Parquet at Flink

This document gives a deep-dive into leveraging Parquet in Flink. For a general introduction to the integration of Hadoop Input Formats, please refer to Hadoop Compatibility

Parquet

"Parquet is a columnar storage format for Hadoop that supports complex nested data."

Parquet is an Apache open source project. Cloudera and Twitter are the major contributors. The idea for Parquet came from Google. They introduced their system called Dremel which brings the advantages of columnar storage and nested data together. Parquet is implementing this concept in Hadoop. Since Flink provides a seamless integration for Hadoop formats, we can leverage all the advantages of Parquet in Flink.

A columnar storage format brings a lot of advantages. The first one is schema projection. This makes it possible to read only those columns which are really needed in your application.

Moreover column stores are highly compression friendly. This means compression algorithms work faster and can compress better, because information entropy per column is lower than per row. E.g. if you imagine a column `phone number`, the values in this column are really similar. Maybe most of them have even the same area code. This high data value locality allows it to apply all kinds of compression. Parquet supports GZIP, LZO and SNAPPY.

Moreover it is possible to use several types of encoding: Bit Packing, Run Length encoding, Dictionary encoding, … Another nice feature of Parquet is the native implementation of predicate pushdown. This makes it possible to filter records on the lowest level.

The key differentiating factor in comparison to other columnar store formats is that Parquet

allows the user to define a schema which can be arbitrarily nested.

Getting started

The idea of this tutorial is to get you started as quickly as possible. Therefore I setup a Github repository. There you can find sample Maven projects which can serve you as templates for your own projects.

At the moment I provide templates for the following use cases:

1. Parquet at Flink - using Java and Protocol Buffers schema definition
2. Parquet at Flink - using Java and Thrift schema definition
3. Parquet at Flink - using Java and Avro schema definition
4. Parquet at Flink - using Scala and Protocol Buffers schema definition

Each project has two main folders: **commons** and **flink**.

In the **commons** folder you put your schema definition IDL file. The Maven `commons/pom.xml` is configured to build classes from the IDL file during compilation. This makes development more convenient, because you don't need to recompile the IDL file by hand whenever there is any minor change in your schema.

In the **flink** folder there are your Flink jobs which read and write Parquet.

So choose your template project, download the corresponding folder and run:

```
$ mvn clean compile package
```

Define your schema

There are several ways to define the schema of your data. This tutorial covers three data serialization frameworks: Avro, Protocol Buffers (protobuf) and Thrift.

The example schema is the data of a person who has a name, id, email address and several phone numbers:

Protobuf | **Thrift** | **Avro**

```
option java_package = "flink.parquet.proto";
option java_outer_classname = "PersonProto";

message Person {
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;

    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }

    message PhoneNumber {
        required string number = 1;
        optional PhoneType type = 2 [default = HOME];
    }

    repeated PhoneNumber phone = 4;
}
```

Flink dependencies

The Flink dependencies will also imported by the templates. The one important thing to note here is that Flink supports Parquet since version `0.8.1`.

Java | **Scala**

```
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-clients</artifactId>
 <version>0.8.1</version>
</dependency>
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-java</artifactId>
 <version>0.8.1</version>
</dependency>
```

Parquet dependencies

The Parquet dependencies are also provided by the templates. But if you want to build your own Maven configuration you will find this interesting:

Protobuf | **Thrift** | **Avro**

```
<dependency>
 <groupId>com.twitter</groupId>
 <artifactId>parquet-hadoop</artifactId>
 <version>1.6.0rc4</version>
</dependency>
<dependency>
 <groupId>com.twitter</groupId>
 <artifactId>parquet-protobuf</artifactId>
 <version>1.6.0rc4</version>
</dependency>
```

Write Parquet

Once you have defined your schema you can generate some objects and put them into a `DataSet<Tuple2<Void, YourClass>>`. Parquet uses `Void` as key which is just a `null` value. Flink provides the class `HadoopOutputFormat` to write in Hadoop formats. You can set the output path, the compression and the type of encoding. Moreover you have to specify your schema class.

The following example describes the output of Protobuf objects. For the other data serialization frameworks you can find the corresponding source in the Github repository.

Java | **Scala**

```
Job job = Job.getInstance();

// Set up Hadoop Output Format
HadoopOutputFormat parquetFormat = new HadoopOutputFormat(new ProtoParquetOut
putFormat(), job);

FileOutputFormat.setOutputPath(job, new Path(outputPath));

ParquetOutputFormat.setCompression(job, CompressionCodecName.SNAPPY);
ParquetOutputFormat.setEnableDictionary(job, true);

ProtoParquetOutputFormat.setProtobufClass(job, Person.class);

// Output & Execute
data.output(parquetFormat);
```

Read Parquet

The cool thing about Parquet is that it recognises on its own, which compression and encoding is used in the current file. Therefore you only have to specify the input path and the schema of the data which you want to read.

Moreover you can apply native predicate pushdown by defining your self-tailored predicates.

How to create your custom predicate is described in the next chapter.

You can also leverage schema projection. Schema projection is highly dependent on the serialization framework. This is described in detail in the chapter schema projection.

The following example describes the input of Protobuf objects. For the other data serialization frameworks you can find the corresponding source in the Github repository.

Java | **Scala**

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
Job job = Job.getInstance();

HadoopInputFormat hadoopInputFormat = new HadoopInputFormat(
        new ProtoParquetInputFormat(), Void.class, Person.Builder.class, job)
;

FileInputFormat.addInputPath(job, new Path(inputPath));

DataSet<Tuple2<Void,Person.Builder>> data = env.createInput(hadoopInputFormat
);
```

Native predicate pushdown

The easiest type of predicate is an equality predicate. In the example below we accept only those records which have the `name = "Felix"`. You can even specify more complex constraints with and, or, not …

If you want to implement more complex predicates, you find more exampleshere.

Java | **Scala**

```
BinaryColumn name = binaryColumn("name");
FilterPredicate namePred = eq(name, Binary.fromString("Felix"));
ParquetInputFormat.setFilterPredicate(job.getConfiguration(), namePred);
```

Schema projection

Schema projection is very important because it can reduce a large number of I/O disk accesses and therefore speeds up reading. Unfortunately Parquet implemented the schema projection for each data serialization framework differently.

In the box below you can find an example how to apply schema projection and read all records without the attribute `phone.type`.

Protobuf | **Thrift** | **Avro**

```
//schema projection: don't read phone type attribute
String projection = "message Person {\n" +
        "  required binary name (UTF8);\n" +
        "  required int32 id;\n" +
        "  optional binary email (UTF8);\n" +
        "  repeated group phone {\n" +
        "    required binary number (UTF8);\n" +
        "  }\n" +
        "}";
ProtoParquetInputFormat.setRequestedProjection(job, projection);
```

Experiments

When it comes to file formats, two metrics are especially interesting:

1. Is the data representation space efficient?
2. Does the format has a good read performance?

In order to get an idea to answer these two questions, I implemented query 55 of the TPC-DS Benchmark.

In SQL query 55 looks like this:

SQL

```
SELECT
     i_brand_id AS brand_id,
     i_brand AS brand,
     SUM(ss_ext_sales_price) AS ext_price
FROM date_dim,
     store_sales,
     item
WHERE date_dim.d_date_sk = store_sales.ss_sold_date_sk
      AND store_sales.ss_item_sk = item.i_item_sk
      AND i_manager_id = 28
      AND d_moy = 11
      AND d_year = 1999
GROUP BY i_brand, i_brand_id
ORDER BY ext_price desc, i_brand_id
LIMIT 100
```

This query joins three tables and selects only very few of the corresponding columns. This emphasizes the strength of Parquet: schema projection.

For the experiments I used the very small scaling factor 20 (which generates 20GB of CSV data). The three tables Date_Dim, Store_Sales and Item are only a part of this data (8GB).

Using Snappy compression and dictionary encoding the three tables are compressed to half their original size. This shows that Parquet is highly space efficient.

To compare the reading performance, I implemented a CSV reader variant for the same query. The result: We gain a speed up of up to 2 using Parquet. This speed up will even increase when it comes to greater scaling factors.

This is the perfect use case for Parquet. We are only interested in 10 columns out of a total of 73 columns. Because of the column store architecture the Parquet reader only needs to read the 10 columns whereas the CSV reader has to read all 73 columns. If this ratio is not this drastic, the CSV reader is faster than the Parquet reader.

You can find my implementation of TPC-DS query 55 on Github.

Moreover, I also implemented query 3 of TPC-H Benchmark:

SQL

```
SELECT
     l_orderkey,
     SUM(l_extendedprice * (1 - l_discount)) AS revenue,
     o_orderdate,
     o_shippriority
FROM customer,
     orders,
     lineitem
WHERE
     c_mktsegment = 'AUTOMOBILE'
     AND c_custkey = o_custkey
     AND l_orderkey = o_orderkey
     AND o_orderdate < date '1995-03-12'
     AND l_shipdate > date '1995-03-12'
GROUP BY
     l_orderkey,
     o_orderdate,
     o_shippriority;
```

The compression works in this case even better. The CSV files are compressed down to one third. But since the column selectivity of this query is not that drastic anymore (10 out of 33 columns) the Parquet reading performance in comparison to the CSV reader is rather bad in this case. For TPC-H query 3, the CSV reader is twice as fast as the Parquet reader.

The implementation of the corresponding query can also be found on Github

These two experiments only give a small glimpse on the performance of Parquet. You find more detailed information on the Parquet web page.

Conclusion

Parquet is an extremely useful format to store real Big Data. It is highly flexible because it is not bound by any limitations in the data schema. Parquet is especially awesome when a query has low selectivity in terms of columns. Then the column store architecture really pays of.