Spark SQL

Ильнур Шугаепов

Spark

RDDs

Definition 1 (RDD):

- *Resilient — отказоустойчивый*
- *Distributed — разбитый на партиции*
- *Dataset*

read-only, partitioned collection of records

RDDs can be manipulated through operations like `map`, `filter`, and `reduce`, which take functions and ship them to nodes on the cluster.

Spark

Fault-tolerance

- Запомним граф вычислений (lineage)
- Тогда если часть данных будет потеряна, то их легко можно восстановить

Spark

Lazy evaluation

- Each RDD represents a "logical plan" to compute a dataset
- Spark waits until action to launch a computation
- Allows to do some simple query optimization, such as pipelining operations (narrow dependencies)

Table of Contents

История

2006 MapReduce
2009 Hive
2009 Pig
2010 Spark
2013 Shark[1]
2015 Impala[2]
2015 Spark SQL[3]

[1] Reynold S Xin et al. “Shark: SQL and rich analytics at scale”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. 2013, pp. 13–24.

[2] Marcel Kornacker et al. “Impala: A Modern, Open-Source SQL Engine for Hadoop.”. In: *Cidr*. Vol. 1. 2015, p. 9.

[3] Michael Armbrust et al. “Spark sql: Relational data processing in spark”. In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 1383–1394.

Spark

Limitations

- Low-level procedural code
- No optimizations

Spark engine does not understand the structure of the data in RDDs (which is arbitrary Java/Python objects) or the semantics of user functions (which contain arbitrary code)

Наблюдение

Most data pipelines are combination of *relational* and *procedural* algorithms.

Цели

1. Support relational processing both within Spark programs (on native RDDs) and on external data sources using a programmer-friendly API.
2. Easily support new data sources, including semi-structured data and external databases amenable to query federation.
3. Enable extension with advanced analytics algorithms such as graph processing and machine learning.

Spark SQL

Main components

Spark SQL — new module
in Apache Spark

DataFrame

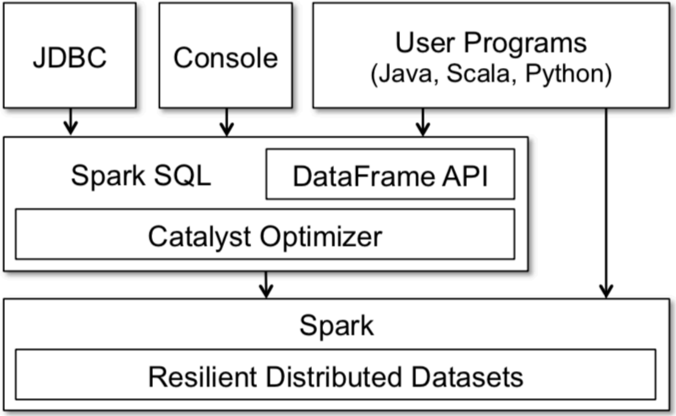
API

Catalyst

Optimizer

Table of Contents

Архитектура



DataFrame API

Definition 2 (DataFrame): *Collection of structured (with schema) records that can be manipulated using Spark's procedural API, or using new relational APIs that allow richer optimizations*

DataFrame

Construction

- From external data sources (HDFS, Hive)
- From existing RDDs (schema inference algorithm)

DataFrame can be viewd as an RDD of *Row* objects, allowing user to call procedural Spark APIs such as `map`

DataFrame

Execution

- Spark DataFrame are *lazy*
- Spark build *logical plan* before execution
- Laziness enables rich optimization
- Logical plan $\rightarrow$ Physical plan

DSL

Users can perform relational operations on DataFrames using a domain-specific language (DSL) similar to Python Pandas

Example

Фильмы с наибольшим средним рейтингом

```
ratings_df \
    .groupby('movie_id') \
    .agg(F.mean('rating').alias('mean_rating'),
         F.count('rating').alias('ratings_count')) \
    .join(movies_df, ratings_df['movie_id'] == movies_df['movieId'],
          how='inner') \
    .sort(F.col('mean_rating').desc()) \
```

Difference with native Spark API

- All of these operators build up an abstract syntax tree (AST) of the expression, which is then passed to Catalyst for optimization.
- This is unlike the native Spark API that takes functions containing arbitrary Scala/Java/Python code, which are then opaque to the runtime engine.

DataFrame construction

Schema inference

- While building DataFrame from RDD user can manually define schema
- Spark SQL can automatically infer the schema of the dataset using reflection
- In Python, Spark SQL samples the dataset to perform schema inference due to the dynamic type system

.cache()

- Method `cache` of DataFrame does the same thing as method `persist` of RDD
- Caching is particularly useful for interactive queries and for the iterative algorithms common in machine learning

UDF

Example

```
def get_release_year(title):
    result = re.match(r'.*(\(\d+\))', title)
    return int(result.group(1)[1:-1]) if result is not None else None

get_release_year_udf = F.udf(get_release_year, IntegerType())

movies_df \
    .withColumn('year', get_release_year_udf('title')) \
```

Резюме

- The DataFrame API lets developers seamlessly mix procedural and relational methods.

Table of Contents

Catalyst contains a general library for representing trees and applying rules to manipulate them

Trees

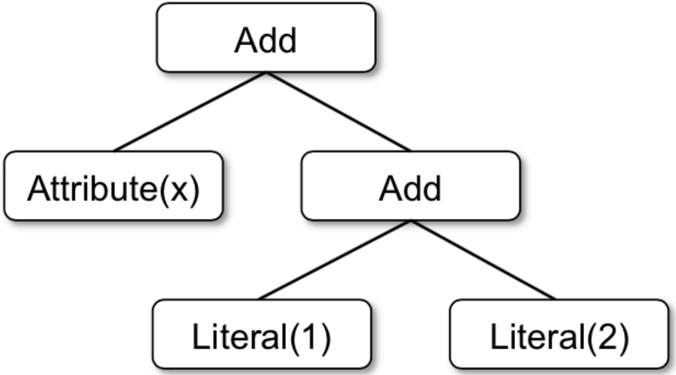


Figure: Catalyst tree for expression `x+(1+2)`

Rules

$Rule\colon\ T \mapsto T'$ — rule maps tree to another tree.

The most common approach is to use a set of pattern matching functions that find and replace subtrees with a specific structure.

Rules

Example

Constant folding

```
tree.transform {
    case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
    case Add(left, Literal(0)) => left
    case Add(Literal(0), right) => right
}
```

Rules

Fixed point

Catalyst groups rules into batches, and executes each batch until it reaches a *fixed point*, that is, until the tree stops changing after applying its rules.

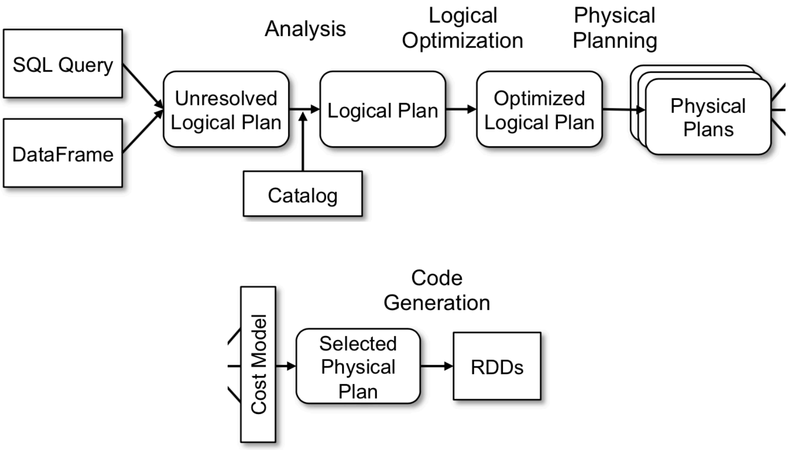


Figure: Phases of query planning in Spark SQL. Rounded rectangles represent Catalyst trees

Example

Query

```
query = """
    SELECT movie_id, COUNT(*), first(title) as title
    FROM ratings INNER JOIN movies ON ratings.movie_id == movies.movieId
    WHERE movies.title LIKE '%(1994)%'
    GROUP BY movie_id
    ORDER BY COUNT(*) DESC
"""

spark.sql(query).explain(True)
```

Example

Unresolved Logical Plan

```
== Parsed Logical Plan ==
'Sort ['COUNT(1) DESC NULLS LAST], true
+- 'Aggregate ['movie_id], ['movie_id, unresolvedalias('COUNT(1), None), first('title, false) AS title#53]
   +- 'Filter 'movies.title LIKE %(1994)%
      +- 'Join Inner, ('ratings.movie_id = 'movies.movieId)
         :- 'UnresolvedRelation `ratings`
         +- 'UnresolvedRelation `movies`
```

Example

Logical Plan

```
== Analyzed Logical Plan ==
movie_id: int, count(1): bigint, title: string
Project [movie_id#1, count(1)#56L, title#53]
+- Sort [count(1)#56L DESC NULLS LAST], true
   +- Aggregate [movie_id#1], [movie_id#1, count(1) AS count(1)#56L, first(title#19,
        false) AS title#53]
      +- Filter title#19 LIKE %(1994)%
         +- Join Inner, (movie_id#1 = movieId#18)
            :- SubqueryAlias `ratings`
            :  +- Relation[user_id#0,movie_id#1,rating#2,timestamp#3] csv
            +- SubqueryAlias `movies`
               +- Relation[movieId#18,title#19,genres#20] csv
```

Example

Optimized Logical Plan

```
== Optimized Logical Plan ==
Sort [count(1)#56L DESC NULLS LAST], true
+- Aggregate [movie_id#1], [movie_id#1, count(1) AS count(1)#56L, first(title#19, false)
      AS title#53]
   +- Project [movie_id#1, title#19]
      +- Join Inner, (movie_id#1 = movieId#18)
         :- Project [movie_id#1]
         :  +- Filter isnotnull(movie_id#1)
         :     +- Relation[user_id#0,movie_id#1,rating#2,timestamp#3] csv
         +- Project [movieId#18, title#19]
            +- Filter ((isnotnull(title#19) && Contains(title#19, (1994))) && isnotnull(
                  movieId#18))
               +- Relation[movieId#18,title#19,genres#20] csv
```

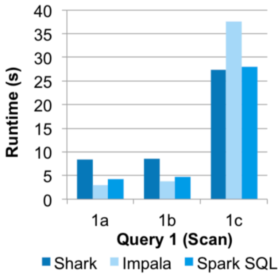
! Filter push down rule !

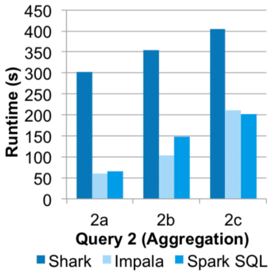
Table of Contents

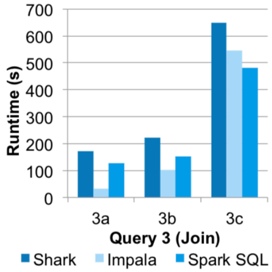
Evaluation of the performance of Spark SQL on two dimensions:

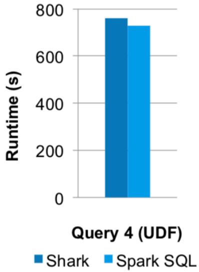
1. SQL query processing performance
2. Spark program performance

Benchmark[4]









[4] Andrew Pavlo. "A comparison of approaches to large-scale data analysis". In: *Proceedings of the 2009 ACM SIGMOD international conference on management of data*. 2009, pp. 165–178.

Distributed Aggregation

Dataset and Task

Dataset 1 billion integer pairs, (a, b) with 100000 distinct values of a

Task compute the average of b for each value of a

Distributed Aggregation

Solutions

Native Spark

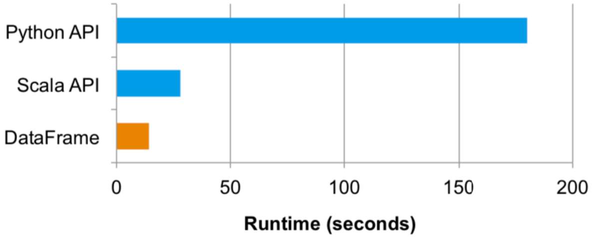
```
sum_and_count = data \
    .map(lambda x: (x.a, (x.b, 1))) \
    .reduceByKey(lambda x, y: (x[0]+y[0], x[1]+y[1])) \
    .collect()

[(x[0], x[1][0] / x[1][1]) for x in sum_and_count]
```

Spark SQL

```
df.groupBy("a").avg("b")
```

Distributed Aggregation

Performance



In the DataFrame API, only the logical plan is constructed in Python, all physical execution is compiled into native Spark code as JVM bytecode.

Armbrust, Michael et al. “Spark sql: Relational data processing in spark”. In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 1383–1394.

Karau, Holden and Rachel Warren. *High performance Spark: best practices for scaling and optimizing Apache Spark*. " O’Reilly Media, Inc.", 2017.

Kornacker, Marcel et al. “Impala: A Modern, Open-Source SQL Engine for Hadoop.”. In: *Cidr*. Vol. 1. 2015, p. 9.

Pavlo, Andrew. “A comparison of approaches to large-scale data analysis”. In: *Proceedings of the 2009 ACM SIGMOD international conference on management of data*. 2009, pp. 165–178.

Xin, Reynold S et al. “Shark: SQL and rich analytics at scale”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. 2013, pp. 13–24.

ml_bd

Вопросы?

Ильнур Шугаепов