How to use Parquet
as a basis for ETL and analytics

Julien Le Dem @J_

Analytics Data Pipeline tech lead, Data Platform

@ApacheParquet

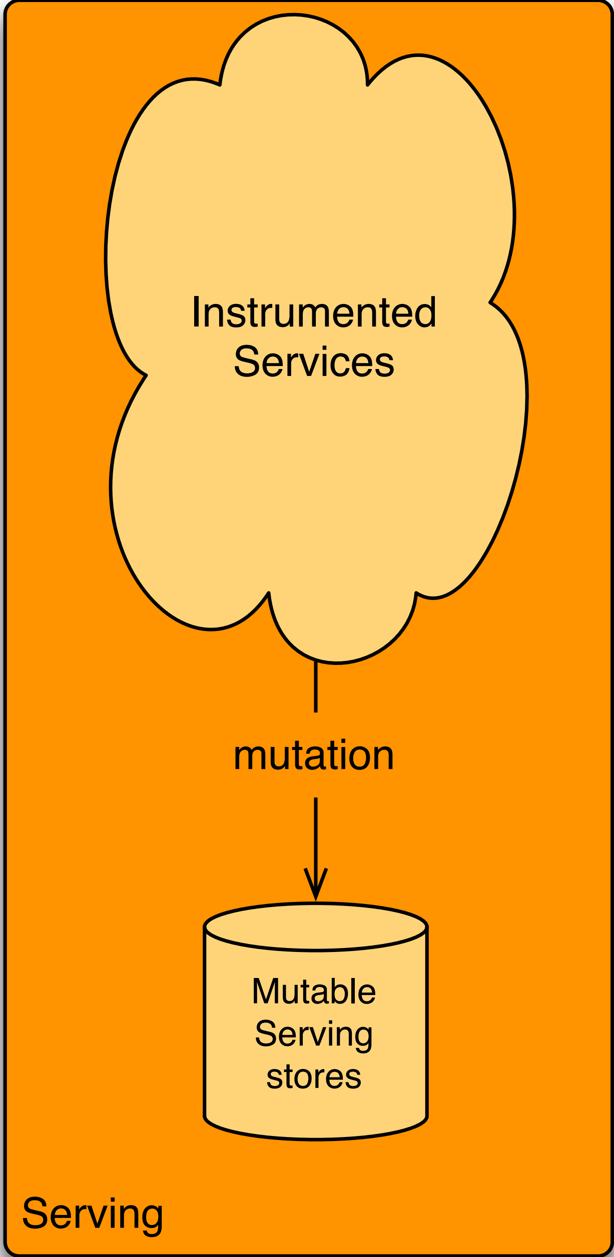
Outline

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

Instrumentation and data collection

Typical data flow

Happy users



Typical data flow

Instrumented Services
log
mutation
Mutable Serving stores
Serving

schema

Streaming log (Kafka, Scribe, Chukwa ...)
Pull
log collection
periodic snapshots
Pull
Data collection

streaming analysis
periodic consolidation
snapshots

Typical data flow

Instrumented Services

log

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

streaming analysis

Streaming computation (Storm, Samza, SparkStreaming..)

mutation

Mutable Serving stores

Serving

Pull

log collection

periodic snapshots

Data collection

periodic consolidation

snapshots

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

Query-efficient format Parquet

Storage (HDFS)

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

automated dashboard

analysis

Typical data flow

Instrumented Services
mutation
Mutable Serving stores
Serving

log
Pull

Streaming log (Kafka, Scribe, Chukwa ...)
Pull
log collection
periodic snapshots
Data collection

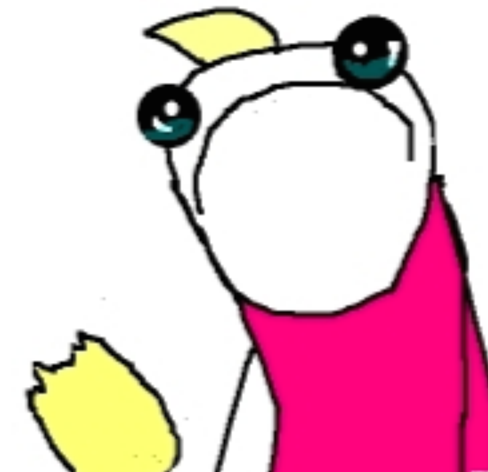
streaming analysis
periodic consolidation
snapshots

Streaming computation (Storm, Samza, SparkStreaming..)
Batch computation (Graph, machine learning, ...)
Query-efficient format Parquet
Storage (HDFS)
ad-hoc queries (Impala, Hive, Drill, ...)
automated dashboard
analysis

Happy Data Scientist

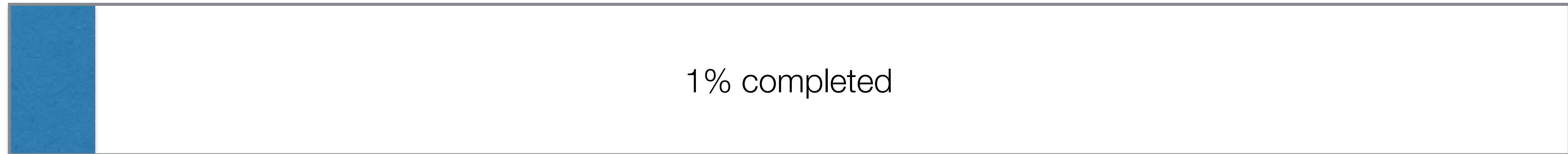
Storing data for analysis

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.

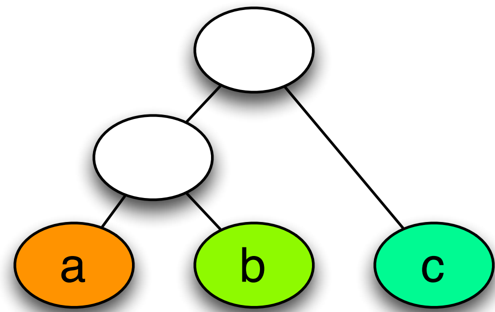
Enter Apache Parquet

Parquet design goals

- Interoperability
- Space efficiency
- Query efficiency

Efficiency

Columnar storage

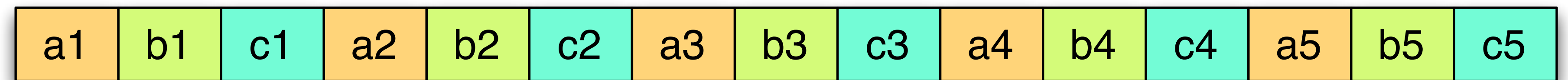


Nested schema

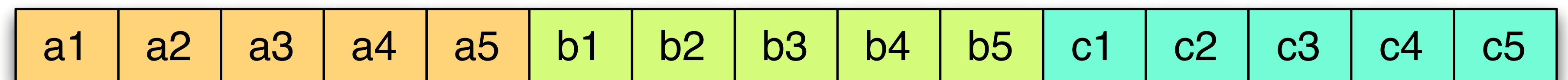
Logical table representation

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

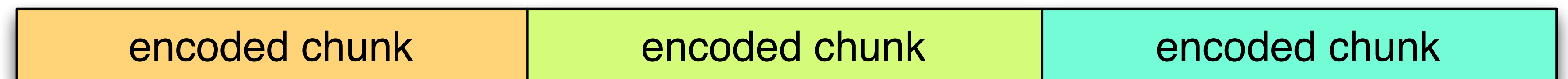
Row layout



Column layout

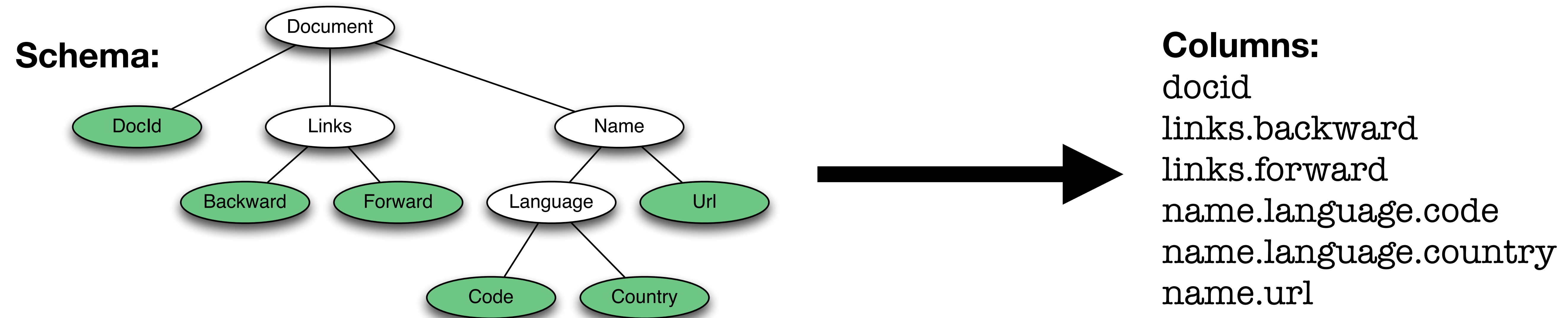


encoding



Parquet nested representation

Borrowed from the Google Dremel paper



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)	=	Read only the data you need!

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

+

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

=

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

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

Java | C++

Object model	Avro	Thrift	Protocol Buffer	Pig Tuple	Hive SerDe	...	Impala
Converters	parquet-avro	parquet-thrift	parquet-proto	parquet-pig	parquet-hive	...	Query execution
Model agnostic	Assembly/striping						
	Column encoding						Encoding
Language agnostic	Parquet file format						

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

Schema in Hadoop

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

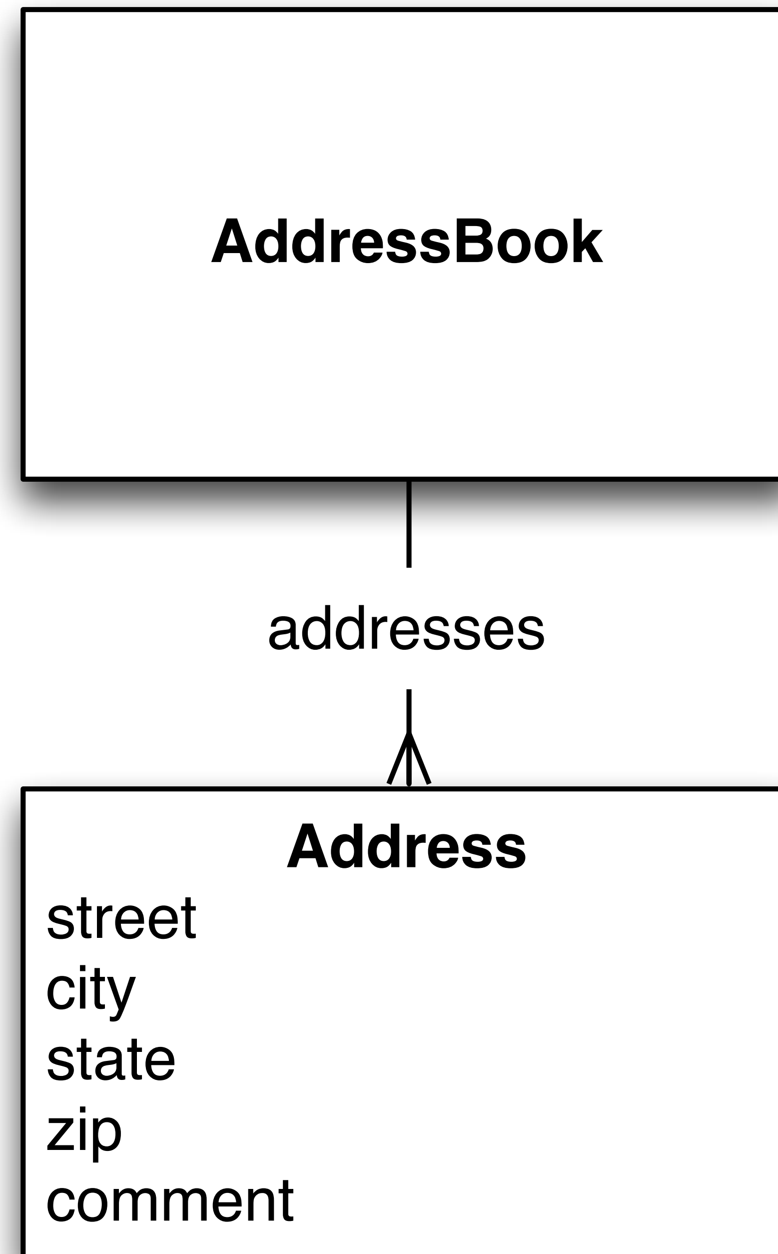
Row oriented binary format:
How records are represented one at a time

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



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;
  }
}
```

Lists are repeated fields

Fields have ids and can be optional, required or repeated

- 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

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;
}
```

explicit collection types

Fields have ids and can be optional or required

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

Avro

```
{
  "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"]}
      ]
    }
  }]
}
```

explicit collection types

null is a type
Optional is a union

- 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)

Write to Parquet

Write to Parquet with Map Reduce

Protocol Buffers:

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

Thrift:

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

Avro:

```
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!

Drill

```
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

Contributors

y-axis	# contributors (0, 10, 20, 30, 40, 50)
x-axis	10/1/2012, 1/1/2013, 4/1/2013, 7/1/2013, 10/1/2013, 1/1/2014, 4/1/2014

Get involved

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

Parquet sync ups:
- Regular meetings on google hangout

Questions

@ApacheParquet

Questions.foreach(answer(_))