Skalering til Big Data

04-05-2018

Christian Thomsen
chr@cs.aau.dk

Agenda

- Last time
- Background for Spark's DataFrames
- Using DataFrames
- How Spark does

Last time

- **MapReduce:** to scale out, brute force, low-level
- **Hadoop:** an open-source version of MapReduce (+more)
- **HDFS:** Hadoop's distributed file system, files stored in replicated blocks, append only (no updates)
- **HBase:** a distributed storage system, rows and (varying) columns, updates possible, low latency compared to HDFS
- **Hive:** SQL on MapReduce
- **Pig:** sequences of transformations, high-level, ad-hoc analysis, interactive use possible
- **Spark**

Spark

- Proposed in 2012 by researchers from UC Berkeley
- Now the most active Apache project
- Easy interactive use with REPL (read, eval, print, loop)
- Offers a rich set of APIs (Scala, Java, and Python)
- Spark can keep datasets in memory between jobs
  - Avoids the expensive I/O that MapReduce has to do
  - In particular good for iterative algorithms and interactive analysis
- Spark **does not use** MapReduce, but is closely integrated with Hadoop
  - Can run on YARN (can also run in other modes, e.g., stand-alone)
  - Can work with HDFS (and AWS S3, Azure Storage, …);
    does not have its own DFS/storage

Spark architecture

- A cluster manager (e.g., YARN) grants resources
- The ***driver process*** maintains information about the application, responds to the user’s code, and schedules work
  - In cluster mode, it runs on a cluster node (and an application is submitted to it)
  - In client mode, it runs on a client machine outside the cluster
- The ***executors*** execute code assigned by the driver and report back to the driver
  - Run on the cluster nodes
  - (but in local mode, everything – incl. the executors – run on a single machine)

A Spark application

- Multiple Spark applications can run on the cluster
- A Spark application has exactly one SparkSession which controls the computations
- On appserver2: **`/opt/spark/bin/pyspark`**

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.3.0
      /_/

Using Python version 2.7.6 (default, Nov 23 2017 15:49:48)
SparkSession available as 'spark'.
>>> spark
<pyspark.sql.session.SparkSession object at 0x7f2518e91550>
```

Resilient Distributed Datasets

- Spark uses *Resilient Distributed Datasets (RDDs)* which are **immutable** partitioned record collections
- An RDD can be created from
  - a collection of objects (the collection is *"parallelized"*)
  - data in stable storage (HDFS files etc.)
  - coarse-grained transformations on existing RDD(s) (e.g., map, filter, and join)
- RDDs can be recomputed as needed or kept in memory
- We saw examples last time
- You can still operate on RDDs – but they are low-level
- The new ***structured APIs*** understand the data format and user code and exploit that to make optimizations
- We will look at them today

Spark SQL

- Spark SQL added in 2014
- Described in M. Armbrust et al. ”Spark SQL: Relational Data Processing in Spark”, SIGMOD, 2015
- Added the structured DataFrame API with declarative, relational operations
- Added the optimizer *Catalyst*

- Great performance gains, flexibility, and cleaner code

Example

- For a set of integer pairs *(a,b)*, compute for each unique value of *a* the average of the *b* values
- ”Old” Spark where we use RDDs directly:

```
sum_cnt = \
  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_cnt]
```

- With DataFrames:

```
df.groupBy(”a”).avg(”b”)
```

[Example taken from M. Armbrust et al.]

Expressions building an AST

- The user performs operations by means of expressions
- An expression can be a string which is parsed by Spark or can be formed by calling a function – **`groupBy, col`**, …
- Operators can form new expressions
  - +, -, /, <, >, ==, …
- This builds an abstract syntax tree (AST) where Spark understands what the user wants to do
- The result is computed in the Spark engine (and not in your Python code)
- Compare to the previous example where Spark has no clue about what the Python lambda expression does

Datasets and DataFrames

- A Dataset<T> is a parameterized, distributed data collection
  - Type-safety: A Dataset<T> can only hold instances of T and there is no need for you to cast back to T.
  - Problems detected at compile time
  - Only available in Scala and Java
- A DataFrame is internally a Dataset<Row>
  - Row is Spark's optimized format with data of Spark's own types
  - Types checked at runtime

- DataFrames are generally the most efficient and easiest abstraction to work with in Spark

DataFrames

- A DataFrame is collection of rows with a fixed number of named and typed columns
  - A DataFrame has (row-wise) partitions distributed across nodes
  - Immutable
- Transformations tell Spark how to transform one DataFrame into another
- Transformations are lazy to allow Spark to optimize
- Actions trigger computations

Agenda

- Last time
- Background for Spark's DataFrames
- **Using DataFrames**
- How Spark does

Example

```
>>> x = spark.range(5).toDF("NewName")

>>> x.take(2)
[Row(NewName=0), Row(NewName=1)]

>>> x.select(x["NewName"] * 2)
DataFrame[(NewName * 2): bigint]

>>> x.select(x["NewName"] * 2).collect()
[Row((NewName * 2)=0), Row((NewName * 2)=2),
Row((NewName * 2)=4), Row((NewName * 2)=6),
Row((NewName * 2)=8)]
```

Working with DataFrames

- We can get the content of a DataFrame df:
  - `df.first()    # returns first row`
  - `df.take(N)    # returns N rows`
  - `df.collect() # returns all rows (maybe MANY!!!)`

- A number of ways to make a new DataFrame from an existing (recall they are immutable)
- To rename all columns: **`df.toDF(”new1”, ”new2”,…)`**

- Expressions to select/remove and manipulate columns
- To refer to a column:

```
from pyspark.sql.functions import col
```

`col(”name”)` or

```
df[”name”] # but not df.col(”name”) or df(”name”)
```

Expressions

- An expression transforms values
- We use **`expr`**:
  `from pyspark.sql.functions import expr`
- **`expr`** can parse transformations and column references: **`expr(”col - 5”)`** or **`expr(”col”) - 5`**.
- **`col`** is just a (simple) expression**:** **`col(”x”)`** is the same as **`expr(”x”)`**

- **`expr(”col”) - 5`** ✓
- **`expr(”col - 5”)`** ✓
- **`col(”col”) - 5`** ✓
- ***`col(”col” - 5)`*** ✗

Selection – projection

- **`select`** similar to SELECT in SQL
- **`df.select(”col1”, ”col2”)`**

- **`df.select(col(”col1”) - 5)`** works,
  `df.select(”col - 5”)` fails,
  `df.select(”col” - 5)` fails,
  `df.select(expr(”col1 - 5”))` works
  `df.select(df.col1 - 5)` works

- **`df.selectExpr(”x”, ”y”, …)`** is the same as
  `df.select(expr(x), expr(”y”), …)`

- **`df.selectExpr(”*”, ”a = b as ident”)`**
- **`df.selectExpr(”sum(a)”, ”avg(a)”)`**

Adding and removing columns

- You can also add a column by using **`withColumn`**
  `df.withColumn(”pi”, lit(3.14))`

- You can remove a column with **`drop:`**
  `df.drop(”colname”)`
  `df.drop(df.colname)` both work

Filtering rows

- Use **`where`** or **`filter`** (different names for the same)
- **`df.where(expr(”col < 5”))`**

- You can use ”AND” in the filter expression or just use **`where`** many times **`df.where(f1).where(f2)`** or use & **`df.where(f1 & f2)`**

- You can use ”OR” in the filter expression or use | **`df.where(f1 | f2)`**

- To only keep unique rows, use **`distinct`** **`df.distinct()`**

Nulls

- A DataFrame's schema tells if a column is nullable, …
- …, but Spark **does not** enforce this!

- On a DataFrame df, we can use df.na for null handling
- **`df.na.drop()`** and **`df.na.drop("any")`** drop all rows with one or more nulls
- **`df.na.drop("all")`** drops rows only holding nulls
- Another argument can be a list of columns to consider
  `df.na.drop("any", subset=["col1", "col5"])`
- To replace null values
  `df.na.fill("n/a") #replaces every null`,
  `df.na.fill("n/a", subset=["a", "b"])` or
  `df.na.fill({"a":"n/a", "b":0, "c":"???"})`

Aggregations

- pyspark.sql.functions contain many aggregation functions
  - min, max, sum, avg, count, …
- **`df.select(count("column"))`**
- **`df.select(countDistinct("column"))`**
- countDistinct is exact – but can be expensive
- Perhaps an approximated answer is good enough
- **`df.select(approx_count _distinct \`**
  `("column", 0.05)) # max 5% error`
- Much faster down to 1% allowed error
  - Shuffling of all data not needed, local computations on partitions, and results get combined
  - See https://databricks.com/blog/2016/05/19/approximate-algorithms-in-apache-spark-hyperloglog-and-quantiles.html and https://queue.acm.org/detail.cfm?id=3104030 for explanations

Grouping

- As in SQL, we can group by certain columns and compute aggregated values for each group
- **`gb = df.groupBy(”col1”, ”col2”)`**
- **`gb`** is a **`GroupedData`** object

- **`gb.max(”col3”, ”col4”)`** to use a single agg. function
- **`gb.agg(max(”col3”), min(”col4”), sum(”col5”))`**
- **`gb.agg({”col3”:”max”, ”col4”:”min”, \`**
  `”col5”:”sum”})`
- Return a DataFrame

- CUBE and WINDOWs as known from SQL also supported

SQL CUBE Example

```
SELECT City, Product,
       SUM(Sales) AS Sales
FROM SalesTable
GROUP BY CUBE(City, Product)
```

City	Product	Sales
Aalborg	Milk	300
Copenhagen	Milk	500
Aalborg	Bread	400
Copenhagen	Bread	600

SalesTable

City	Product	Sales
Aalborg	Milk	300
Copenhagen	Milk	500
Aalborg	Bread	400
Copenhagen	Bread	600
Aalborg	*null*	700
Copenhagen	*null*	1100
null	Milk	800
null	Bread	1000
null	*null*	1800

Normal GROUP BY

```
SELECT City, SUM(Sales)
FROM SalesTable
GROUP BY City
```

City	Product	Sales
Aalborg	Milk	300
Copenhagen	Milk	500
Aalborg	Bread	400
Copenhagen	Bread	600

SalesTable

City	SUM
Aalborg	700
Copenhagen	1100

SQL Window Example

```
SELECT City, Product, Sales,
            SUM(Sales)
OVER (PARTITION BY City)
FROM SalesTable
```

City	Product	Sales
Aalborg	Milk	300
Copenhagen	Milk	500
Aalborg	Bread	400
Copenhagen	Bread	600

SalesTable

City	Product	Sales	SUM
Aalborg	Milk	300	700
Aalborg	Bread	400	700
Copenhagen	Milk	500	1100
Copenhagen	Bread	600	1100

SQL Window Example II

```
SELECT City, Product, Sales,
            RANK()
OVER (PARTITION BY City
          ORDER BY Sales DESC)
FROM SalesTable
```

City	Product	Sales
Aalborg	Milk	300
Copenhagen	Milk	500
Aalborg	Bread	400
Copenhagen	Bread	600

SalesTable

City	Product	Sales	RANK
Aalborg	Bread	400	1
Aalborg	Milk	300	2
Copenhagen	Bread	600	1
Copenhagen	Milk	500	2

SQL

- A DataFrame can be registered as a view
- It can then be queried with SQL!
  - No performance difference between DataFrame code and SQL
  - Optimization happens *across* DataFrame expressions and SQL
- **`df.createOrReplaceTempView("viewname")`**
- We can query with **`spark.sql`**:

```
res = spark.sql("select x, max(y), avg(z) \
                from viewname group by x")
```

- **`spark.sql`** returns a DataFrame which we can use in the usual way (and it is computed lazily)

SQL

- Other applications can also query the data via JDBC/ODBC

- We can combine (Scala|Java|Python) DataFrame code and SQL and use the most convenient way for a given step

- Easy for users to work with
- DataFrames can be defined gradually by different functions
- If statements, loops, etc. can be used

User-defined functions (UDFs)

- We can create our own functions and register them
- ```
  from pyspark.sql.functions import udf
  myudf = udf(myfunc)
  df.select(myudf(col("column")))  #works
  df.select(expr("myudf(column)")) #FAILS!
  ```

- To be able to use it in string expressions, we use
- ```
  spark.udf.register("myudf", myfunc, \
  xxxType())
  df.select(expr("myudf(column)")) #now works
  ```
- myudf can then also be used from SQL
  ```
  SELECT myudf(column) FROM t;
  ```

UDFs

- When you make a UDF, it is serialized and sent to the workers
- If your function is written in Scala/Java, it will run in the JVM on the worker
- If it is written in Python, a worker launches a Python process and has to serialize data between the JVM and Python process (both ways). Expensive ☹

- *Why don't we have the same problem when we use pyspark.sql.functions in pyspark –* ***`df.select(expr(…))`*** *?*

Joining data

- `joinexp = df1["id"] == df2["val"]`
- `df1.join(df2, joinexp)`

- `df1.join(df2, joinexp,` ***`type`***`)` for `type =`
  - `"inner" #default`
  - `"outer"`
  - `"left_outer"`
  - `"right_outer"`
  - `"left_semi"`
  - `"left_anti"`
  - `"cross"`

- Or in SQL…

Joining data

- A join can be very expensive if all nodes have to get data from each other
- If the DataFrames are partitioned in the same way, joins can be done locally
- A small DataFrame can also be *broadcast* to all workers, such that joins can be done locally

Reading into DataFrames

- We read via a DataFrameReader available in **`spark.read`**

```
df = spark.read.\
   format(”csv”).\          #def: ”parquet”
   option(key, value).\     #Can be used 0-* times
   schema(mySchema)\.       #optional
   load(filename)           #also ”/dir/*.csv”
```

- Formats: csv, json, parquet, orc, jdbc, text
- Options depend on the format – see options in the book
- The option *mode* decides what to do with malformed records
  - dropMalformed
  - failFast
  - permissive – for malformed records, all fields are null except _corrupt_record holding the malformed data in a string

Example

```
df = spark.read.\
   format(”csv”).\
   option(”inferSchema”, ”true”).\
   option(”header”, ”true”).\
   load(”directory/*.csv”)
```

Could also be written as

```
df = spark.read.\
   option(”inferSchema”, ”true”).\
   option(”header”, ”true”).\
   csv(”directory/*.csv”)
```

Schemas

- Spark can infer the schema – schema-on-read
- The most specific possible type is used
- Fine for ad hoc analysis, but for production, it is recommended to define the schema manually:

```
from pyspark.sql.types import *
mySchema = StructType([
   StructField(”Name1”, StringType, True),
   StructField(”Name2”, IntegerType, False),
   StructField(”Name3”, DateType, True),
   ...
])
df = spark.read.schema(mySchema).csv(”f.csv”)
```

Saving DataFrames

- Similar to reading

```
df.write.format(”csv”) \
   .mode(”overwrite”) \
   .option(”path”, ”/home/chr/output”) \
   .option(”header”, ”true”) \
   .save()
```

- Note that the path is a directory where files will be created for each partition of the DataFrame
- We could also use **`.partitionBy(”year”)`** and we would get subdirectories year=2017, year=2018, …
- Files can also be bucketed (~ hash partitioned)

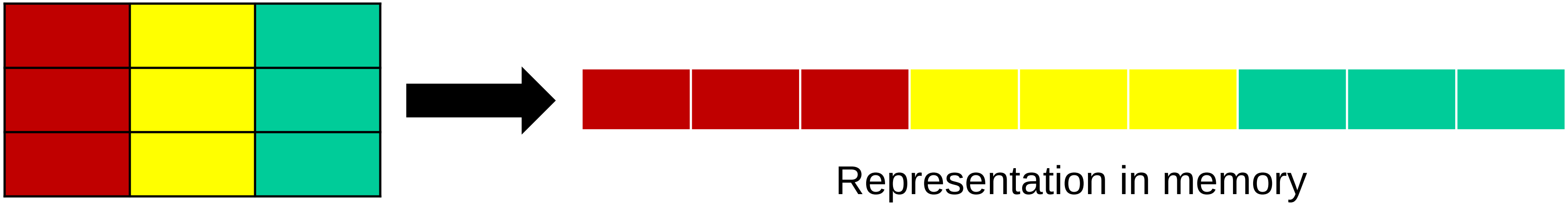
Agenda

- Last time
- Background for Spark's DataFrames
- Using DataFrames
- **How Spark does**

Caching

- Like RDDs, DataFrames can be cached in memory
- **`df.cache()`** or **`df.persist(`*`level`*`)`**

- Spark uses columnar storage in memory



Conceptual DataFrame

Representation in memory

- Compression (run-length, dictionary encoding) reduces space usage

The Catalyst optimizer

- Spark SQL introduced the Catalyst optimizer
- Two goals:
  - Make it easy to add new optimization techniques and features
  - Make the optimizer extensible by external developers

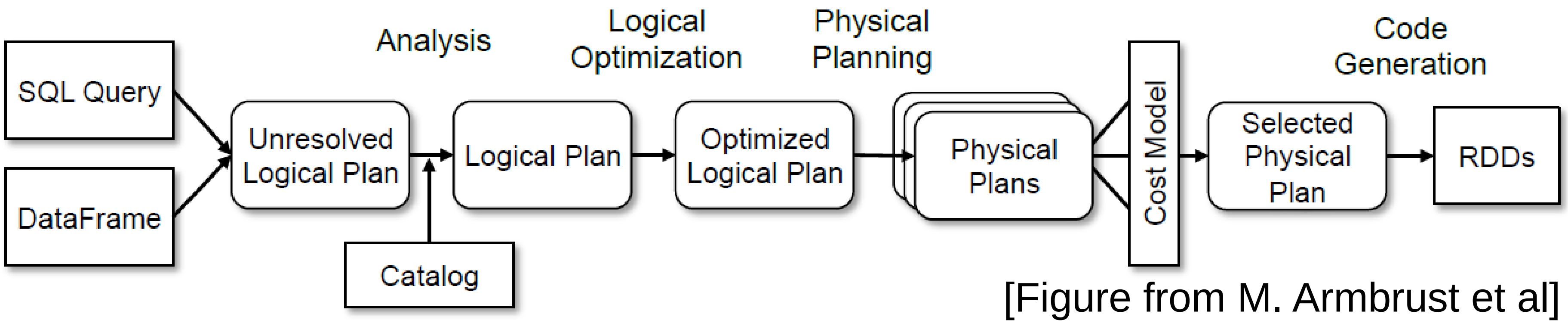
- Catalyst considers *trees* and *rules* to manipulate them
- A rule is a function from a tree to another tree
- A rule can run arbitrary code, but often ”only” does pattern matching where subtrees with a certain structure are replaced
  - Subtract(Literal(a), Literal(b)) ➔ Literal(a – b)

Catalyst

- Catalyst tests where a given rule can be applied
  - Thus, rules can ignore cases where they *don't* apply
  - And therefore existing rules don't have to be changed when more features are added

- Rules are grouped into batches and executed until a fix-point is reached

Query planning



[Figure from M. Armbrust et al]

Analysis

- Starts with an AST from the SQL parser or from a DataFrame object
- Spark builds a logical plan (a tree)
- *Unresolved* when type or existence of an attribute/column is unknown
- DataFrames are computed lazily, but analyzed eagerly

Query planning

Logical optimization

- Standard rules are applied to the tree
- Constant folding (24 * 60 * 60 ➔ 86,400)
- Projection pruning (read only the needed columns)
- Boolean simplification
- Predicate pushdown (apply predicates/filters early)
- And more rules…, e.g.,
  - LIKE ‘a%’ ➔ String.startsWith(”a”)
  - LIKE ‘%a%’ ➔ String.contains(”a”)

Query planning

Physical planning

- Spark generates one or more physical plans matching the execution engine
- More rule-based optimization:
  - Pipeline projections or filters into a single map operation
  - Push operations into sources which support projection or predicate pushdown
    - For example, a DBMS can do this
    - False positives are allowed
- A cost-model is used to choose among the plans

Query planning

Code generation

- Instead of *interpreting* the selected plan, Spark generates byte-code to run on the JVM
- Much processing done on data in memory
  ➔ CPU bound
  ➔ code generation can speed up the execution

Extension points

- Catalyst designed to be extensible
- Batches of rules can be added to optimization phases

- Support for new data sources can also be added
  - Different interfaces to implement depending on what the source is capable of – pruned scan, pruned and filtered scan

- New data types can also be added by providing mappings to/from built-in types

Mini-project

- Assignment for my part on Moodle
- Nguyen’s part will follow

- Hand in by email to Nguyen AND me no later than Monday 11-06-2018 9:00
  - Each student should send an assignment
  - Make sure your name is written on the front page
  - Make sure to add page numbers