s or table’s directory depending on whether the table is a partitioned table or not
Query Compiler:

1. SQL Parser - uses Antlr to generate the abstract syntax tree (AST) for the query;

2. Type checking and Semantic Analysis - fetches information (input and output tables) from Metastore to build a logical plan (AST -> DAGs -> MR);

3. Optimization - consists of a chain of transformations such that the operator DAG resulting from one transformation is passed as input to the next transformation.

Optimizations:

- Column pruning
- Predicate pushdown
- Partition pruning
- Map side joins
- Join reordering
Architecture - Hive

Execution engine:
1. tasks are executed in the order of their dependencies
2. each deponent task is only executed if all of its prerequisites have been executed
Architecture - Shark

Spark
Modified Compiler
Cache Manager

Hive Architecture

Meta store

Client
CLI
JDBC

Driver
Cache Mgmt

SQL Parser
Query Optimizer
Physical Plan
Execution

Spark

Hadoop Storage (HDFS, S3, ...)

Partial DAG Execution (PDE)

Lack of statistics for fresh data and the prevalent use of UDFs necessitate dynamic approaches to query optimization; PDE allows dynamic alternation of query plans based on statistics collected at run-time.
Partial DAG Execution (PDE)

Gather customizable statistics at per-partition granularities while materializing map output

- partition sizes, record counts (skew detection)
- “heavy hitters”
- approximate histograms

Can alter query plan based on such statistics

- map join vs shuffle join
- symmetric vs non-symmetric hash join
- skew handling

data is sent to the master in compression to minimize the size
**Engine Extensions**

**Columnar Memory Store**
- Inefficient caching strategies for JVM objects in Hive; column-oriented storage.

<table>
<thead>
<tr>
<th>Row Storage</th>
<th>Column Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 john 4.1</td>
<td>1 john 4.1</td>
</tr>
<tr>
<td>2 mike 3.5</td>
<td>2 mike 3.5</td>
</tr>
<tr>
<td>3 sally 6.4</td>
<td>3 sally 6.4</td>
</tr>
</tbody>
</table>

**Benefits**
- Compact representation
- CPU efficient compression
- Cache locality
Engine Extensions

Distributed Data Loading
data loaded into memory as columnar representations in parallel by each Spark tasks;
individual compression strategies within each data loading task requires trivial additional cost to maintain compression metadata.

Data Co-partitioning
two tables are frequently joined together (TPC-H benchmark)
HDFS is schema-agnostic; Spark allows co-partitioning two tables on a common key;
Shark’s optimizer uses map tasks to perform the join rather than an expensive shuffle.

CREATE TABLE l_mem TBLPROPERTIES ("shark.cache"=true) AS SELECT *
FROM lineitem DISTRIBUTE BY L_ORDERKEY;

CREATE TABLE o_mem TBLPROPERTIES ("shark.cache"=true,
Engine Extensions

Partition Statistics and Map Pruning

partition statistics - information collected and sent to master program including:

- the range of each column
- enum columns
- the number of distinct values

map pruning - when a query is issued, Shark evaluates the query predicates against all partition statistics
Machine Learning Support

Shark supports machine learning as a first-class citizen. It is possible to express certain machine learning algorithms in SQL. Language Integration enables a unified system for SQL, machine learning, and streaming. Both share the same set of workers and caches.

```scala
// Example code snippet

def logRegress(points: RDD[Point]): Vector {
  var w = Vector(D, _ => 2 * rand.nextDouble - 1)
  for (i <- 1 to ITERATIONS) {
    val gradient = points.map { p =>
      val denom = 1 + exp(-p.y * (w dot p.x))
      (1 / denom - 1) * p.y * p.x
    }.reduce(_ + _)
    w -= gradient
  }
  w
}

val users = sql2rdd("SELECT * FROM user u JOIN comment c ON c.uid=u.uid")

val features = users.mapRows { row =>
  new Vector(extractFeature1(row.getInt("age")),
              extractFeature2(row.getString("country")),
...}

val trainedVector = logRegress(features.cache())
```

Execution Engine Integration
Implementation

Memory-based Shuffle

Both Spark and Hadoop write map output files to disk. Shark modified the shuffle phase to materialize map outputs in memory.

Temporary Object Creation

Burdens on the JVM’s garbage collector will slow tasks and jobs. Shark operators and RDD transformation minimize temporary object creations.

Bytecode Compilation of Expression Evaluators

Specialized Data Structures
**Discussion**

Why are Previous MapReduce-Based Systems Slow?

1. expensive data materialisation for fault tolerance
2. inferior data layout (lack of indices)
3. costlier execution strategies

**Intermediate Outputs**

- MR saves outputs of map tasks in case reduce tasks fail;
- HDFS stores output of each step on disk in multiple MR steps

**Data Format and Layout**

- “schema-on-read” in MapReduce
- Hive supports “table partitions”
- columnar memory storage in Shark
- Shark - co-partition (derives from Spark)
- to exploit: random read
Discussion

Why are Previous MapReduce-Based Systems Slow?

Execution Strategies

- sorting data before each shuffle and writing the outputs of each MR stage to
  in Hive - limitations of the rigid, one-pass MR
- Spark supports hash-based distributed aggregation and general task DAGs
- partial DAG execution select execution plans based on run-time data statistics
  - collecting fine-grained statistics about range of keys;
  - allowing switches to a completely different join strategy (rather than just
    selecting the number of reduce tasks in DryadLINQ)

Task Scheduling Cost

- the overhead of launching tasks
- Hadoop: 5-10 secs delays (3 secs heartbeat)
- Spark: thousands of tasks per sec (RPC)
- Hive’s performance is sensitive to the number of tasks
Discussion

Other Benefits of the Fine-Grained Task Model

- Elasticity
- Multitenancy
Outline

Introduction
  • Background
  • Related Works
  • Motivations
  • Contributions

Methodology
  • Architecture
  • Engine Extensions
  • Machine Learning Support
  • Implementation
  • Discussion

Conclusions

References
Conclusions

Leveraging a modern MapReduce engine and techniques from data
analysis, Shark supports both SQL and complex analytics efficiently, while
maintaining fault-tolerance.
Outline

Introduction
  • Background
  • Related Works
  • Motivations
  • Contributions

Methodology
  • Architecture
  • Engine Extensions
  • Machine Learning Support
  • Implementation
  • Discussion

Conclusions

References


