Parquet

Columnar Storage for the People

Lars George
EMEA Chief Architect @ Cloudera
lars@cloudera.com

About Me

- EMEA Chief Architect at Cloudera
- Apache Committer
  - HBase and Whirr
- O'Reilly Author
  - HBase – The Definitive Guide
    - Now in Japanese!
- Contact
  - lars@cloudera.com
  - @larsgeorge

日本語版も出ました!

Introduction

For **analytical** workloads it is often advantageous to store the data in a **layout** that is more amenable to the way it is accessed.

Parquet is an **open-source** file **format** that strives to do exactly that, i.e. provide an **efficient** layout for analytical queries.

We will be looking some **context** from various **companies**, the results observed in production and benchmarks, and finally do a bit of a format **deep-dive**.

Example: Twitter

- Twitter's **Data**
  - **230M+** monthly active users generating and consuming **500M+** tweets a day
  - **100TB+** a day of compressed data
  - Huge scale for instrumentation, user graph, derived data, etc.
- **Analytics** Infrastructure
  - Several **1K+** node Hadoop clusters
  - Log Collection Pipeline
  - Processing Tools

Example: Twitter

Twitter's Use-case

- Logs available on HDFS
- Thrift to store logs
- Example schema: 87 columns, up to 7 levels of nesting

```
struct LogEvent {
  1: optional logbase.LogBase log_base
  2: optional i64 event_value
  3: optional string context
  4: optional string referring_event
...
  18: optional EventNamespace event_namespace
  19: optional list<Item> items
  20: optional map<AssociationType,Association> associations
  21: optional MobileDetails mobile_details
  22: optional WidgetDetails widget_details
  23: optional map<ExternalService,string> external_ids
}
```

```
struct LogBase {
  1: string transaction_id,
  2: string ip_address,
  ...
  15: optional string country,
  16: optional string pid,
}
```

Example: Twitter

Goal:

"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 I/O heavy
- Incrementally take advantage of column based storage in existing framework
- Not tied to any framework in particular

Columnar Storage

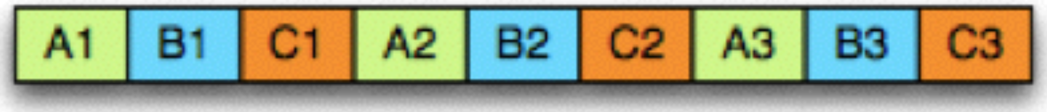
- **Limits** I/O to data actually needed
  - Loads **only** the columns that need to be accessed
- **Saves** space
  - Columnar layout compresses **better**
  - Type **specific** encodings
- Enables vectorized execution engines

Columnar vs Row-based

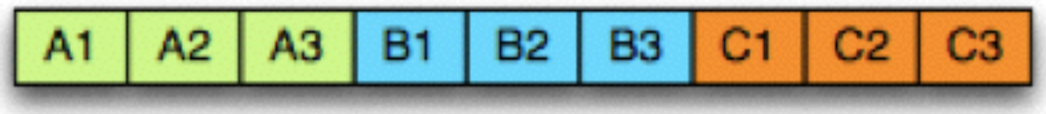
Here is an example of translating a logical table schema. First the example table:

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

In a row-based layout each row follows the next:



While for a column-oriented layout it stores one column after the next:



Parquet Intro

Parquet defines a common file format, which is

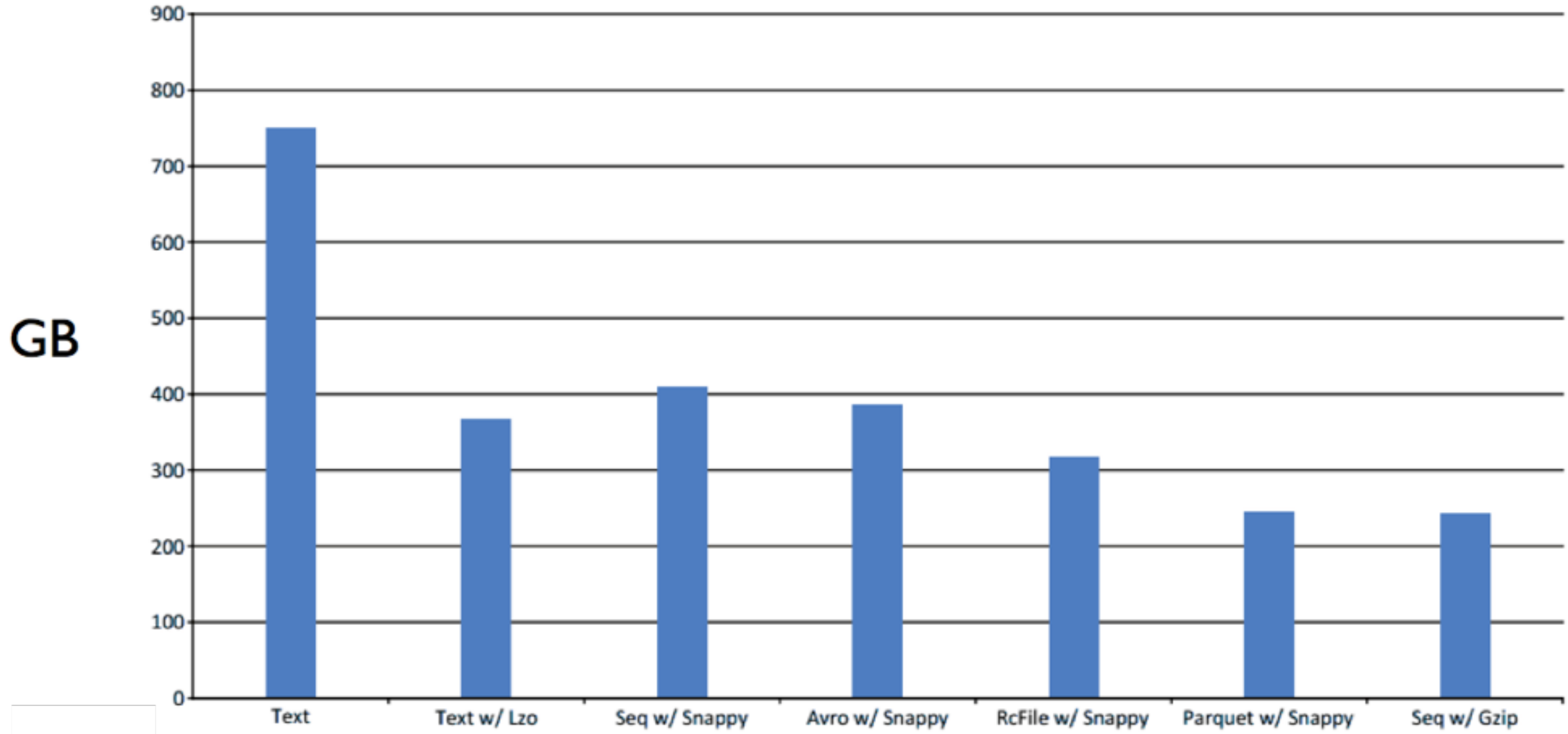
language independent and

formally specified.

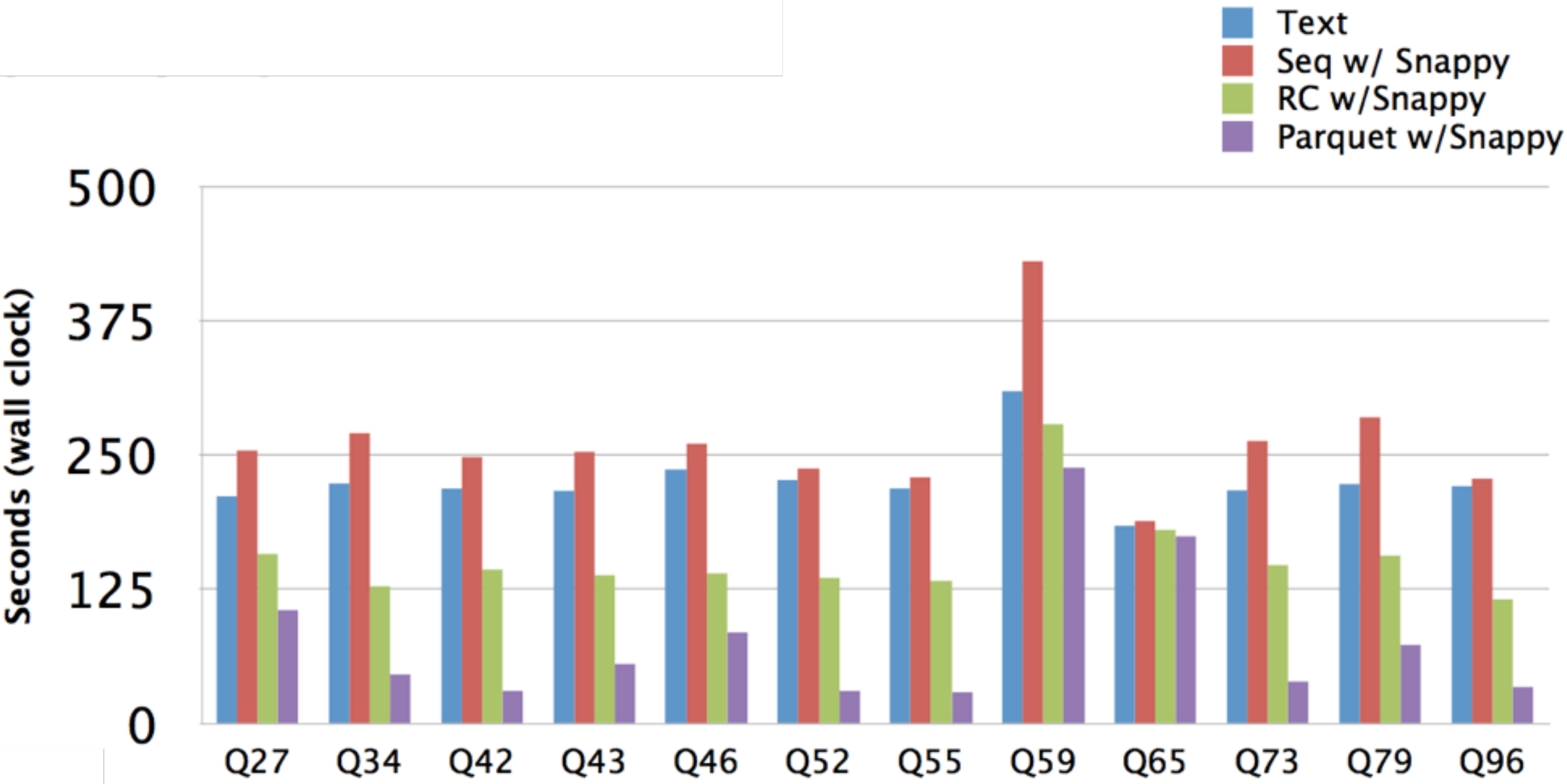
Implementation exist in Java for MapReduce and C++, which is used by Impala.

Example: Impala Results

- TPC-H lineitem table @ 1TB scale factor



Example: Impala TPC-DS



Example: Criteo

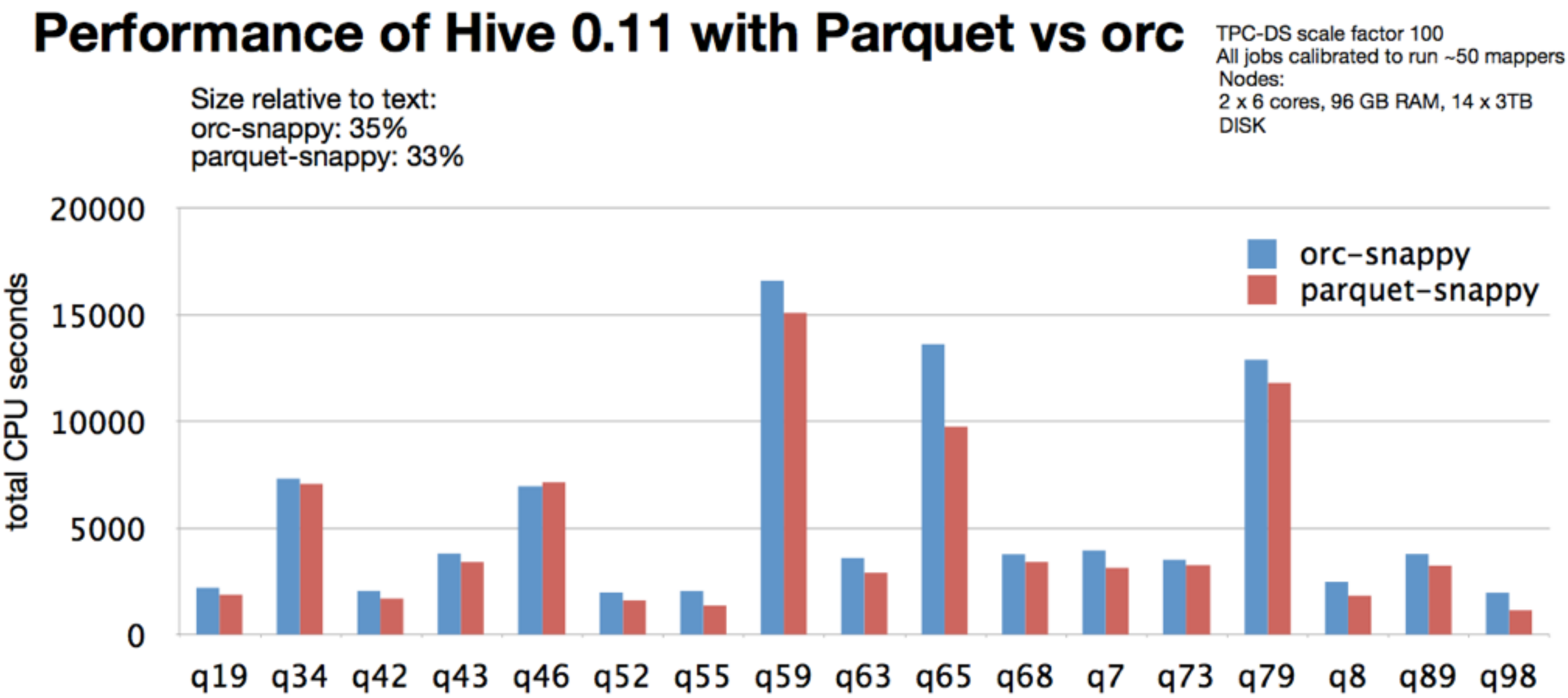
- **Billions** of new events per day
- Roughly **60** columns per log
- **Heavy** analytic workload
- **BI** analysts using Hive and RCFile
- **Frequent** schema modifications

- **Perfect** use case for Parquet + Hive!

Parquet + Hive: Basic Requirements

- MapReduce **compatibility** due to Hive
- Correctly handle **evolving** schemas across Parquet files
- Read **only** the columns use by query to minimize data read
- **Interoperability** with other execution engines, for example Pig, Impala, etc.

Performance



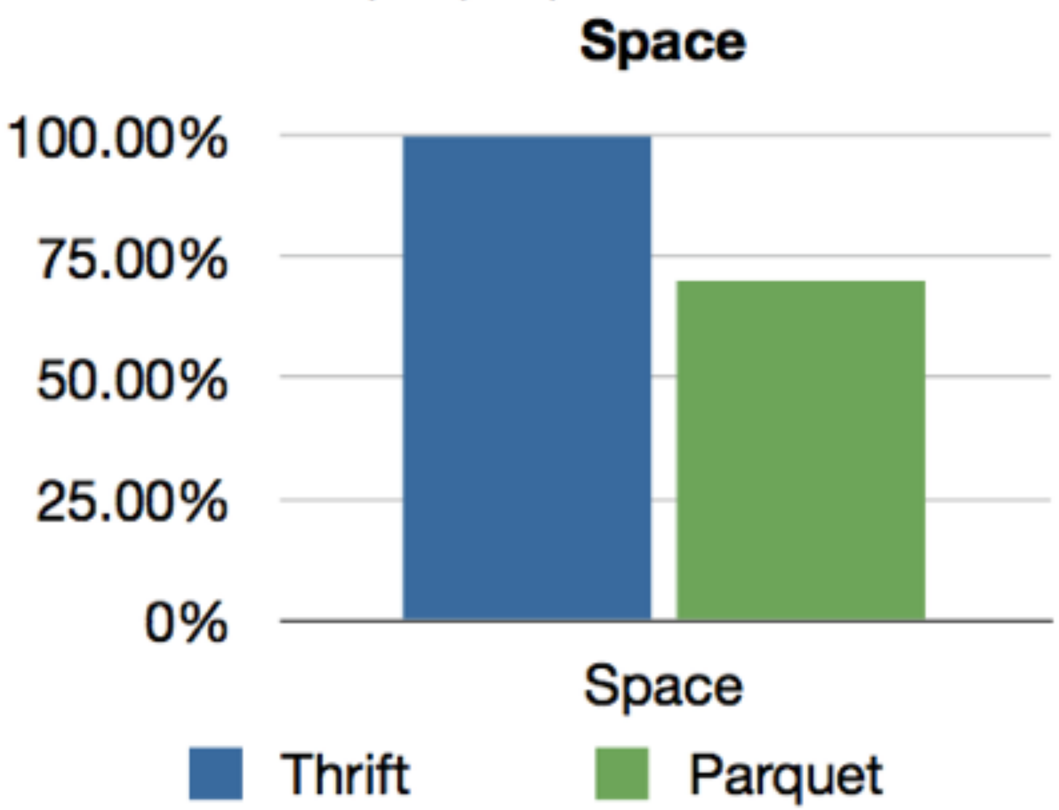
Performance

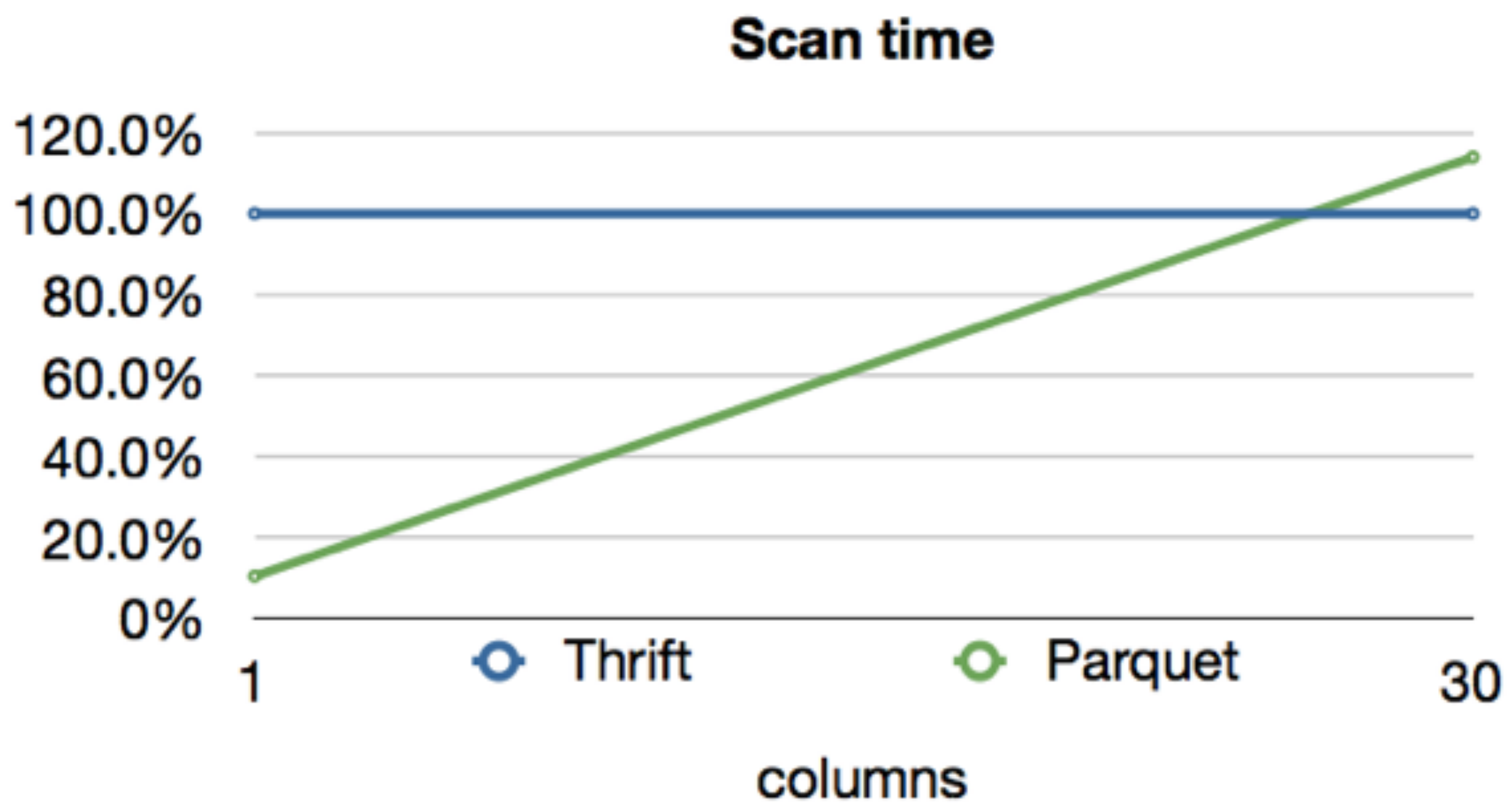
Twitter: production results

Data converted: similar to access logs. 30 columns.

Original format: Thrift binary in block compressed files (LZO)

New format: Parquet (LZO)





- **Space saving:** 30% using the same compression algorithm
- **Scan + assembly time compared to original:**
  - One column: 10%
  - All columns: 110%

Example: Twitter

- **Petabytes** of storage saved
- Example jobs taking **advantage** of projection push down:
  - Job 1 (Pig): reading **32%** less data -> **20%** task time saving
  - Job 2 (Scalding): reading 14 out of 35 columns, reading **80%** less data -> **66%** task time saving
  - **Terabytes** of scanning saved every day

Parquet Model

- The algorithm is **borrowed** from Google Dremel's **ColumnIO** file format
- Schema is defined in a **familiar** format
- Supports **nested** data structures
- Each cell is **encoded** as triplet: repetition level, definition level, and the value
- Level values are **bound** by the depth of the schema
  - Stored in a **compact** form

Parquet Model

- Schema **similar** to Protocol Buffers, but with simplifications (e.g. no Maps, Lists or Sets)
    - These complex types can be expressed as a combination of the other features
- Root of schema is a group of fields called a **message**
- Field **types** are either **group** or **primitive** type with **repetition** of required, optional or repeated
    - exactly one, zero or one, or zero or more

Example Schema

```
message AddressBook {
  required string owner;
  repeated string ownerPhoneNumbers;
  repeated group contacts {
    required string name;
    optional string phoneNumber;
  }
}
```

Represent Lists/Sets

Schema: List of Strings	**Data:** `[ "a", "b", "c", ...]`

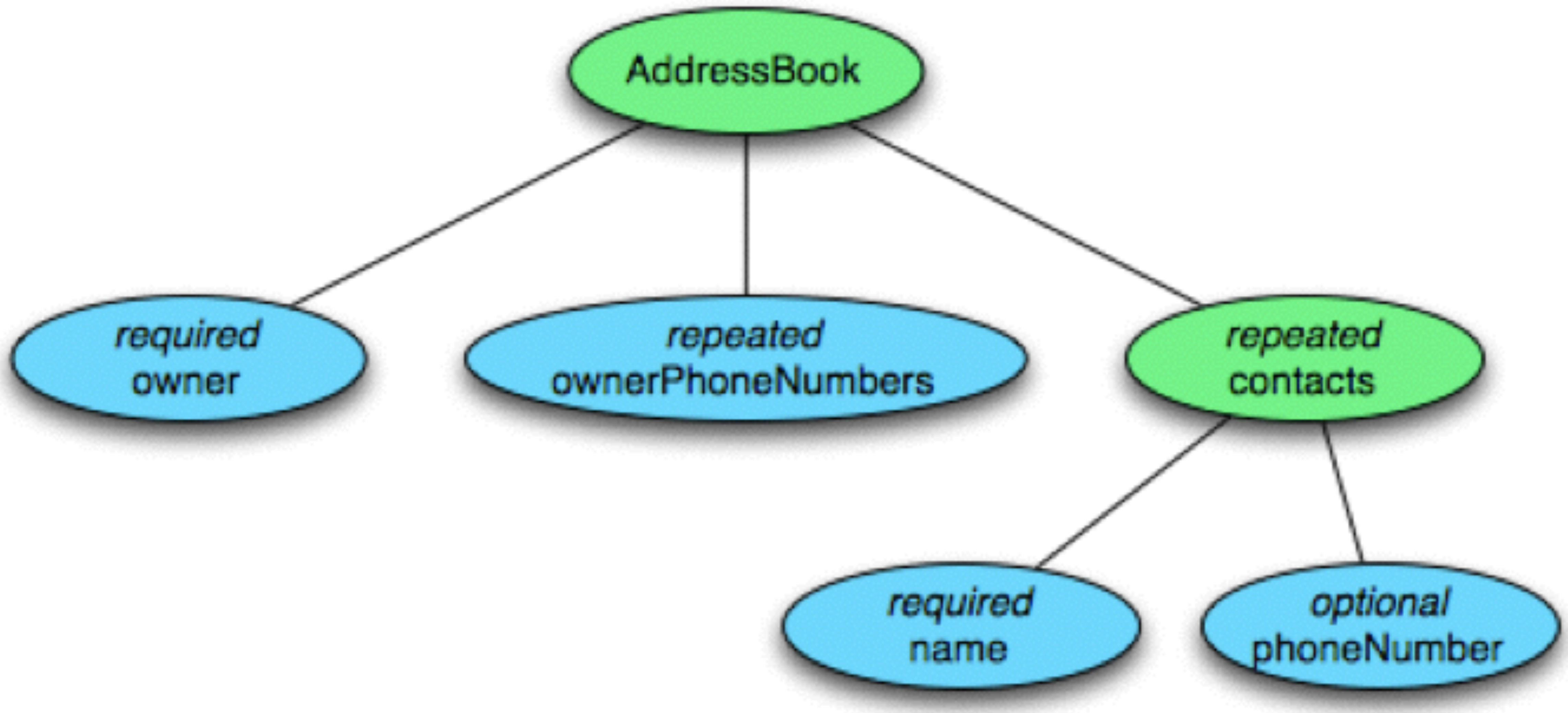
`message ExampleList {`
`     repeated string list;`
`}`
	`{`
	`     list: "a",`
	`     list: "b",`
	`     list: "c",`
	`     …`
	`}`

Representing Maps

Schema: Map of strings to strings	**Data:** `{"AL" => "Alabama", ...}`

<pre>message ExampleMap {
     repeated group map {
        required string key;
        optional string value;
     }
}</pre>
	<pre>{
	     map: {
	          key: "AL",
	          value: "Alabama"
	     },
	     map: {
	          key: "AK",
	          value: "Alaska"
	     },
	     …
	}</pre>

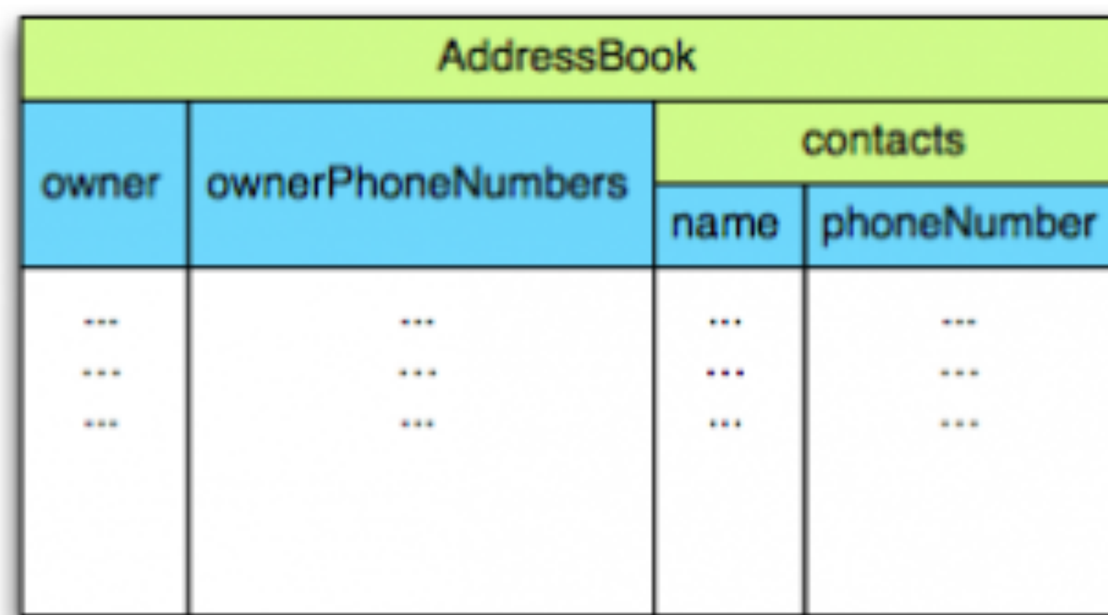
Schema as a Tree



Field per Primitive

Primitive fields are mapped to the columns in the columnar format, shown in blue here:

Column	Type
`owner`	string
`ownerPhoneNumbers`	string
`contacts.name`	string
`contacts.phoneNumber`	string



Levels

The **structure** of the record is captured for each value by **two** integers called **repetition** level and **definition** level.

Using these two levels we can fully **reconstruct** the nested structures while still being able to store each primitive **separately**.

Definition Levels

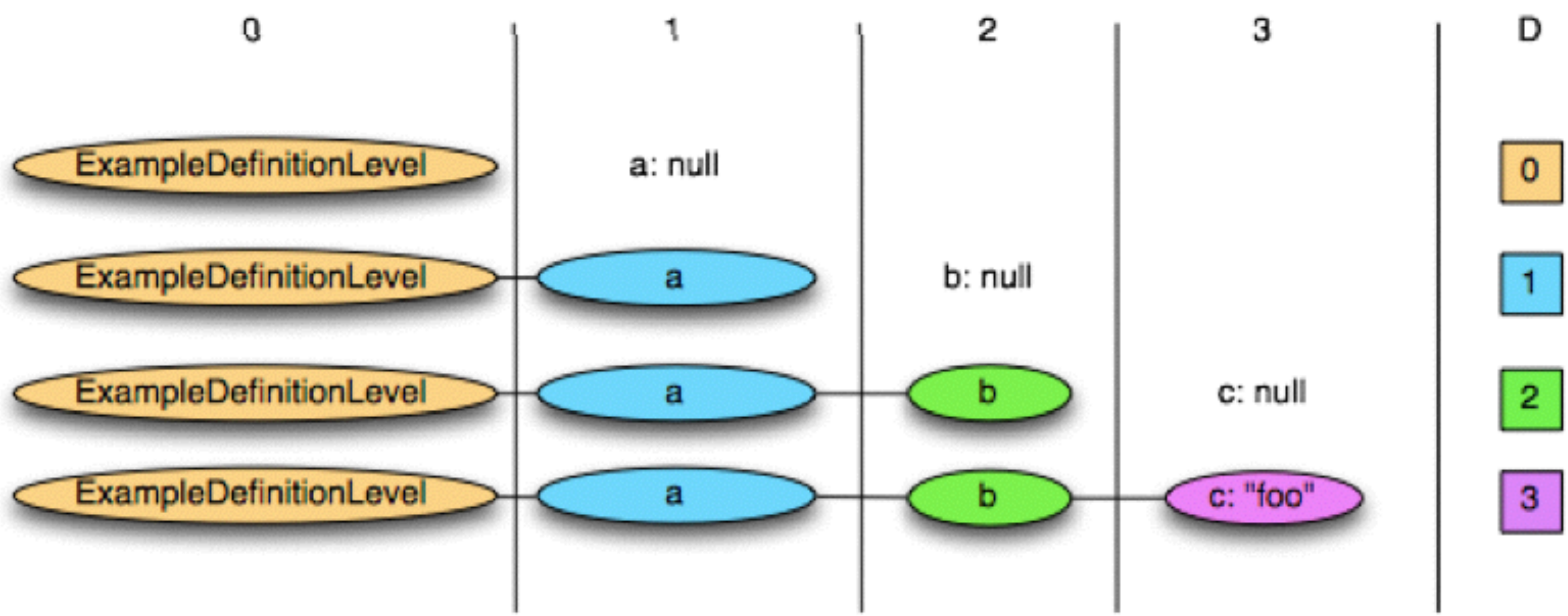
Example:

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

Contains one column “a.b.c” where all fields are optional and can be null.

Definition Levels

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



Definition Levels

Example with a required field:

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

Value	Definition Level
`a: null`	0
`a: { b: null }`	Impossible, as `b` is required
`a: { b: { c: null } }`	1
`a: { b: { c: “foo” } }`	2 (actually defined)

Repetition Levels

Repeated fields **require** that we store where a lists **starts** in a column of values, since these are stored **sequentially** in the same place. The repetition level **denotes** per value where a new lists starts, and are basically a **marker** which also indicates the level where to start the new list.

Only levels that are **repeated** need a repetition level, i.e. optional or required fields are never repeated and can be **skipped** while attributing repetition levels.

Repetition Levels

Schema:	**Data:** [[a,b,c],[d,e,f,g]],[[h],[i,j]]

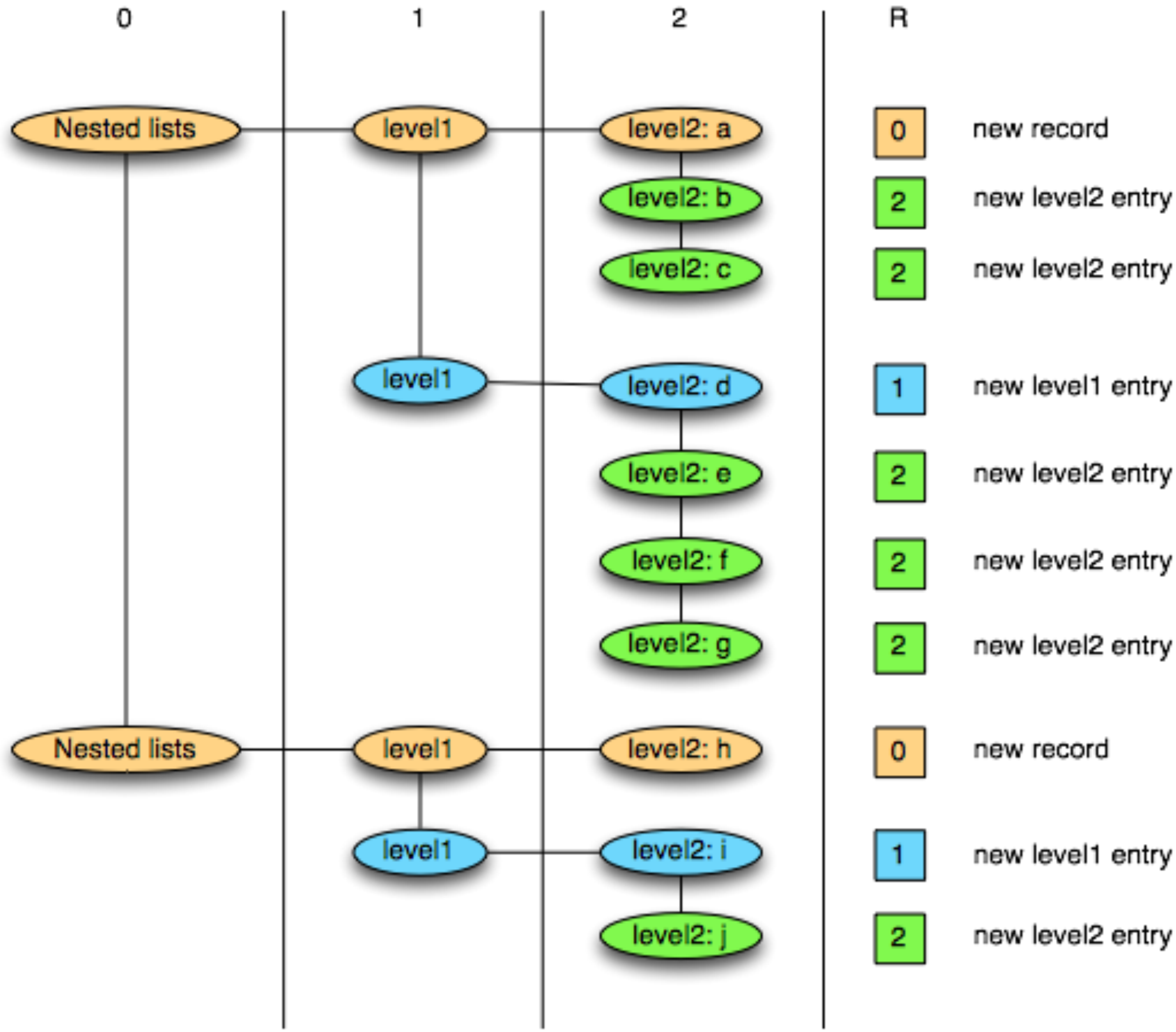
<pre>message nestedLists {
   repeated group level1 {
      repeated string level2;
   }
}</pre>
	<pre>{
	     level1: {
	          level2: a
	          level2: b
	          level2: c
	     },
	     level1: {
	          level2: d
	          level2: e
	          level2: f
	          level2: g
	     }
	}
	{
	     level1: {
	          level2: h
	     },
	     level1: {
	          level2: i
	          level2: j
	     }
	}</pre>

Repetition Levels

Repetition level	Value
0	a
2	b
2	c
1	d
2	e
2	f
2	g
0	h
1	i
2	j

- 0 marks every new **record** and **implies** creating a new **level1** and **level2** list
- 1 marks every new **level1** list and **implies** creating a new **level2** list as well
- 2 marks every new **element** in a **level2** list

Repetition Levels



Combining the Levels

Applying the two to the AddressBook example:

Column	Max Definition level	Max Repetition level
`owner`	0 (`owner` is *required*)	0 (no repetition)
`ownerPhoneNumbers`	1	1 (*repeated*)
`contacts.name`	1 (`name` is *required*)	1 (`contacts` is *repeated*)
`contacts.phoneNumber`	2 (`phoneNumber` is *optional*)	1 (`contacts` is *repeated*)

In particular for the column "contacts.phoneNumber", a defined phone number will have the maximum definition level of 2, and a contact without phone number will have a definition level of 1. In the case where contacts are absent, it will be 0.

Example: AddressBook

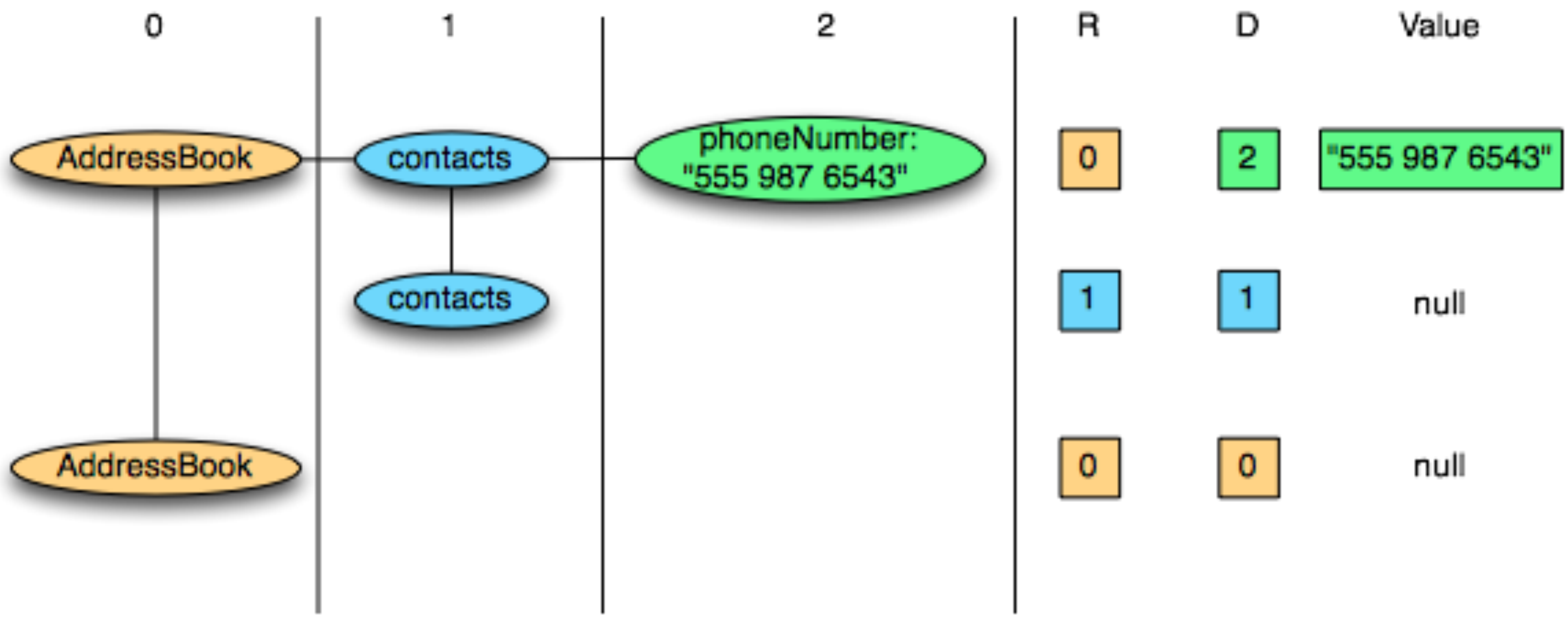
```
AddressBook {
  owner: "Julien Le Dem",
  ownerPhoneNumbers: "555 123 4567",
  ownerPhoneNumbers: "555 666 1337",
  contacts: {
    name: "Dmitriy Ryaboy",
    phoneNumber: "555 987 6543",
  },
  contacts: {
    name: "Chris Aniszczyk"
  }
}
AddressBook {
  owner: "A. Nonymous"
}
```

Looking at contacts.phoneNumber

Example: AddressBook

```
AddressBook {
  contacts: {
    phoneNumber: "555 987 6543"
  }
  contacts: {
  }
}
AddressBook {
}
```

Example: AddressBook



Example: AddressBook

When writing:

- contacts.phoneNumber: “555 987 6543”
  - new record: R = 0
  - value defined: D = max (2)
- contacts.phoneNumber: NULL
  - repeated contacts: R = 1
  - only defined up to contacts: D = 1
- contacts: NULL
  - new record: R = 0
  - only defined up to AddressBook: D = 0

R	D	Value
0	2	"555 987 6543"
1	1	NULL
0	0	NULL

Example: AddressBook

During reading

- **R=0, D=2, Value = “555 987 6543”:**
  - R = 0 means a new record. We recreate the nested records from the root until the definition level (here 2)
  - D = 2 which is the maximum. The value is defined and is inserted.
- **R=1, D=1:**
  - R = 1 means a new entry in the contacts list at level 1.
  - D = 1 means contacts is defined but not phoneNumber, so we just create an empty contacts.
- **R=0, D=0:**
  - R = 0 means a new record. we create the nested records from the root until the definition level
  - D = 0 => contacts is actually null, so we only have an empty AddressBook

Example: AddressBook

```
AddressBook {
  contacts: {
    phoneNumber: "555 987 6543"
  }
  contacts: {
  }
}
AddressBook {
}
```

Storing Levels

Each primitive type has **three** sub columns, though the overhead is **low** thanks to the columnar representation and the fact that values are **bound** by the **depth** of the schema, resulting in only a few bits used.

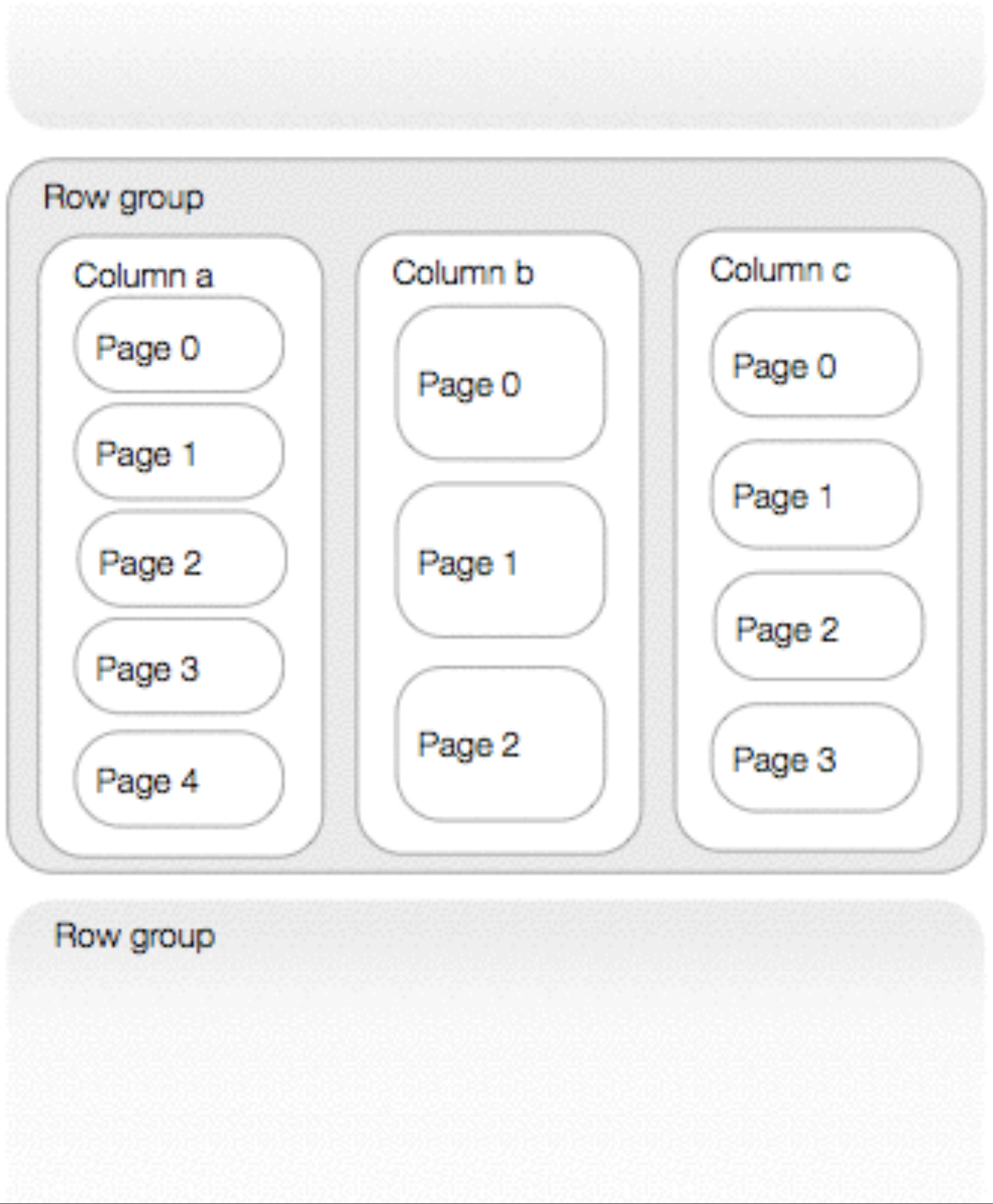
When all fields are required in a flat schema we can **omit** the levels altogether since they are would always be **zero**.

Otherwise compression, such as RLE, takes care of condensing data efficiently.

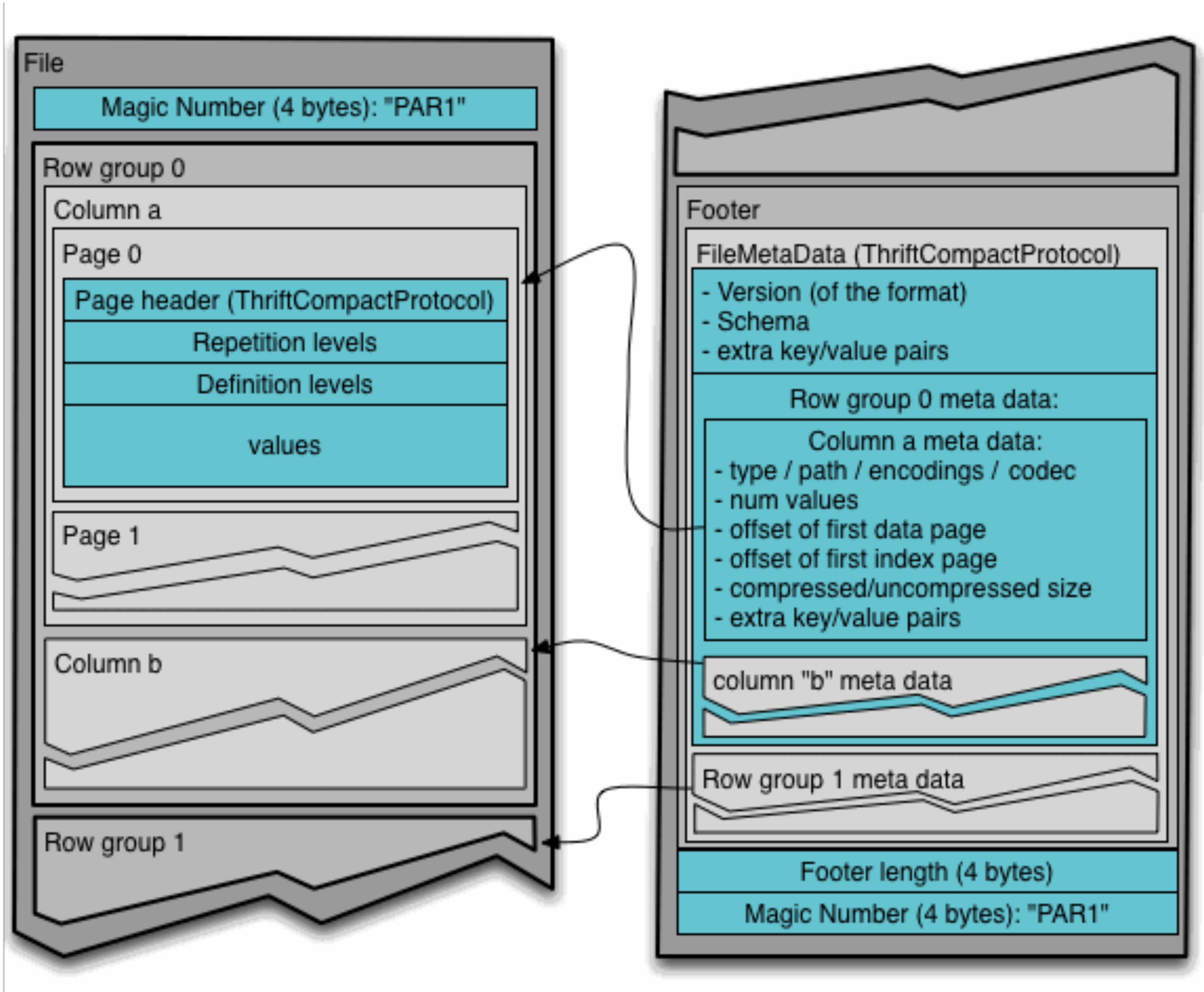
File Format

- Row Groups: A group of rows in columnar format
  - Max size buffered in memory while writing
  - One (or more) per split while reading
  - About 50MB < row group < 1GB
- Columns Chunk: Data for one column in row group
  - Column chunks can be read independently for efficient scans
- Page: Unit of access in a column chunk
  - Should be big enough for efficient compression
  - Min size to read while accessing a single record
  - About 8KB < page < 1MB

File Format


Row group
Column a
Page 0
Page 1
Page 2
Page 3
Page 4
Column b
Page 0
Page 1
Page 2
Column c
Page 0
Page 1
Page 2
Page 3
Row group

File Format



- Layout
  - Row groups in columnar format
  - footer contains column chunks offset and schema
- Language independent
  - Well defined format
  - Hadop and Impala support

Integration

Hive and Pig natively support projection push-down. Based on the query executed only the columns for the fields accessed are fetched.

MapReduce and other tools use a globbing syntax, for example:

```
field1;field2/**;field4{subfield1,subfield2}
```

This 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 (RLE)
  - Used in combination with bit packing
  - Cheap compression
  - Works well for definition level of sparse columns

Encodings

…continued:

- Dictionary Encoding
  - Useful for columns with few (<50k) distinct values
  - When applicable, compresses better and faster than heavyweight algorithms (e.g. gzip, lzo, snappy)
- Extensible
  - Defining new encodings is supported by the format

Future

- Parquet 2.0
  - More encodings
    - Delta encodings, improved encodings
  - Statistics
    - For query planners and predicate pushdown
  - New page format
    - skip ahead better

Questions?

- Contact: @larsgeorge, lars@cloudera.com
- Sources
  - Parquet Sources: https://github.com/parquet/parquet-format
  - Blog Post with Info: https://blog.twitter.com/2013/dremel-made-simple-with-parquet
  - Impala Source: https://github.com/cloudera/impala
  - Impala: http://www.cloudera.com/content/cloudera/en/campaign/introducing-impala.html