Chapter 4. Structured API Overview

Review:
Spark is a distributed programming model in which the user specifies transformations. Multiple transformations build up a directed acyclic graph of instructions. An action begins the process of executing that graph of instructions, as a single job, by breaking it down into stages and tasks to execute across the cluster. The logical structures that we manipulate with transformations and actions are DataFrames and Datasets. To create a new DataFrame or Dataset, you call a transformation. To start computation or convert to native language types, you call an action.

Schema

- a schema defines the column names and types of a DataFrame
- Can define schemas manually or read a schema from a data source (often called schema on read).
- Schemas consist of type defs

Catalyst

- Spark engine
- Maintains its own type information through the planning and processing of work
- Wide variety of execution optimizations that make significant differences

Even if we use Spark's Structured APIs from Python or R, the majority of our manipulations will operate strictly on Spark types. Spark will convert an expression written in an inputlanguage to Spark's internal Catalyst representation of that same type information. It then willoperate on that internal representation

Within Structured APIs, there are two more APIs, the "untyped" DataFrames and the "typed" Datasets.

- DataFrames have types, but Spark maintains them completely and only checks whether those types line up to those specified in the schema at runtime.
- Datasets check whether types conform to the specification at compile time. Datasets are only available to Java Virtual Machine (JVM)–based languages (Scala and Java).

Columns

- represent a simple type like an integer or string, or a complex type like an array, map, or null value.

- Spark tracks all of this type information for you and offers a variety of ways to transform columns.

Row

- Each record in a DataFrame must be of type Row
- Create these rows manually from SQL, from Resilient Distributed Datasets (RDDs), from data sources, or manually from scratch

The following tables provide the detailed type information for each of Spark's language bindings.

Table 4-1. Python type reference

Data type	Value type in Python	API to access or create a data type
ByteType	int or long. Note: Numbers will be converted to 1-byte signed integer numbers at runtime. Ensure that numbers are within the range of –128 to 127.	ByteType()
ShortType	int or long. Note: Numbers will be converted to 2-byte signed integer numbers at runtime. Ensure that numbers are within the range of –32768 to 32767.	ShortType()
IntegerType	int or long. Note: Python has a lenient definition of "integer." Numbers that are too large will be rejected by Spark SQL if you use the IntegerType(). It's best practice to use LongType.	IntegerType()
LongType	long. Note: Numbers will be converted to 8-byte signed integer numbers at runtime. Ensure that numbers are within the range of –9223372036854775808 to 9223372036854775807. Otherwise, convert data to decimal.Decimal and use DecimalType.	LongType()
FloatType	float. Note: Numbers will be converted to 4-byte single-precision floating-point numbers at runtime.	FloatType()

Data type	Value type in Python	API to access or create a data type
ByteType	int or long. Note: Numbers will be converted to 1-byte signed integer numbers at runtime. Ensure that numbers are within the range of –128 to 127.	ByteType()
ShortType	int or long. Note: Numbers will be converted to 2-byte signed integer numbers at runtime. Ensure that numbers are within the range of –32768 to 32767.	ShortType()
IntegerType	int or long. Note: Python has a lenient definition of “integer.” Numbers that are too large will be rejected by Spark SQL if you use the IntegerType(). It’s best practice to use LongType.	IntegerType()
LongType	long. Note: Numbers will be converted to 8-byte signed integer numbers at runtime. Ensure that numbers are within the range of –9223372036854775808 to 9223372036854775807. Otherwise, convert data to decimal.Decimal and use DecimalType.	LongType()
FloatType	float. Note: Numbers will be converted to 4-byte single-precision floating-point numbers at runtime.	FloatType()
DoubleType	float	DoubleType()
DecimalType	decimal.Decimal	DecimalType()
StringType	string	StringType()
BinaryType	bytearray	BinaryType()
BooleanType	bool	BooleanType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
ArrayType	list, tuple, or array	ArrayType(elementType, [containsNull]). Note: The default value of containsNull is True.
MapType	dict	MapType(keyType, valueType, [valueContainsNull]). Note: The default value of valueContainsNull is True.
StructType	list or tuple	StructType(fields). Note: `fields` is a list of StructFields. Also, fields with the same name are not allowed.
StructField	The value type in Python of the data type of this field (for example, Int for a StructField with the data type IntegerType)	StructField(name, dataType, [nullable]) Note: The default value of nullable is True.

Structured API Execution

1. Write DataFrame/Dataset/SQL Code
2. If Valid, Spark converts to *Logical Plan*
3. Spark transforms this Logical Plan to a Physical Plan, checking for optimizations along the way. The *Catalyst Optimizer* decides how the code should be executed and lays

out a plan for doing so.

4. Executes this Physical Plan (RDD manipulations) on the cluster

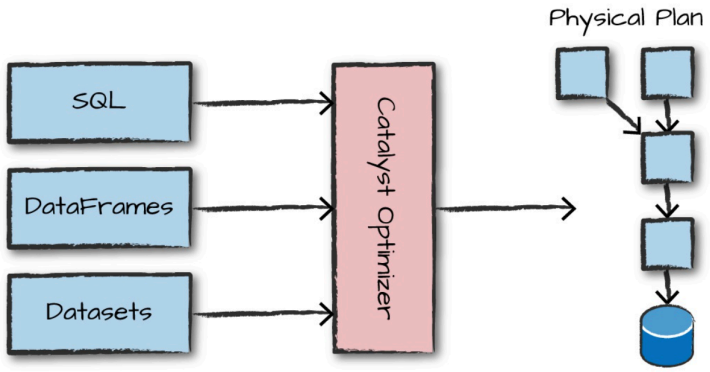

Figure 4-1. The Catalyst Optimizer

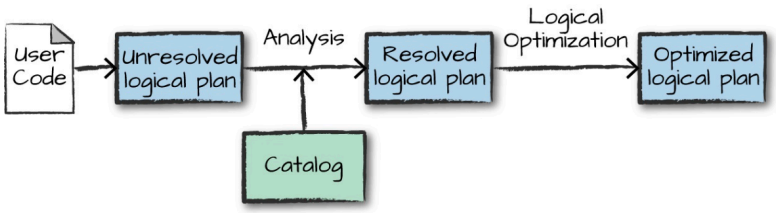

Figure 4-2. The structured API logical planning process

Unresolved logical plan

- represents a set of abstract transformations that do not refer to executors or drivers
- purely to convert the user's set of expressions into the most optimized version
- unresolved because although your code might be valid, the tables or columns that it refers to might or might not exist.

Spark uses the catalog, a repository of all table and DataFrame information, to resolve columns and tables in the analyzer. The analyzer might reject the unresolved logical plan if the required table or column name does not exist in the catalog. If the analyzer can resolve it, the result is passed through the Catalyst Optimizer, a collection of rules that attempt to optimize the logical plan by pushing down predicates or selections. Packages can extend the Catalyst to include their own rules for domain-specific optimization

Physical plan

- specifies how the logical plan will execute on the cluster by generating different

physical execution strategies and comparing them through a cost model

- example of the cost comparison might be choosing how to perform a given join by looking at the physical attributes of a given table (how big the table is or how big its partitions are)

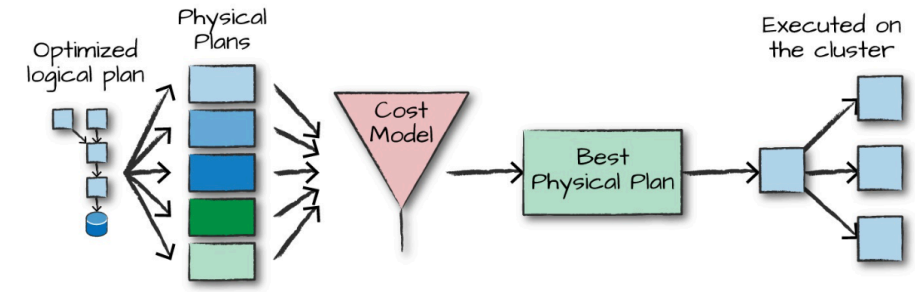

Figure 4-3. The physical planning process

Execution - Spark performs further optimizations at runtime, generating native Java bytecode that can remove entire tasks or stages during execution.