How to use Parquet as a basis for ETL and analytics

Julien Le Dem @J_
Analytics Data Pipeline tech lead, Data Platform

@ApacheParquet
Outline

- Instrumentation and data collection
- Storing data efficiently for analysis
- Openness and Interoperability
Instrumentation and data collection
Typical data flow

Happy users

Instrumented Services

mutation

Mutable Serving stores
Typical data flow

- Instrumented Services
- Mutation
- Log
- Scribe
- Chukwa
- Streaming log
- Pull
- Log collection
- Periodic snapshots
- Periodic consolidation
- Snapshots
- Streaming analysis
Typical data flow

Instrumented Services

Mutation

Serving stores

Logging

Streamlining log (Kafka, Scribe, Chukwa ...)

Data collection

Streaming log

Periodic snapshots

Pull

Analysis

Streaming computation (Storm, Samza, SparkStreaming...)

Batch computation (Graph, machine learning, ...)

Query-efficient format (Parquet)

Storage (HDFS)

Ad-hoc queries (Impala, Hive, Drill, ...)

Automated dashboard

Pull

Periodic consolidation

Snapshots

Format (Parquet)

Streaming analysis

Consolidation

Periodic snapshots

Log collection

Pull
Typical data flow

- **Instrumented Services**
  - Mutation
  - Pull
- **Mutable Serving stores**
- **Serving**
- **Streaming log** (Kafka, Scribe, Chukwa ...)
  - Log collection
  - Periodic snapshots
  - Pull
- **Data collection**
  - Streaming analysis
  - Periodic consolidation
  - Snapshots
- **Batch computation** (Graph, machine learning, ...)
  - Query-efficient format
    - Parquet
  - Storage (HDFS)
  - Ad-hoc queries
    - Impala, Hive, Drill, ...
  - Automated dashboard
- **Analysis**

Happy Data Scientist
Storing data for analysis
Producing a lot of data is easy

Producing a lot of derived data is even easier.
Solution: Compress all the things!
Scanning a lot of data is easy

... but not necessarily fast.
Waiting is not productive. We want faster turnaround.
Compression but not at the cost of reading speed.
Interoperability not that easy

We need a storage format interoperable with all the tools we use and keep our options open for the next big thing.
Enter Apache Parquet
Parquet design goals

- Interoperability
- Space efficiency
- Query efficiency
Efficiency
Columnar storage

Logical table representation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
<td>c2</td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
<td>c3</td>
</tr>
<tr>
<td>a4</td>
<td>b4</td>
<td>c4</td>
</tr>
<tr>
<td>a5</td>
<td>b5</td>
<td>c5</td>
</tr>
</tbody>
</table>

Row layout

```
a1 b1 c1 a2 b2 c2 a3 b3 c3 a4 b4 c4 a5 b5 c5
```

Column layout

```
a1 a2 a3 a4 a5 b1 b2 b3 b4 b5 c1 c2 c3 c4 c5
```

Encoded chunk

```
encoded chunk encoded chunk encoded chunk
```
Parquet nested representation

Borrowed from the Google Dremel paper

Schema:

Columns:
docid
links.backward
links.forward
name.language.code
name.language.country
name.url

https://blog.twitter.com/2013/dremel-made-simple-with-parquet
## Statistics for filter and query optimization

- **Vertical partitioning** (projection push down)
- **Horizontal partitioning** (predicate push down)

### Vertical partitioning

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
<td>c2</td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
<td>c3</td>
</tr>
<tr>
<td>a4</td>
<td>b4</td>
<td>c4</td>
</tr>
<tr>
<td>a5</td>
<td>b5</td>
<td>c5</td>
</tr>
</tbody>
</table>

### Horizontal partitioning

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
<td>c2</td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
<td>c3</td>
</tr>
</tbody>
</table>

### Read only the data you need!

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
<td>c2</td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
<td>c3</td>
</tr>
</tbody>
</table>

Adding the vertical and horizontal partitions gives:

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
<td>c2</td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
<td>c3</td>
</tr>
<tr>
<td>a4</td>
<td>b4</td>
<td>c4</td>
</tr>
<tr>
<td>a5</td>
<td>b5</td>
<td>c5</td>
</tr>
</tbody>
</table>
```
Properties of efficient encodings

- Minimize CPU pipeline bubbles:
  highly predictable branching
  reduce data dependency

- Minimize CPU cache misses
  reduce size of the working set
The right encoding for the right job

- Delta encodings: for sorted datasets or signals where the variation is less important than the absolute value. (timestamp, auto-generated ids, metrics, …) Focuses on avoiding branching.

- Prefix coding (delta encoding for strings)
  When dictionary encoding does not work.

- Dictionary encoding:
  small (60K) set of values (server IP, experiment id, …)

- Run Length Encoding:
  repetitive data.
Interoperability
Interoperable

- Model agnostic
  - Assembly/striping
  - Column encoding
  - Parquet file format

- Language agnostic
  - Java
    - Avro
    - Thrift
    - Protocol Buffer
    - Pig Tuple
    - Hive SerDe
  - C++
    - Impala
      - Query execution
      - Encoding

- Converters
  - parquet-avro
  - parquet-thrift
  - parquet-proto
  - parquet-pig
  - parquet-hive
  - ...

- Object model
  - Java
    - Avro
    - Thrift
    - Protocol Buffer
    - Pig Tuple
    - Hive SerDe
  - C++
    - Impala
Frameworks and libraries integrated with Parquet

**Query engines:**
Hive, Impala, HAWQ,
IBM Big SQL, Drill, Tajo,
Pig, Presto

**Frameworks:**
Spark, MapReduce, Cascading,
Crunch, Scalding, Kite

**Data Models:**
Avro, Thrift, ProtocolBuffers,
POJOs
Schema management
Hadoop does not define a standard notion of schema but there are many available:
  - Avro
  - Thrift
  - Protocol Buffers
  - Pig
  - Hive
  - ...
And they are all different
What they define

**Schema:**
- Structure of a record
- Constraints on the type

**Row oriented binary format:**
- How records are represented one at a time
What they *do not* define

**Column oriented binary format:**
Parquet reuses the schema definitions and provides a common column oriented binary format
Example: address book

```
AddressBook

addresses

Address

street
city
state
zip
comment
```
Protocol Buffers

message AddressBook {
  repeated group addresses = 1 {
    required string street = 2;
    required string city = 3;
    required string state = 4;
    required string zip = 5;
    optional string comment = 6;
  }
}

- Allows recursive definition
- Types: Group or primitive
- binary format refers to field ids only => Renaming fields does not impact binary format
- Requires installing a native compiler separated from your build

Lists are repeated fields

Fields have ids and can be optional, required or repeated
Thrift

struct AddressBook {
  1: required list<Address> addresses;
}

struct Addresses {
  1: required string street;
  2: required string city;
  3: required string state;
  4: required string zip;
  5: optional string comment;
}

- No recursive definition
- Types: Struct, Map, List, Set, Union or primitive
- Binary format refers to field ids only \Rightarrow Renaming fields does not impact binary format
- Requires installing a native compiler separately from the build

**explicit collection types**

**Fields have ids and can be optional or required**
Avro

```json
{
  "type": "record",
  "name": "AddressBook",
  "fields": [{
    "name": "addresses",
    "type": "array",
    "items": {
      "type": "record",
      "fields": [
        {"name": "street", "type": "string"},
        {"name": "city", "type": "string"},
        {"name": "state", "type": "string"},
        {"name": "zip", "type": "string"},
        {"name": "comment", "type": ["null", "string"]}
      ]
    }
  }]
}
```

- Allows recursive definition
- Types: Records, Arrays, Maps, Unions or primitive
- Binary format requires knowing the write-time schema
  - more compact but not self descriptive
  - renaming fields does not impact binary format
- generator in java (well integrated in the build)

null is a type
- Optional is a union

explicit collection types
Write to Parquet
Write to Parquet with Map Reduce

**Protocol Buffers:**

```java
job.setOutputFormatClass(ProtoParquetOutputFormat.class);
ProtoParquetOutputFormat.setProtobufClass(job, AddressBook.class);
```

**Thrift:**

```java
job.setOutputFormatClass(ParquetThriftOutputFormat.class);
ParquetThriftOutputFormat.setThriftClass(job, AddressBook.class);
```

**Avro:**

```java
job.setOutputFormatClass(AvroParquetOutputFormat.class);
AvroParquetOutputFormat.setSchema(job, AddressBook.SCHEMA$);
```
Write to Parquet with Scalding

// define the Parquet source
case class AddressBookParquetSource(override implicit val dateRange: DateRange)
    extends HourlySuffixParquetThrift[AddressBook]("/my/data/address_book", dateRange)

// load and transform data
...
    pipe.write(ParquetSource())
Write with Parquet with Pig

STORE mydata
    INTO 'my/data'
    USING parquet.pig.ParquetStorer();
Query engines
Scalding

loading:
new FixedPathParquetThrift[AddressBook]("my", "data") {
  val city = StringColumn("city")
  override val withFilter: Option[FilterPredicate] =
    Some(city === "San Jose")
}

operations:
p.map( (r) => r.a + r.b )
p.groupBy( (r) => r.c )
p.join
...
Pig

loading:
mydata = LOAD 'my/data' USING parquet.pig.ParquetLoader();

operations:
A = FOREACH mydata GENERATE a + b;
B = GROUP mydata BY c;
C = JOIN A BY a, B BY b;
Hive

loading:
create table parquet_table_name (x INT, y STRING)
    ROW FORMAT SERDE 'parquet.hive.serde.ParquetHiveSerDe'
    STORED AS
    INPUTFORMAT "parquet.hive.MapredParquetInputFormat"
    OUTPUTFORMAT "parquet.hive.MapredParquetInputFormat";

operations:
    SQL!
Impala

loading:
create table parquet_table (x int, y string) stored as parquetfile;
insert into parquet_table select x, y from some_other_table;
select y from parquet_table where x between 70 and 100;

operations:
  SQL!
SELECT * FROM dfs.`/my/data`
Spark SQL

loading:
val address = sqlContext.parquetFile("/my/data/addresses")

operations:
val result = sqlContext
  .sql("SELECT city FROM addresses WHERE zip == 94707")
result.map((r) => ...)

Community
Parquet timeline

- Fall 2012: Twitter & Cloudera merge efforts to develop columnar formats
- March 2013: OSS announcement; Criteo signs on for Hive integration
- July 2013: 1.0 release. 18 contributors from more than 5 organizations.
- May 2014: Apache Incubator. 40+ contributors, 18 with 1000+ LOC. 26 incremental releases.
- Parquet 2.0 coming as Apache release
Thank you to our contributors

[Graph showing contributors over time with annotations for Open Source announcement and 1.0 release]
Get involved

Mailing lists:
  - dev@parquet.incubator.apache.org

Parquet sync ups:
  - Regular meetings on google hangout
Questions

Questions.foreach( answer(_) )

@ApacheParquet