Hadoop Data Formats

Prof. Dr. Stephan Trahasch
Hochschule Offenburg

Outline

- Motivation
- Avro
- Parquet
- Optimized Row-Columnar - ORC
- Summary

In the Beginning…

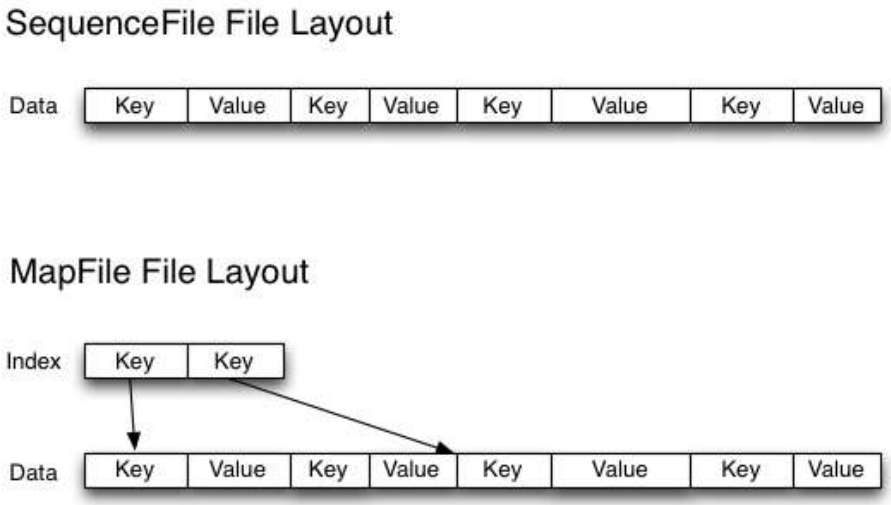
- Hadoop applications used text or SequenceFile
  - Text is slow and not splittable when compressed
  - SequenceFile only supports key and value and user-defined serialization
- Hive added RCFile
  - User controls the columns to read and decompress
  - No type information and user-defined serialization
  - Finding splits was expensive
- Avro files created
  - Type information included!
  - Had to read and decompress entire row

Hadoop File Formats: Text Files

- Text files are the most basic file type in Hadoop
  - Can be read or written from virtually any programming language
  - Comma- and tab-delimited files are compatible with many applications
- Text files are human readable, since everything is a string
  - Useful when debugging
- At scale, this format is inefficient
  - Representing numeric values as strings wastes storage space
  - Difficult to represent binary data such as images
  - Often resort to techniques such as Base64 encoding
  - Conversion to/from native types adds performance penalty
- Good interoperability, but poor performance

Hadoop File Formats: Sequence Files

- SequenceFiles store key-value pairs in a binary container format
- Less verbose and more efficient than text files
- Capable of storing binary data such as images
- Format is Java-specific and tightly coupled to Hadoop



http://blog.cloudera.com/wp-content/uploads/2011/03/sequencefile.png

Good performance, but poor interoperability

Hadoop File Formats: Avro Data Files

- Efficient storage due to optimized binary encoding
- Widely supported throughout the Hadoop ecosystem
- Ideal for long-term storage of important data
  - Can read and write from many languages
  - Embeds schema in the file, so will always be readable
  - Schema evolution can accommodate changes

Excellent interoperability and performance

Best choice for general-purpose storage in Hadoop

Columnar Formats

Hadoop also supports columnar format

- These organize data storage by column, rather than by row
- Very efficient when selecting only a small subset of a table's columns

id	name	city	occupation	income	phone
1	Alice	Palo Alto	Accountant	85000	650-555-9748
2	Bob	Sunnyvale	Accountant	81500	650-555-8865
3	Bob	Palo Alto	Dentist	196000	650-555-7185
4	Bob	Palo Alto	Manager	87000	650-555-2518
5	Carol	Palo Alto	Manager	79000	650-555-3951
6	David	Sunnyvale	Mechanic	62000	650-555-4754

Organization of data in traditional row--based formats

id	name	city	occupation	income	phone
1	Alice	Palo Alto	Accountant	85000	650-555-9748
2	Bob	Sunnyvale	Accountant	81500	650-555-8865
3	Bob	Palo Alto	Dentist	196000	650-555-7185
4	Bob	Palo Alto	Manager	87000	650-555-2518
5	Carol	Palo Alto	Manager	79000	650-555-3951
6	David	Sunnyvale	Mechanic	62000	650-555-4754

Organization of data in columnar formats

Hadoop File Formats: Parquet Files

- Parquet is a columnar format developed by Cloudera and Twitter
  - Supported in Spark, MapReduce, Hive, Pig, Impala, and others
  - Schema metadata is embedded in the file (like Avro)
- Uses advanced optimizations described in Google's Dremel paper
  - Reduces storage space
  - Increases performance
- Most efficient when adding many records at once
- Some optimizations rely on identifying repeated patterns


Dremel: Interactive Analysis of Web-Scale Datasets

Excellent interoperability and performance

Best choice for column-based access patterns

Outline

- Motivation
- Avro
- Parquet
- Optimized Row-Columnar - ORC
- Summary

Avro

- Doug Cutting created Avro, a data serialization and RPC library, to help improve data interchange, interoperability, and versioning in Hadoop Eco System.
- Serialization & RPC Library and also storage format.

What led to a new serialization mechanism?

- Thrift is not splittable and codegen required.
- Dynamic reading is not possible.
- Sequence files does not have schema evolution.
- Evolved as in-house serialization and RPC library for Hadoop. Good overall performance.

Some Features of Avro

- Dynamic Access – No need of Code generation for accessing the data.
- Untagged Data – Which allows better compression
- Platform in-dependent – Has libraries in Java, Scala, Python, Ruby, C and C#. Compressible and splittable – Complements the parallel processing systems such as MR and Spark.
- Schema Evolution:
  "Data models evolve over time", and it's important that your data formats support your need to modify your data models. Schema evolution allows you to add, modify, and in some cases delete attributes, while at the same time providing backward and forward compatibility for readers and writers

Some Features of Avro

- Row Based
- Direct mapping from/to JSON
- Best compatibility for evolving data schemas
- Provides rich data structures
- Map keys can only be strings (could be seen as a limitation)
- Compact binary form
- Extensible schema language
- Untagged data
- Bindings for a wide variety of programming languages
- Dynamic typing
- Provides a remote procedure call
- Supports block compression
- Avro files are splittable

Primitive Types

Name	Description	Java Equivalent
`null`	An absence of a value	`null`
`boolean`	A binary value	`boolean`
`int`	32–bit signed integer	`int`
`long`	64–bit signed integer	`long`
`float`	Single--precision floating point value	`float`
`double`	Double--precision floating point value	`double`
`bytes`	Sequence of 8–bit unsigned bytes	`java.nio.ByteBuffer`
`string`	Sequence of Unicode characters	`java.lang.CharSequence`

Complex Types

Name	Description
`record`	A user-defined type composed of one or more named fields
`enum`	A specified set of values
`array`	Zero or more values of the same type
`map`	Set of key-value pairs; key is string while value is of specified type
`union`	Exactly one value matching a specified set of types
`fixed`	A fixed number of 8-bit unsigned bytes

Example

```
{
"type": "record",
"name": "Meetup",
"fields":  [ {
    "name": "name",
    "type": "string”,
    "order" :
    "descending"
}, {
    "name": "value",
    "type": ["null",
    "string”]
}
…
]
}
```

- JSON Format is used to define schema
- Schemas will be exchanged to ensure the data correctness (RPC)
- You can specify order (Ascending or Descending) for fields.
- Simpler than IDL (Interface Definition Language) of Protocol Buffers and thrift

Basic Schema Example

Excerpt from a SQL CREATE TABLE statement

```
CREATE TABLE employees
   (id INT, name STRING, title STRING, bonus INT)
```

Equivalent Avro schema

```
{"namespace":"com.loudacre.da
 ta",  "type":"record",
 "name":"Employee
 ",  "fields":[
     {"name":"id", "type":"int"},
     {"name":"name", "type":"string"},
     {"name":"title", "type":"string"},
     {"name":"bonus", "type":"int"}]
}
```

Complex Types

The following example shows a record with an enum and a string array

```
{"namespace":"com.loudacre.data",
"type":"record",  "name":"CustomerServi
"fields":[
    {"name":"id", "type":"int"},
    {"name":"agent", "type":"string"},
    {"name":"category","type":{
          "name":"CSCategory", "type":"enum",
          "symbols":["Order","Shipping","Device"]}
    },
    {"name":"tags", "type":{
          "type":"array", "items":"string"}
    }]
}
```

The **category** field has three enumerated possible values

tags is an array of strings

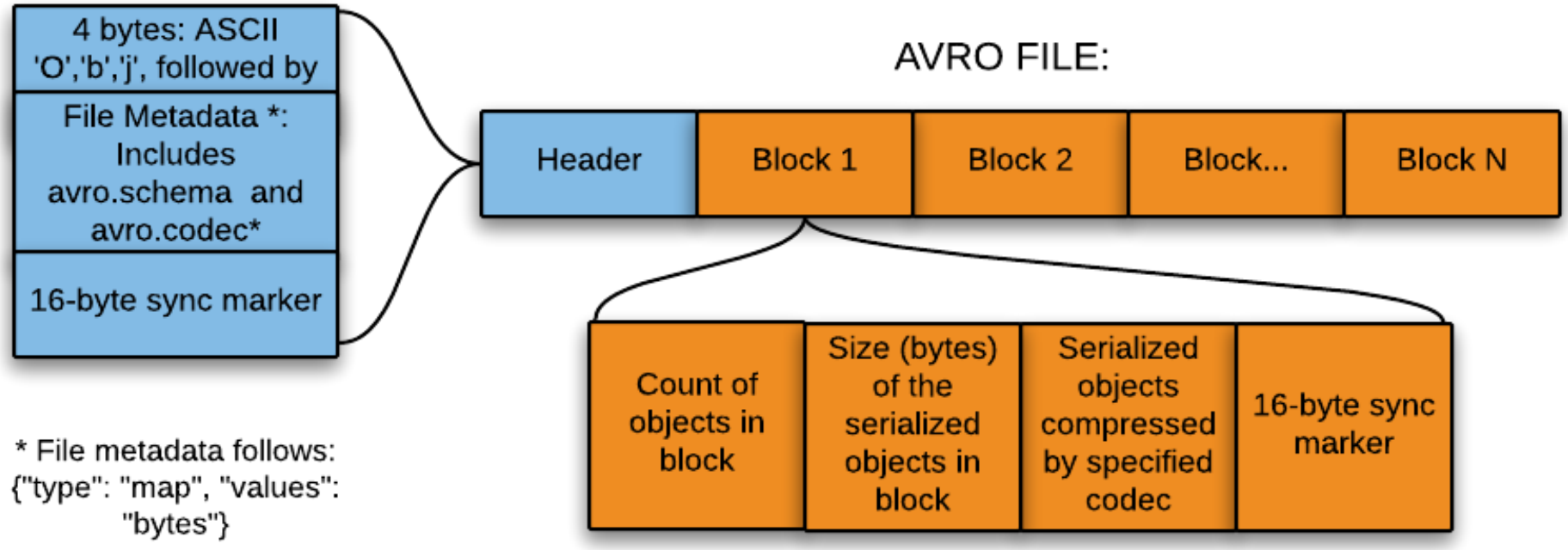
Documenting Your Schema

```
{"namespace":"com.loudacre.data",
 "type":"record",
 "name":"WebProduct",
 "doc":"Item currently sold in Loudacre's online store",
 "fields":[
    {"name":"id", "type":"int", "doc":"Product SKU"},
    {"name":"shipwt", "type":"int",
          "doc": "Shipping weight, in pounds"},
    {"name":"price", "type":"int",
          "doc":"Retail price, in cents (US)"}]
}
```

Avro also defines a container file format for storing Avro records

- Also known as “Avro data file format”
  - Similar to Hadoop SequenceFile format
  - Cross-language support for reading and writing data
  - Supports compressing blocks (groups) of records
- It is “splittable” for efficient processing in Hadoop
- This format is self-describing
  - Each file contains a copy of the schema used to write its data
  - All records in a file must use the same schema

File Structure



Inspecting Avro Data Files with Avro Tools

- The binary format makes debugging difficult
- Each Avro release contains an Avro Tools JAR file
- Allows you to read the schema or data for an Avro file
- Available for download from the Avro Web site

```
$ avro-tools tojson mydatafile.avro
{"name":"Alice","salary": 56500,"city":"Anaheim"}
{"name":"Bob","salary": 51400,"city":"Bellevue"}

$ avro-tools getschema mydatafile.avro
{
  "type" : "record",
  "name" :
  "DeviceData",
  "namespace" : "com.loudacre.data",    ...rest of schema follows
```

Schema Evolution

- The structure of your data will change over time
  - Fields may be added, removed, changed, or renamed
  - In SQL, these are handled with ALTER TABLE statements
- These changes can break compatibility with many formats
  - Objects serialized in SequenceFiles become unreadable
- Data written to Avro data files is always readable
  - The schema used to write the data is embedded in the file itself
  - However, an application reading data might expect the new structure
- Avro has a unique approach to maintaining forward compatibility
- A reader can use a different schema than the writer

Schema Evolution

```
{"namespace": "com.loudacre.data",
 "type": "record",
 "name": "CustomerContact",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name", "type": "string"},
     {"name": "faxNumber", "type": "string"}
]}
```

Schema Evolution

```
{"namespace":
 "com.loudacre.data",  "type":
 "record",

 "name":
 "CustomerContact",
 "fields": [
      {"name": "customerId", "type": "long"},
      {"name": "name", "type": "string"},
      {"name": "prefLang", "type": "string"},
      {"name": "email", "type": "string"}
]}
```

- Rename `id` field to `customerId` and change type from `int` to `long`
- Remove `faxNumber` field
- Add `prefLang` field
- Add `email` field

Schema Evolution

```
{"namespace":
 "com.loudacre.data",  "type":
 "record",

 "name":
 "CustomerContact",
 "fields": [
    {"name":"customerId",
         "type":"long",
         "aliases":["id"]},
    {"name":"name", "type":"string"},
    {"name":"prefLang", "type":"string"},
    {"name":"email", "type":"string"}
]}
```

If you rename a field, you must specify an alias for the old name(s)

Schema Evolution

```
{"namespace": "com.loudacre.data",
"type": "record",
"name": "CustomerContact",
"fields": [
    {"name":"customerId", "type":"long", "aliases":["id"]},
    {"name":"name", "type":"string"},
    {"name":"prefLang", "type":"string", "default":"en_US"},
    {"name":"email",
             "type":["null","string"],"default":null}
]}
```

Newly-added fields will lack values for records previously written

You must specify a default value

The following are some changes that might break compatibility

- Changing the record's name or namespace attributes
- Adding a new field without a default value
- Removing a symbol from an enum
- Removing a type from a union
- Modifying a field's type to one that could result in truncation
- To handle these incompatibilities
- Read your old data (using the original schema)
- Modify data as needed in your application
- Write the new data (using the new schema)
- Existing readers/writers may need to be updated to use new schema

Using Avro with Sqoop and Hive

```
$ sqoop import --connect jdbc:mysql://localhost/loudacre \
 --username training --password training --table accounts \
 --target-dir /loudacre/accounts_avro  --as-avrodatafile
```

```
CREATE TABLE order_details_avro STORED AS AVRO TBLPROPERTIES
('avro.schema.url'='hdfs://localhost/loudacre/accounts_schema.
json');
```

```
CREATE TABLE order_details_avro STORED
  AS AVRO
  TBLPROPERTIES ('avro.schema.literal'=
    '{"name": "order",
      "type": "record",
      "fields": [
          {"name":"order_id", "type":"int"},
          {"name":"cust_id", "type":"int"},
          {"name":"order_date", "type":"string"}
       ]}');
```

Outline

- Motivation
- Avro
- Parquet
- Optimized Row-Columnar - ORC
- Summary

PARQUET - Introduction

- Columnar storage format that come out of a collaboration between Twitter and Cloudera based on Dremel
- To have a state of the art columnar storage available across the Hadoop platform
  - Hadoop is very reliable for big long running queries but also IO heavy.
  - Incrementally take advantage of column based storage in existing framework.
  - Not tied to any framework in particular
- Columnar Storage
  - Limits IO to data actually needed:
    loads only the columns that need to be accessed.
  - Saves space:
    Columnar layout compresses better
  - Type specific encodings.

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

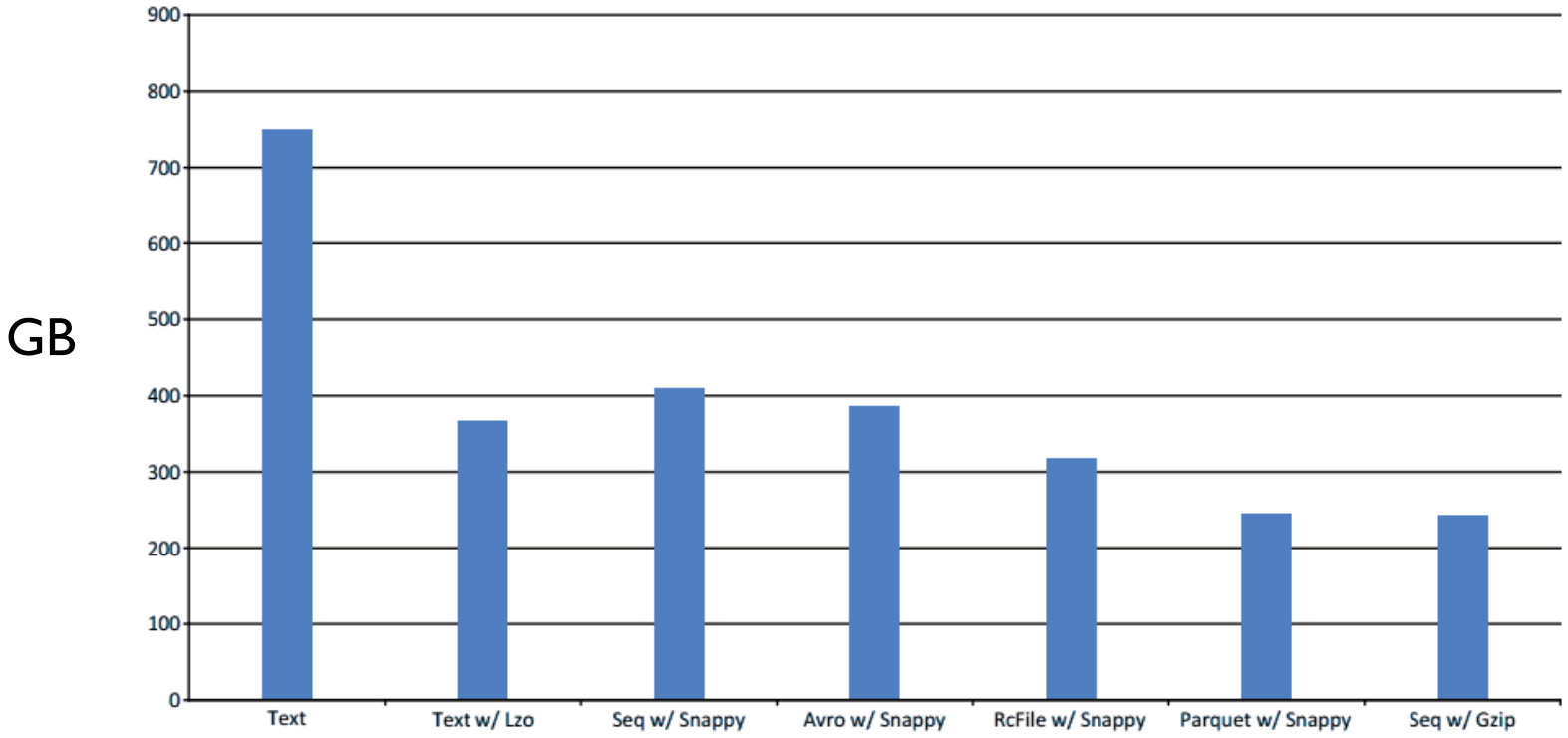
A1	A2	A3	B1	B2	B3	C1	C2	C3

PARQUET - Introduction

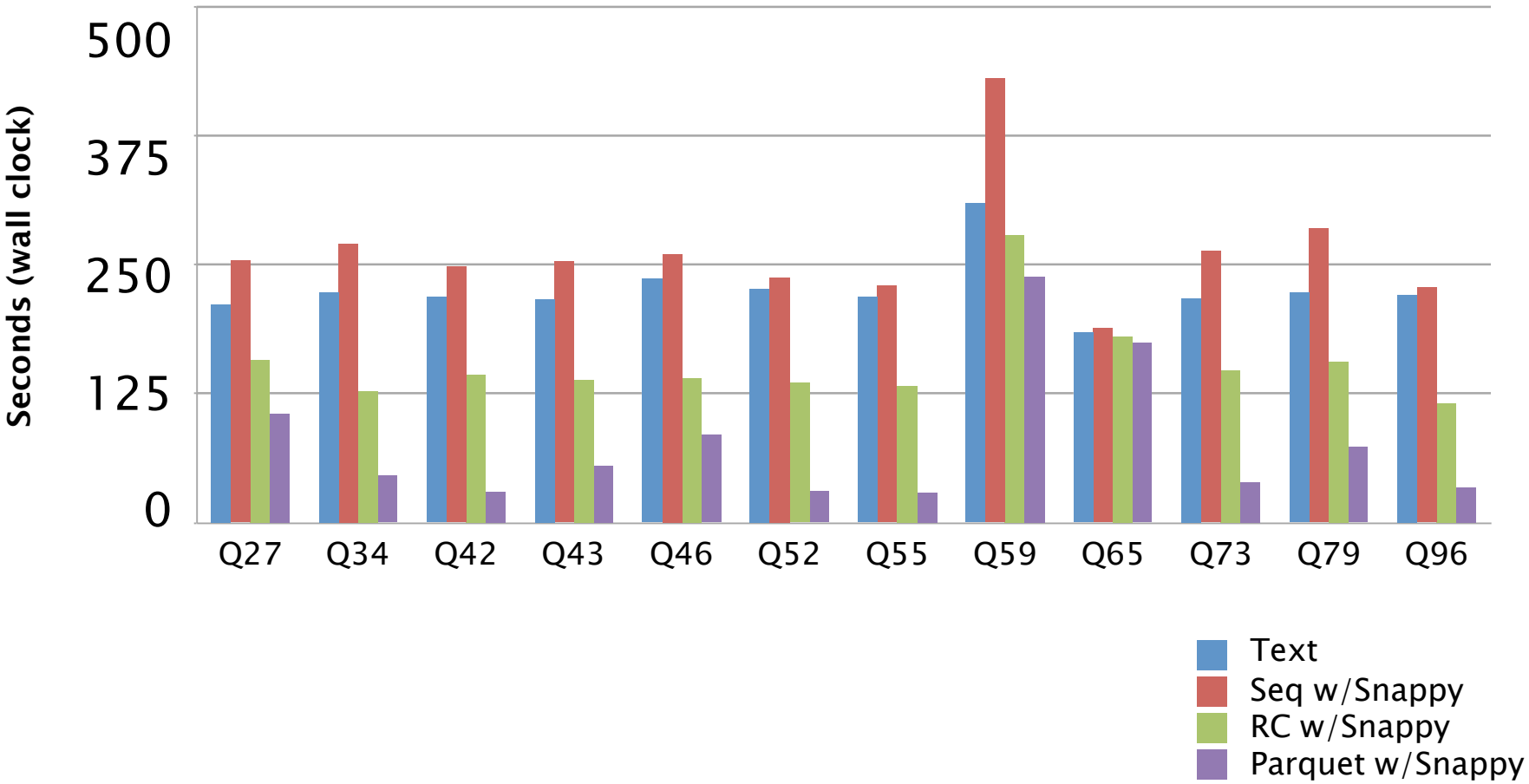
- Allows compression. Currently supports Snappy and Gzip.
- Well supported over Hadoop eco system.
- Very well integrated with Spark SQL and DataFrames.
- Predicate pushdown:
  Projection and predicate pushdowns involve an execution engine pushing the projection and predicates down to the storage format to optimize the operations at the lowest level possible.
- I/O to a minimum by reading from a disk only the data required for the query.
- Language independent. Supports Scala, Java, C++, Python.

Results in Impala

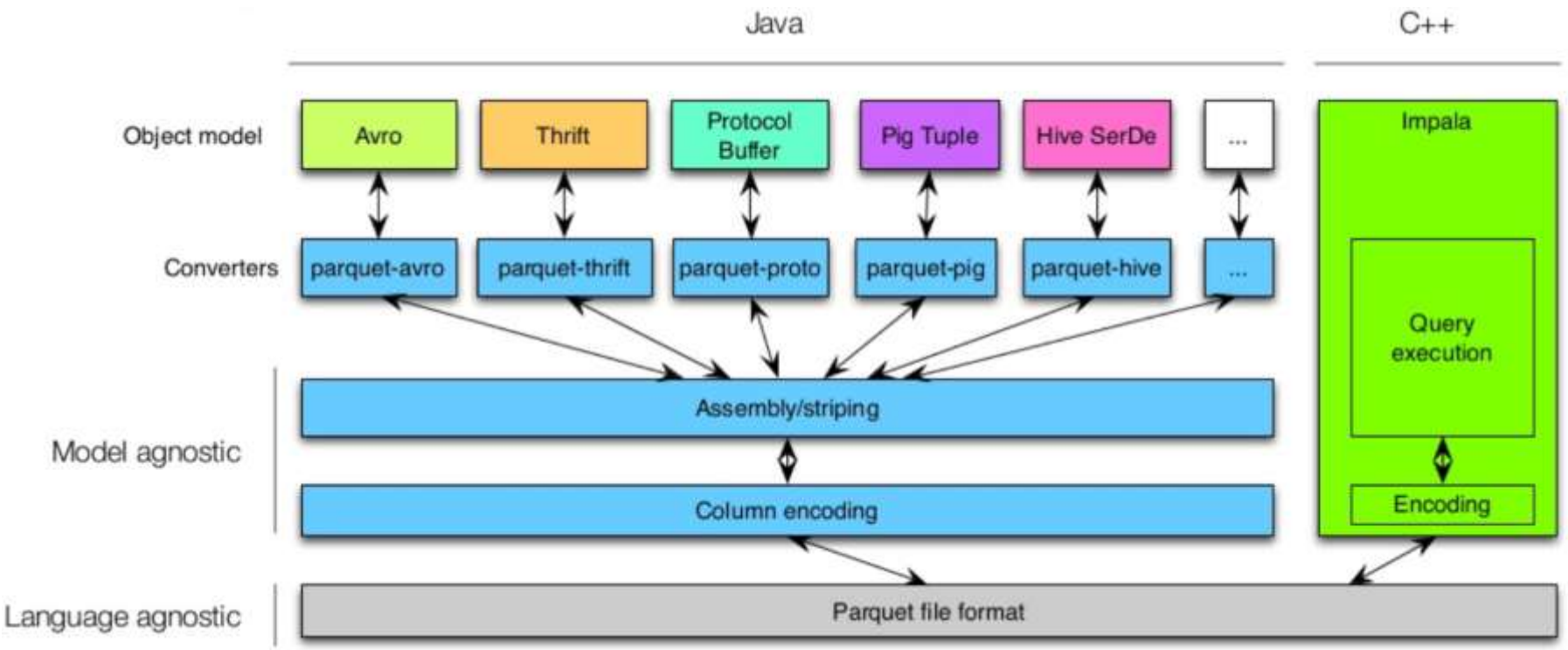
TPC-H lineitem table @ 1TB scale factor



Impala query times on TPC-DS

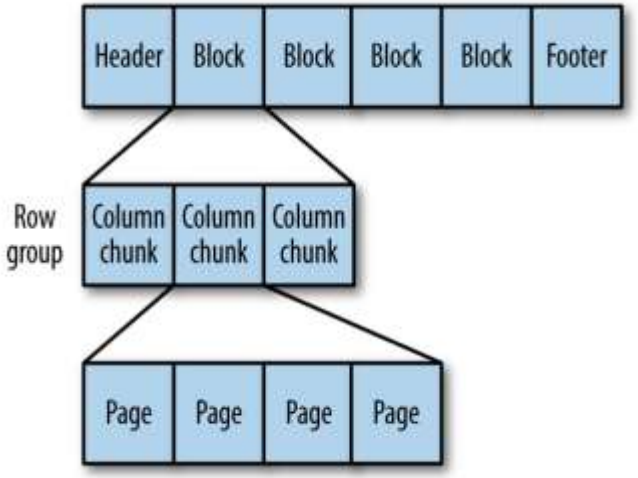


Interoperable



Format

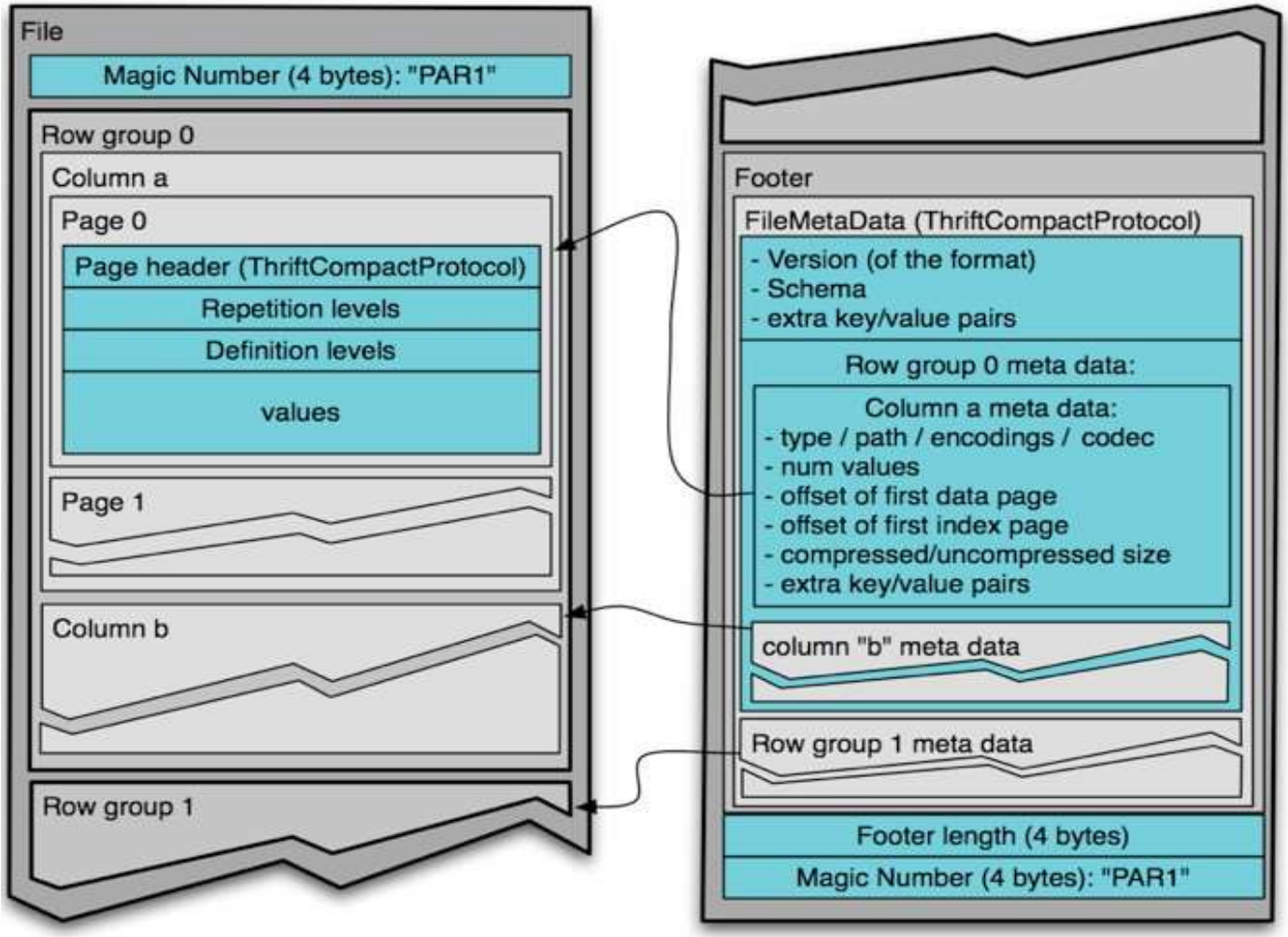
- Row group:
  A group of rows in columnar format.
  - Max size buffered in memory while writing.
  - One (or more) per split while reading.
  - roughly: 50MB < row group < 1 GB



- Column chunk:
  The data for one column in a row group.
  - Column chunks can be read independently for efficient scans.

- Page: Unit of access in a column chunk.
  - Should be big enough for compression to be efficient.
  - Minimum size to read to access a single record (when index pages are available). roughly: 8KB < page < 1MB

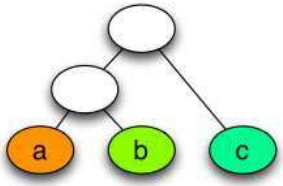
Format



Layout:

Row groups in columnar format. A footer contains column chunks offset and schema.

Format



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

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

encoding

encoded chunk	encoded chunk	encoded chunk

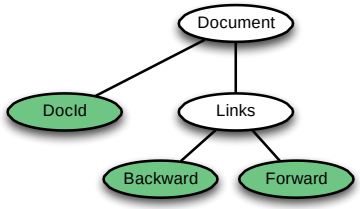
Image: http://de.slideshare.net/cloudera/hadoop-summit-36479635

Nested record shredding/assembly

- Algorithm borrowed from Google Dremel's column IO
- Each cell is encoded as a triplet:
  repetition level, definition level, value.
- Level values are bound by the depth of the schema:
  stored in a compact form.

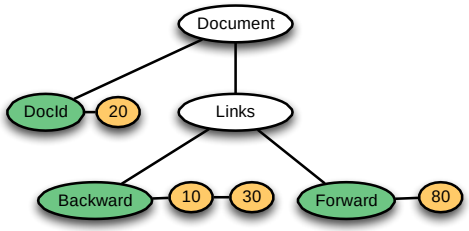
Schema:

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward;
  }
}
```



Record:

```
DocId: 20
Links
  Backward: 10
  Backward: 30
  Forward:  80
```



Columns	Max rep. level	Max def. level
DocId	0	0
Links.Backward	1	2
Links.Forward	1	2

Column	Value	R	D
DocId	20	0	0
Links.Backward	10	0	2
Links.Backward	30	1	2
Links.Forward	80	0	2

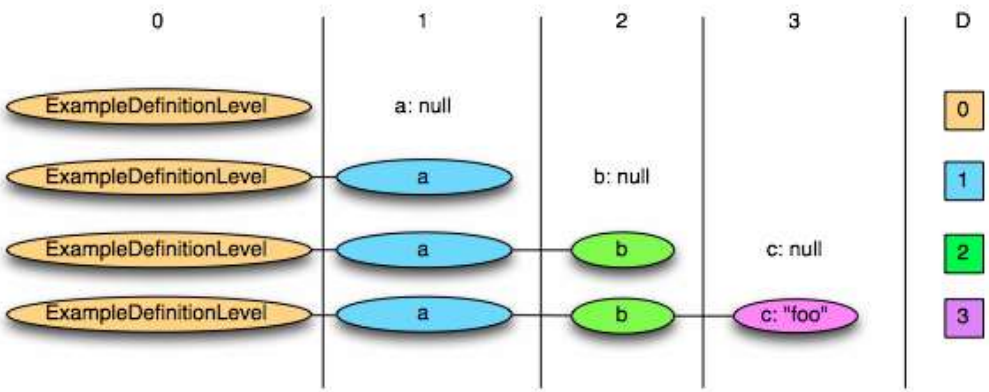
Definition Level

To support nested records we need to store the level for which the field is null.

Definition level: from 0 at the root of the schema up to the maximum level for this column.

```
message
ExampleDefinitionLevel {
  optional group a {
    optional group b {
      optional string c;
    }
   }
}
```

Value	Definition Level
a: null	0
a: { b: null }	1
a: { b: { c: null } }	2
a: { b: { c: "foo" } }	3 (actually defined)



Repetition Level

Repetition level is the level at which we have to create a new list for the current value.

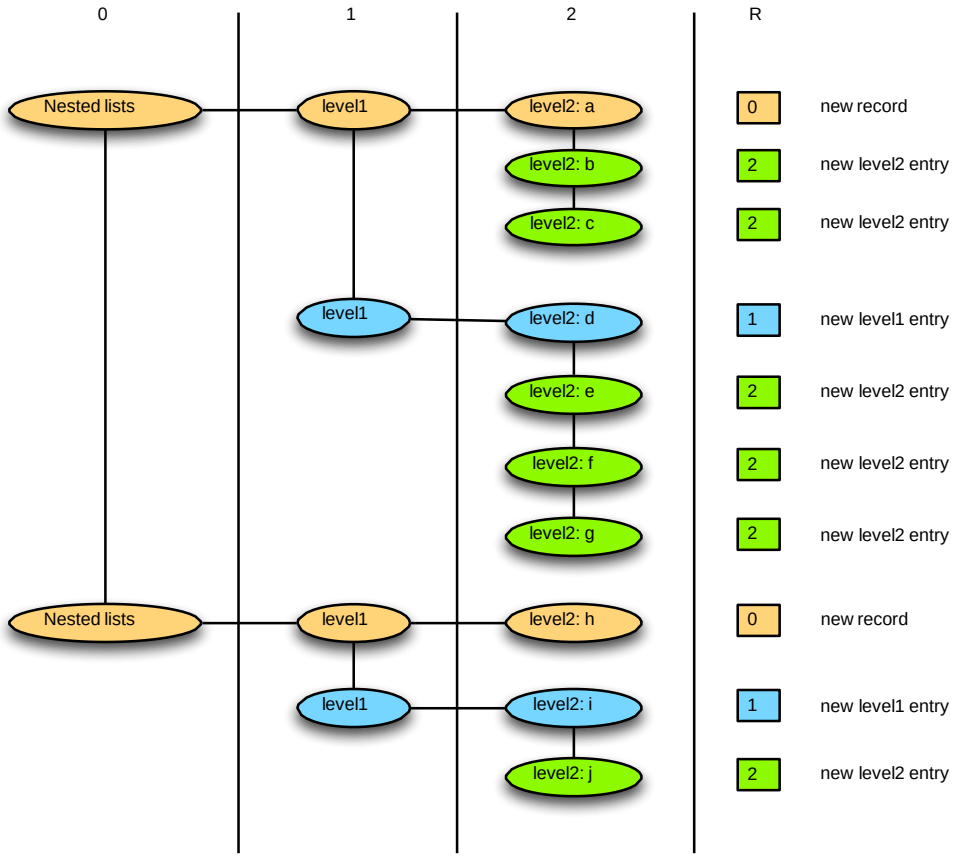
```
message nestedLists {
  repeated group level1 {
    repeated string level2;
  }
}
```

Records:

```
[[a,b,c],[d,e,f,g]],[[h],[i,j]]
```

Columns:

```
Level: 0,2,2,1,2,2,2,0,1,2
Data:  a,b,c,d,e,f,g,h,i,j
```



more details: https://blog.twitter.com/2013/dremel-made-simple-with-parquet

Example

Figure 2. Two sample nested records and their schema.

```
DocId: 10                          r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

```
DocId: 20          r2
Links
  Backward: 10
  Backward: 30
  Forward: 80
Name
  Url: 'http://C'
```

Figure 3. Column-striped representation of the data in Figure 2.

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

Figure 4. Repetition and definition levels: delta between paths.

Name.Language.Code

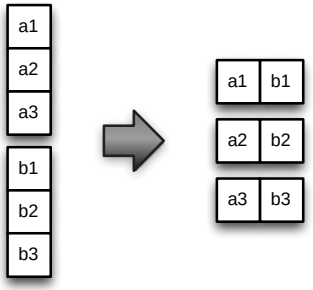
value	r	d
en-us	**0**	2
en	**2**	2
NULL	**1**	1
en-gb	**1**	2
NULL	**0**	1

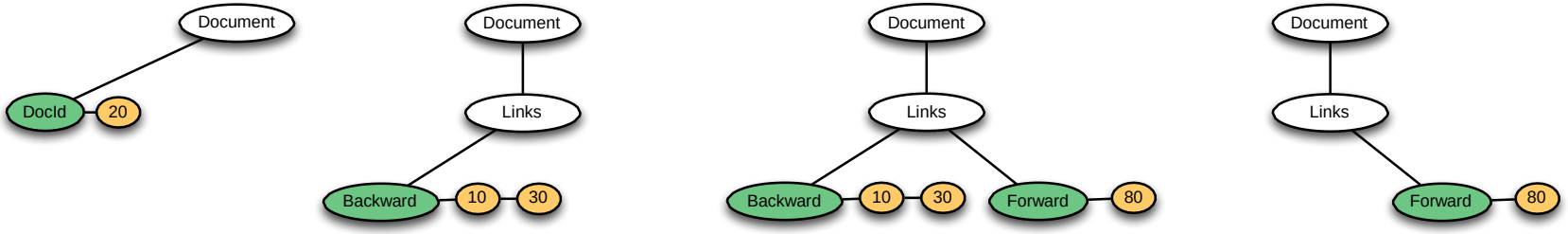
: common prefix

r_1.Name$_1$.Language$_1$.Code: **'en-us'**
r_1.Name$_1$.Language$_2$.Code: **'en'**
r_1.Name$_2$
r_1.Name$_3$.Language$_1$.Code: **'en-gb'**
r_2.Name$_1$

Reading assembled records

- Record level API to integrate with existing row based engines (Hive, Pig, M/R).
- Aware of dictionary encoding: enable optimizations.
- Assembles projection for any subset of the columns: only those are loaded from disc.





Projection push down

Automated in Pig and Hive:

Based on the query being executed only the columns for the fields accessed will be fetched.

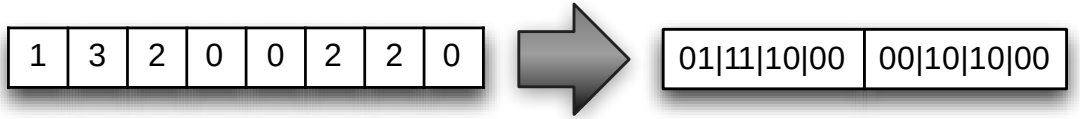
Explicit in MapReduce, Scalding and Cascading using globing syntax.

Example: field1;field2/**;field4/{subfield1,subfield2}
Will return:

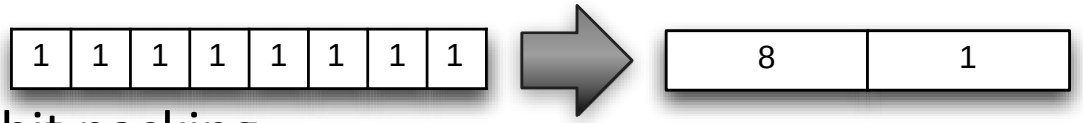
- field1
- all the columns under field2
- subfield1 and 2 under field4 but not field3

Encodings

- Bit packing:
  - Small integers encoded in the minimum bits required
  - Useful for repetition level, definition levels and dictionary keys



- Run Length Encoding:
  - Used in combination with bit packing
  - Cheap compression
  - Works well for definition level of sparse columns.



- Dictionary encoding:
  - Useful for columns with few (< 50,000) distinct values
  - When applicable, compresses better and faster than heavyweight algorithms (gzip, lzo, snappy)

Extensible: Defining new encodings is supported by the format

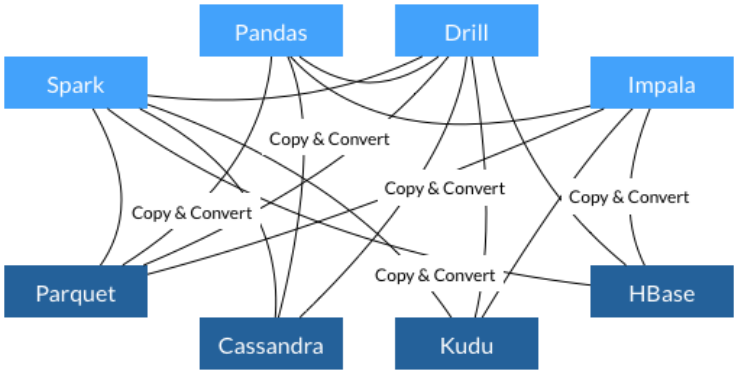
Using Parquet with Hive

```
CREATE TABLE order_details_parquet (
    order_id INT, prod_id INT)
    STORED AS PARQUET;
```

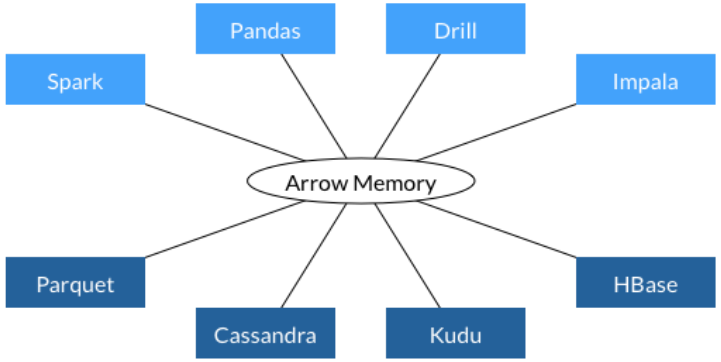
Apache Arrow

- Parquet: Common need for on disk columnar.
- Arrow: Common need for in memory columnar.
- Arrow is building on the success of Parquet.
- Top-level Apache project

Before



With Arrow



Outline

- Motivation
- Avro
- Parquet
- Optimized Row-Columnar - ORC
- Summary

Apache ORC – Optimized Row-Columnar File

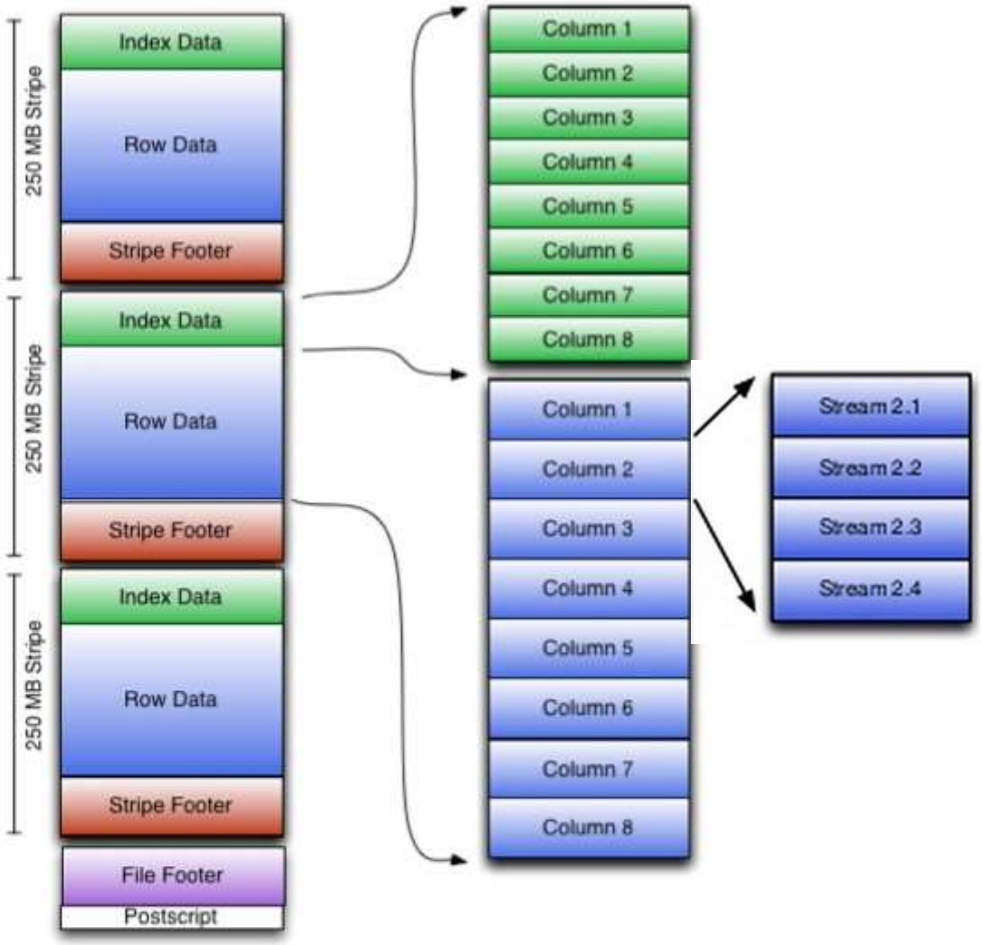
- Columnar format
- Enables user to read & decompress just the bytes they need
- Fast
- Indexed
- Self-describing
  - Includes all of the information about types and encoding
- Rich type system
  - All of Hive's types including timestamp, struct, map, list, and union

File Structure

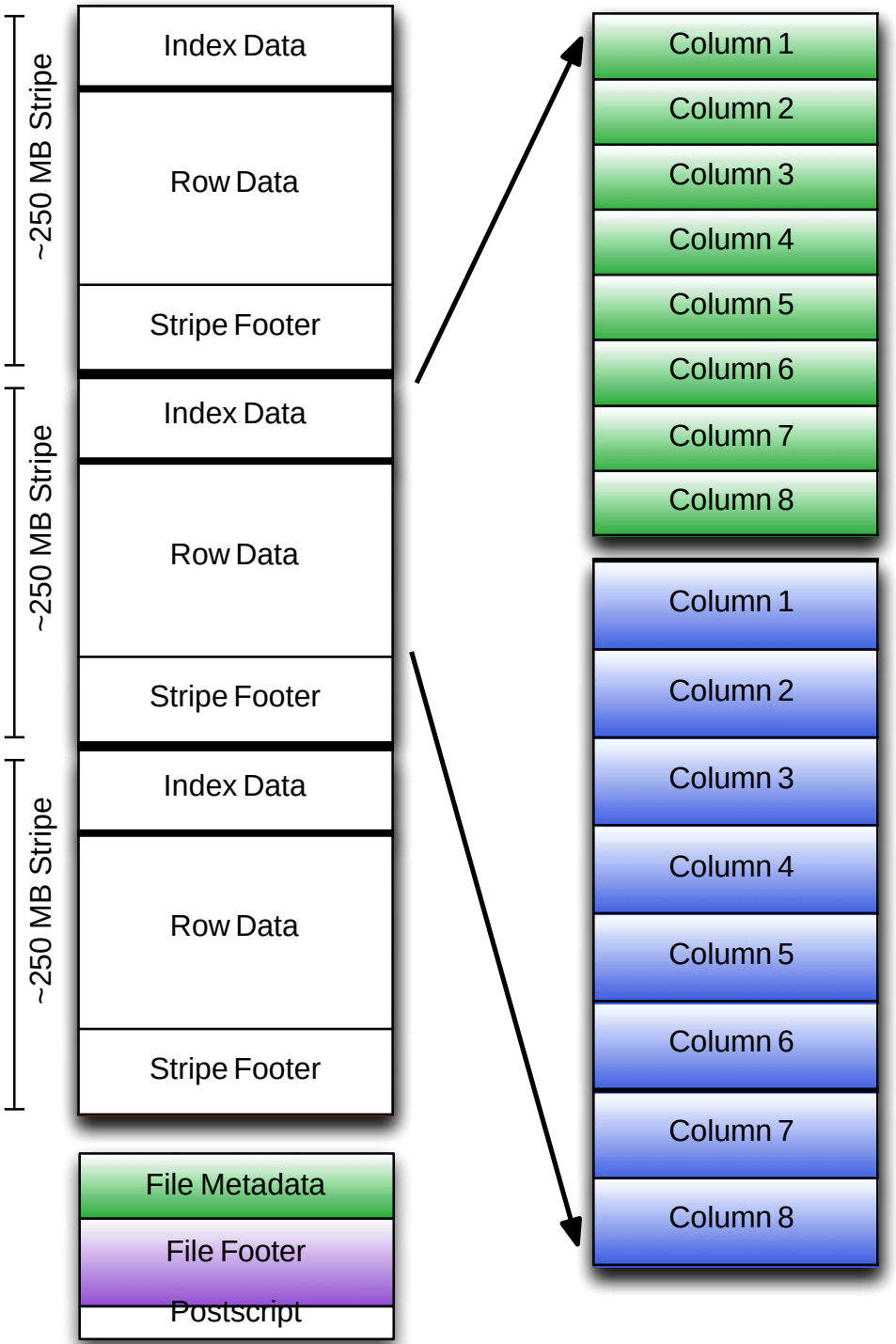
- File contains a list of stripes, which are sets of rows
  - Default size is 256MB
  - Large stripe size enables efficient reads
- Footer
  - Contains the list of stripe locations
  - Type description
  - File and stripe statistics
- Postscript
  - Compression parameters
  - File format version

File Structure: Stripes

- Stripes
  - Indexes
    Row group indexes and Bloom Filter interleaved
    - Min/Max stats, Positions for every 10K rows
  - Data
    - Multiple streams per column encoded and compressed independently
- Stripe Footer
  - Locations to streams, type of encoding



File Layout


~250 MB Stripe
Index Data
Row Data
Stripe Footer
~250 MB Stripe
Index Data
Row Data
Stripe Footer
~250 MB Stripe
Index Data
Row Data
Stripe Footer
File Metadata
File Footer
Postscript
Column 1
Column 2
Column 3
Column 4
Column 5
Column 6
Column 7
Column 8
Column 1
Column 2
Column 3
Column 4
Column 5
Column 6
Column 7
Column 8

Hive and Orc

```
create table my_table (name string,
address string) stored as orc;
```

```
create table Addresses (
name string, street string, city string, state
string, zip int)
stored as orc
tblproperties ("orc.compress"="SNAPPY");
```

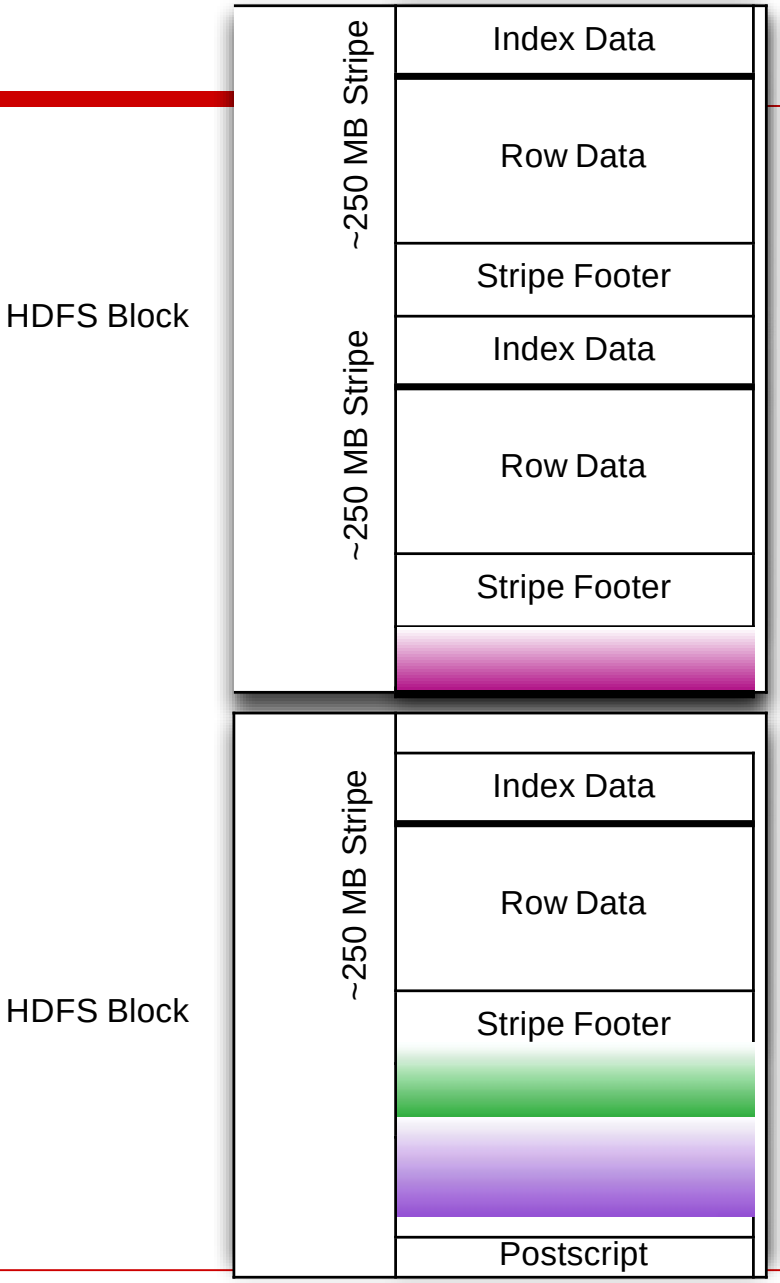
Performance

- Stripe size makes a huge difference in performance
- orc.stripe.size or hive.exec.orc.default.stripe.size
- Controls the amount of buffer in writer. Default is 250MB
- Trade off
  - Large stripes = Large more efficient reads
  - Small stripes = Less memory and more granular processing splits

- Multiple files written at the same time will shrink stripes
  - Use Hive's hive.optimize.sort.dynamic.partition
  - Sorting dynamic partitions means a one writer at a time

HDFS Block Padding

- The stripes don't align exactly with HDFS blocks
- HDFS scatters blocks around cluster
- Often want to pad to block boundaries
- Costs space, but improves performance
- hive.exec.orc.default.block.padding – true
- hive.exec.orc.block.padding.tolerance – 0.05



Predicate Push Down

- Reader is given a SearchArgument
  - Limited set predicates over column and literal value
  - Reader will skip over any parts of file that can’t contain valid rows

- ORC indexes at three levels:
  - File
  - Stripe
  - Row Group (10k rows)

- Reader still needs to apply predicate to filter out single rows

Row Pruning

- Every primitive column has minimum and maximum at each level
  - Sorting your data within a file helps a lot
  - Consider sorting instead of making lots of partitions
- Writer can optionally include bloomfilters
  - Provides a probabilistic bitmap of hashcodes
  - Only works with equality predicates at the row group level
  - Requires significant space in the file
  - Manually enabled by using orc.bloom.filter.columns
  - Use orc.bloom.filter.fpp to set the false positive rate (default 0.05)
  - Set the default charset in JVM via -Dfile.encoding=UTF-8

Row Pruning Performance Example

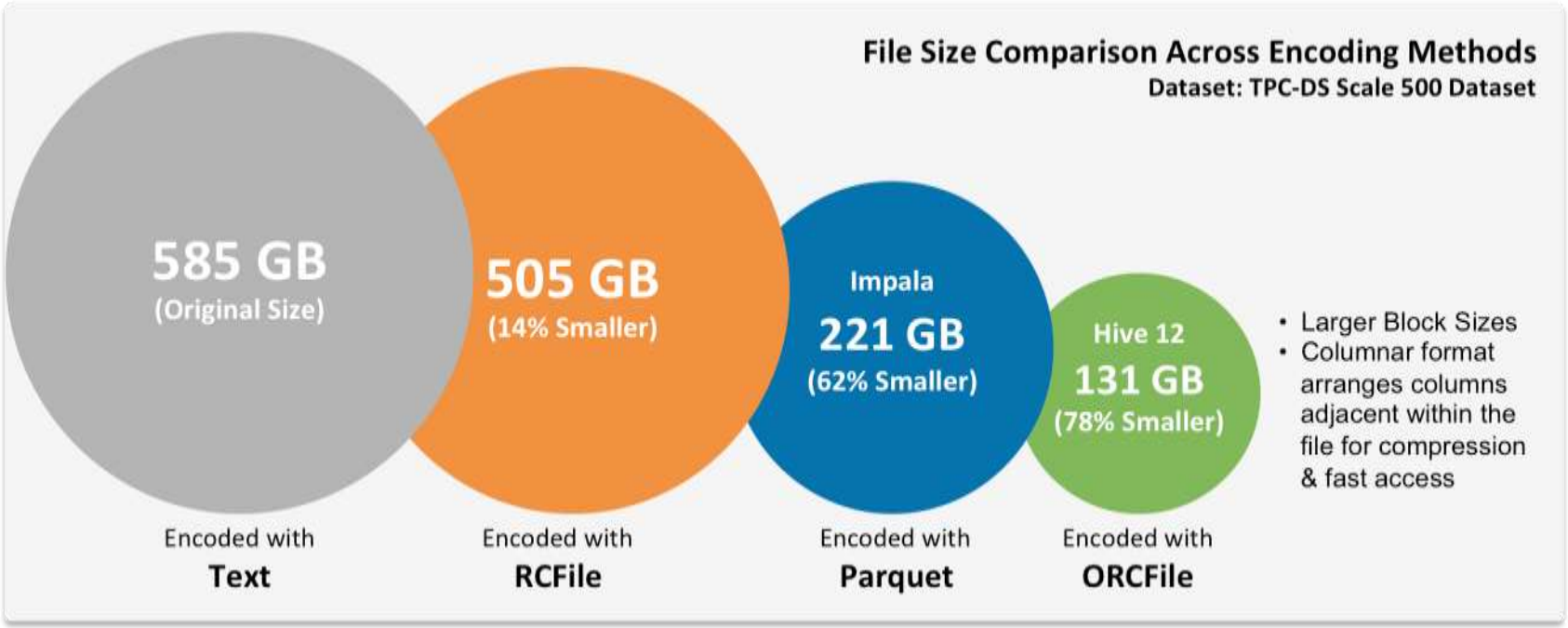
TPC-DS
from tpch1000.lineitem where l_orderkey = 1212000001;

- Rows Read
  - Nothing – 5,999,989,709
  - Min/Max – 540,000
  - BloomFilter – 10,000

- Time Taken
  - Nothing – 74 sec
  - Min/Max – 4.5 sec
  - BloomFilter – 1.3 sec

Summary

- Hadoop and its ecosystem support many file formats
- Selecting the format for your data set involves several considerations like tool compatibility, expected lifetime, storage and performance requirements
- Choose from the three main Hadoop file format options
  - Text – Good for testing and interoperability
  - Avro – Best for general purpose performance and evolving schemas
  - Parquet – Best performance for column-oriented access patterns
- Avro is a serialization framework that includes a data file format
  - Compact binary encodings provide good performance
  - Supports schema evolution for long-term storage



Source: http://hortonworks.com/blog/orcfile-in-hdp-2-better-compression-better-performance/

ORC Use Cases

Facebook

Compression
Saved more than 1,400 servers worth of storage.[2]

Compression
Compression ratio increased from 5x to 8x globally.[2]

Spotify®

CPU
32x less CPU when using ORC versus Avro.[3]

IO
16x less HDFS read when using ORC versus Avro.[3]

https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/

http://www.slideshare.net/AdamKawa/a-perfect-hive-query-for-a-perfect-meeting-hadoop-summit-2014