Storage and Indexing

CELEBRATING 30 YEARS
Marlan and Rosemary Bourns
College of Engineering

Overview

- We covered storage of unstructured files in HDFS
  - Partition into blocks
  - Replicate to data nodes
- This lecture will cover the storage of structured and semi-structured data
  - Row vs column formats
  - Data-aware partitioning
  - Indexing in big data
  - Big-data-specific file formats

Challenges

- Big-data applications typically scan a very large file
- In-situ processing, i.e., no separate data ingestion process
- Need to work efficiently with raw files in common formats

Row-oriented Stores

Row

Field 1 | Field 2 | Field 3 | …

- CSV and JSON formats are examples of traditional row-oriented data formats
- Discussion questions:
  - How schema is stored in each one?
  - How flexible is each one for adding additional fields?

Traditional Column Stores

Header ID:int Name:string Email:string

Column1 1564 1567 1568 1569 1572 …

Column2 Paul Xu Jyeshta Nora Alex …

Column3 paul@gmail.com xu@163.com nil nil alex@live.com

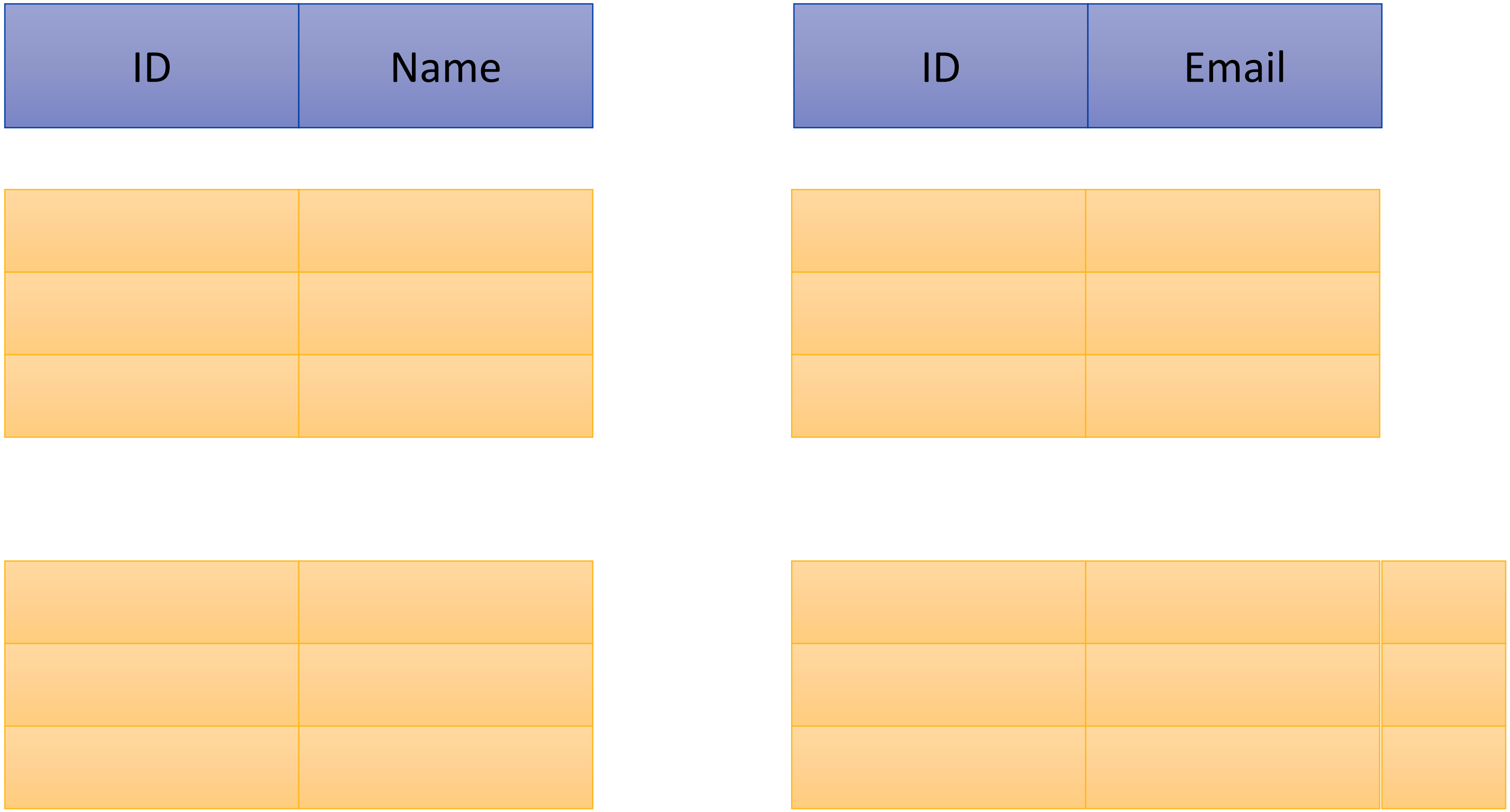
Pros/Cons of Column Formats

- Pros
  - Faster projection
  - Column compression
  - Efficient aggregation
- Cons
  - Not extensible. Cannot easily add more fields
  - Slower when combining multiple columns
  - Slower joins

Partitioned Column Format

- Used in most big-data key-value stores
- Aware of block partitioning in distributed file systems
- Uses row partitioning to group records together
  - Typically based on size
- Uses column partitioning to group relevant columns
  - Typically based on user-provided logic

Partitioned Column Format



ID

Name

ID

Email

Indexing in Big Data

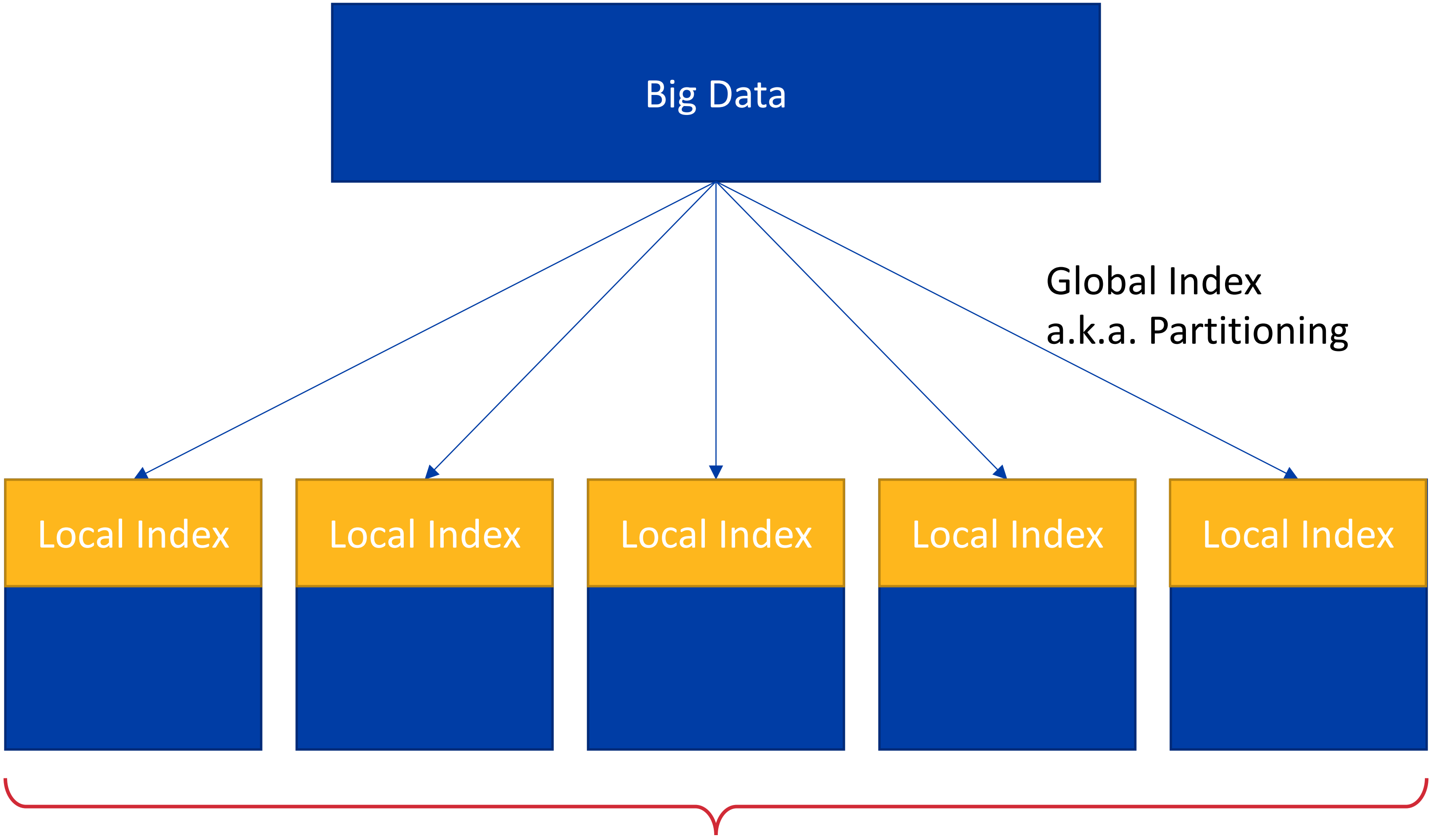
Indexing

- A means for speeding up some queries
- Can help avoiding full scans
- Traditional DBMS indexes
  - B+-tree
  - R-tree
  - Hash indexes
  - Bitmap indexes
- Drawback of traditional indexes
  - Existing implementations cannot scale to big data
  - Use random reads/writes not supported in HDFS

Clustered/Unclustered Indexes

- Clustered indexes
  - Organize records to match the order of the index
  - Good for both point and range queries
  - Can only build one index per dataset
- Unclustered indexes
  - Records are kept as-is
  - Good only for point queries and very small ranges
  - Supports multiple indexes per dataset
  - Rely on random access
- Unclustered indexes are less useful in HDFS. Why?

Distributed Indexes



Big Data

Global Index
a.k.a. Partitioning

Local Index

Local Index

Local Index

Local Index

Local Index

HDFS Blocks

Hash Partitioning

- Advantages
  - Requires one scan over the data
  - Flexible on number of partitions
  - With a good hash function, provides a good load balance
- Drawbacks
  - Supports only point queries
  - Highly skewed key distribution will result in unbalanced partitions

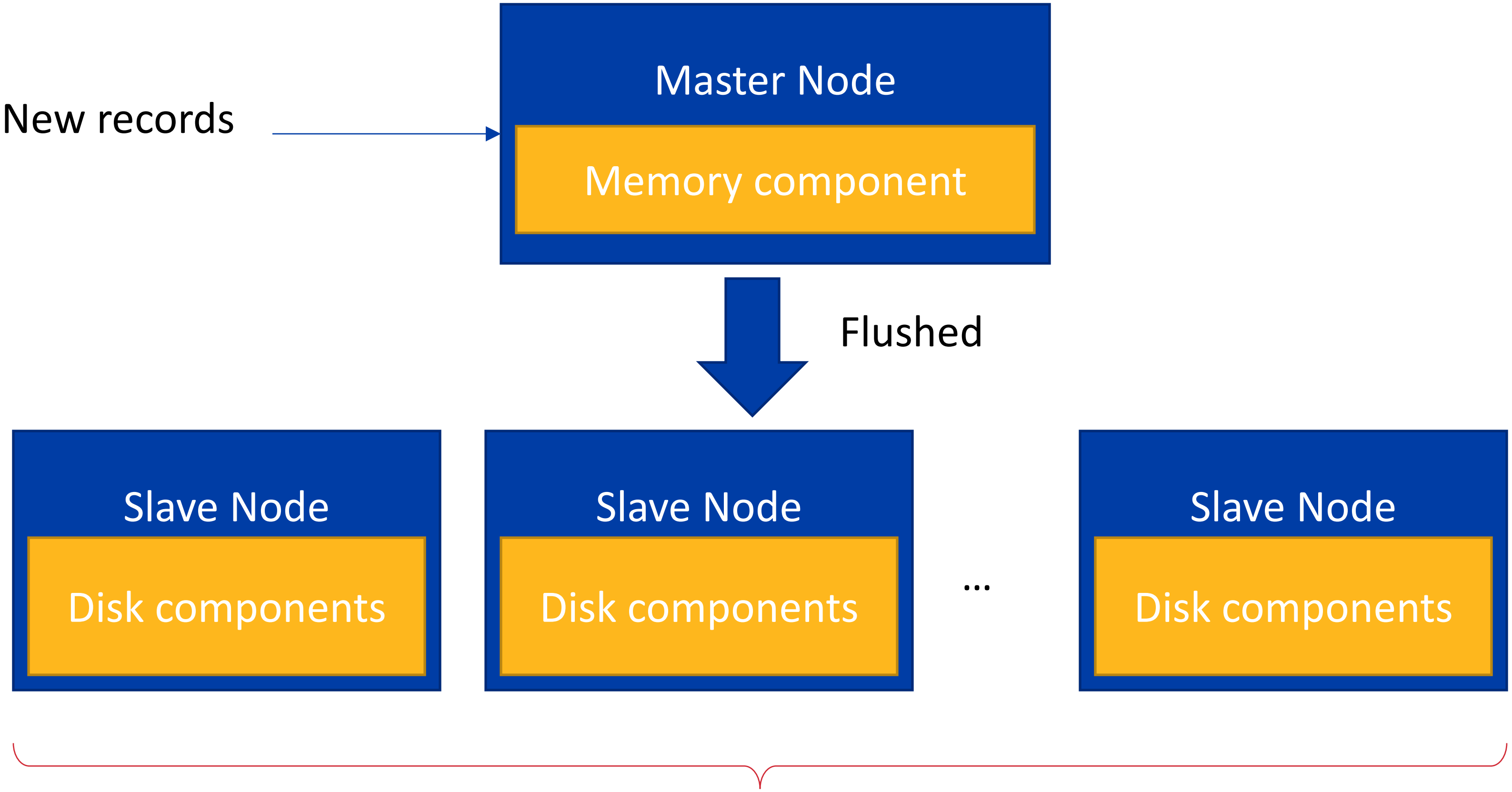
Range Partitioning

- How to find partition boundaries?
- Traditionally, partition boundaries evolve as records are inserted
- Not possible in HDFS where random writes are not allowed
- A common solution
  - Sample the input data (one scan)
  - Calculate partition boundaries (driver machine)
  - Partition the data (one scan)

Dynamic Partitioning

- Very challenging in big data
- Cannot modify existing blocks
- How to insert a record into *closed* ranges?
- Common solution: Log-structured merge-tree (LSM-tree)

LSM Tree



New records

Master Node

Memory component

Flushed

Slave Node

Disk components

Slave Node

Disk components

…

Slave Node

Disk components

Compact and merge (e.g., External merge sort)

Local Indexing

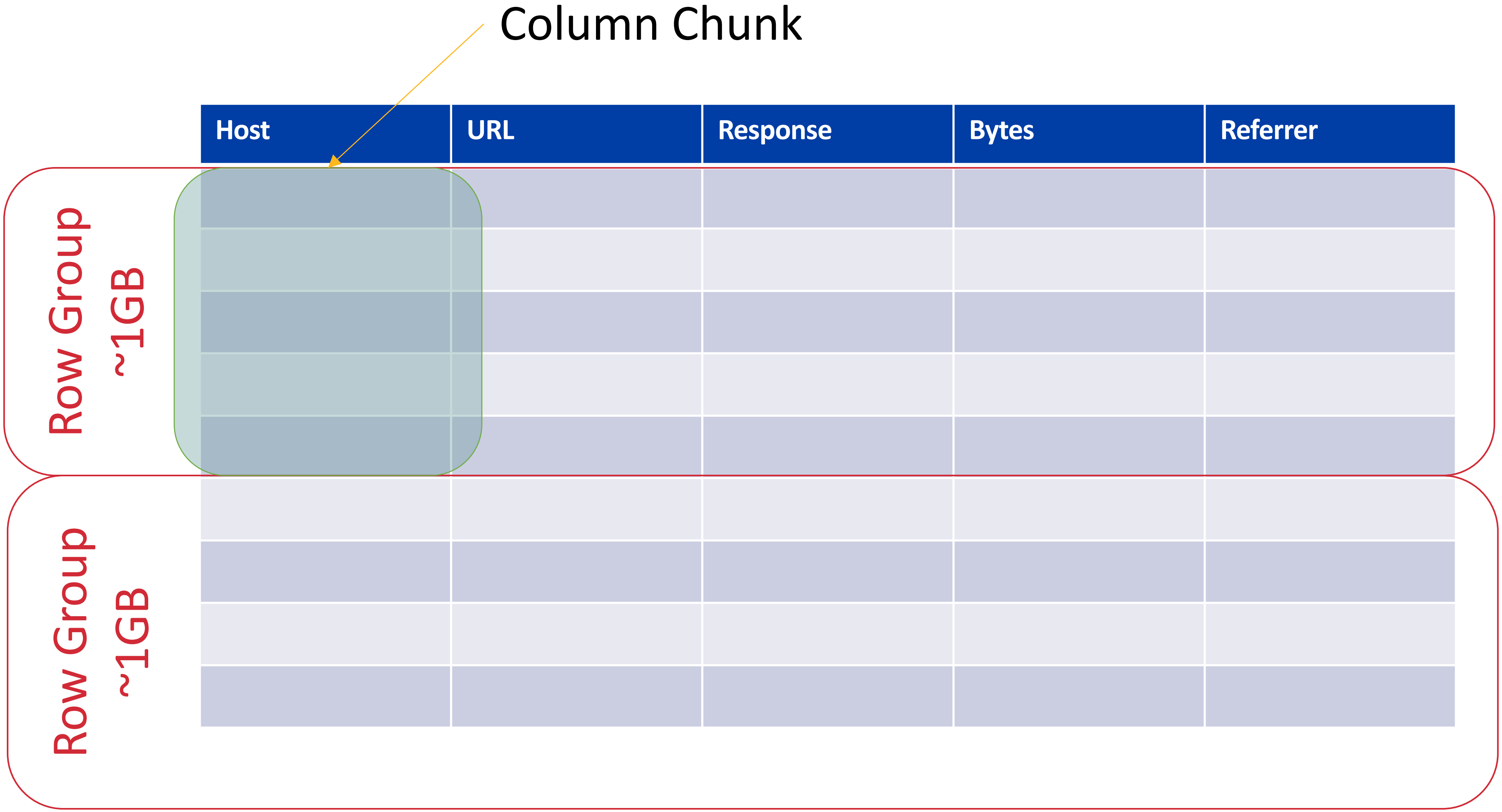
- Relatively easier
- Computed locally in each block before it gets written to disk
- Appended/prepended to the data block
- Given the small size of the block, it can be completely constructed in main-memory before the block is written
- Examples
  - Bloom filter
  - Sorting

Apache Parquet File Format

Apache Parquet

- A column format designed for big data
- Based on Google Dremel
- Designed for distributed file systems
- Supports nesting
- Language independent, can be processed in C++, Java, or other formats

Parquet Overview

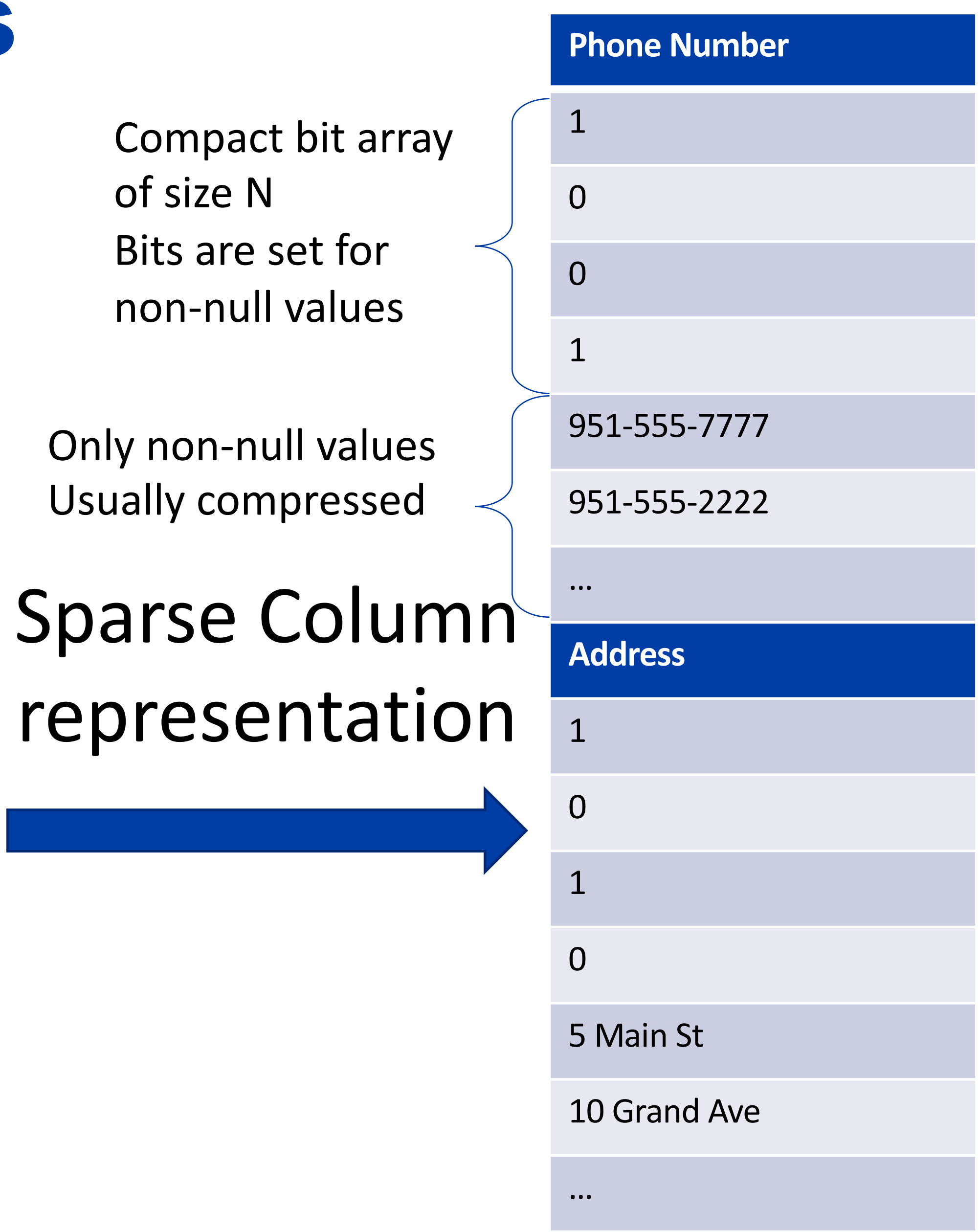


Column Chunk

- A sequence of values of the same type
- In the absence of repetition and nesting, storing one column chunk is straight-forward
- We can store all values as a list
- Values can be compressed or encoded using any of the popular method
- When compressed, each column chunk is further split into *pages* of 16KB each
- Nesting, Repetition, and Nulls , Oh My!

Sparse Columns

Phone Number	Address
951-555-7777	5 Main St
Null	Null
Null	10 Grand Ave
951-555-2222	null
…	…



Sparse Column representation

Compact bit array of size N

Bits are set for non-null values

Only non-null values

Usually compressed

Phone Number

1

0

0

1

951-555-7777

951-555-2222

…

Address

1

0

1

0

5 Main St

10 Grand Ave

…

Nesting

Address	
Street Number	Street Name
5	Main St
Null	Null
10	Grand Ave
Null	Null
100	Null
Null	Google St

Ambiguous!
How do you distinguish between the following records:
{ Phone Number: “951-555-7777”, Address: null}
{ Phone Number: “951-555-1111”, Address: {Number: null, Name: null}

Repetition

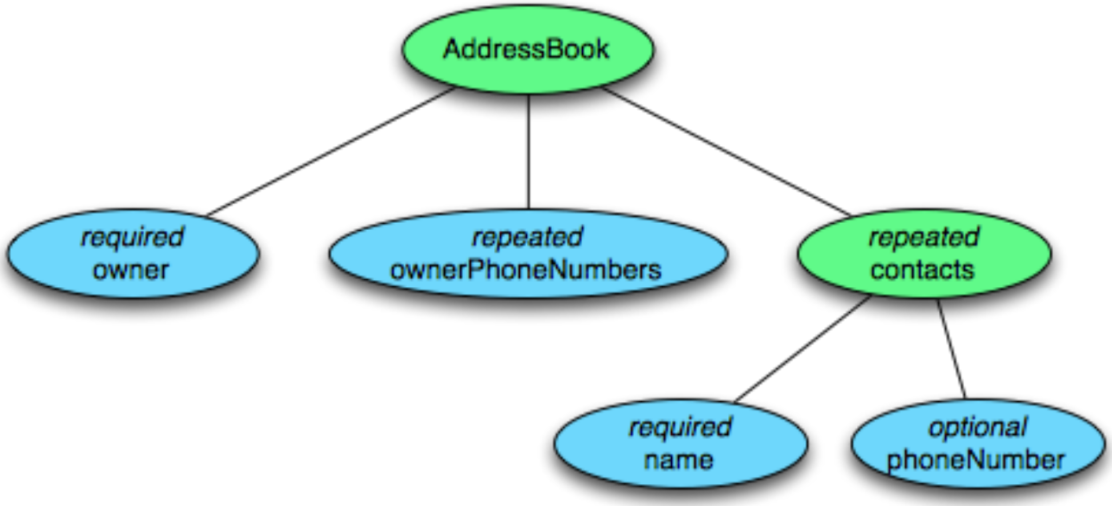
Phone Number
951-555-7777 951-555-3333 951-555-1111
Null
Null
951-555-2222
…

Sparse Column representation

Phone Number
1
0
0
1
951-555-7777
951-555-3333
951-555-1111
951-555-2222
…

Ambiguous!
How to assign values to records?

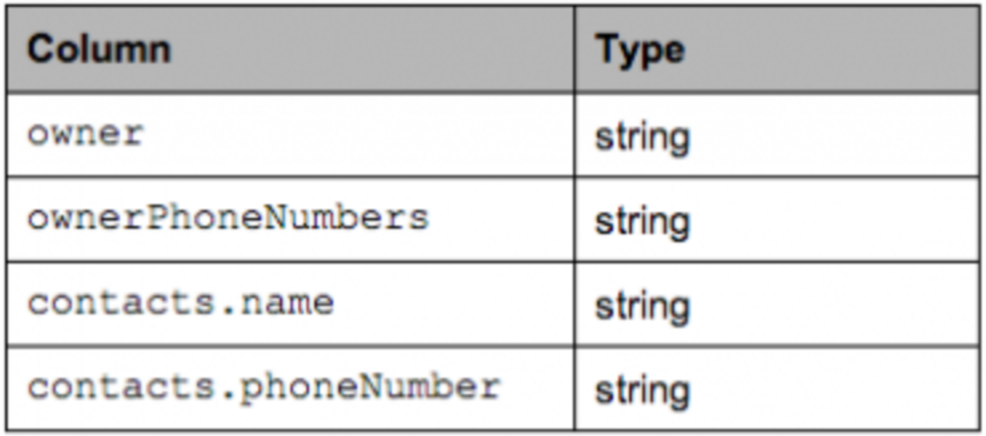
Nesting and Null in Parquet



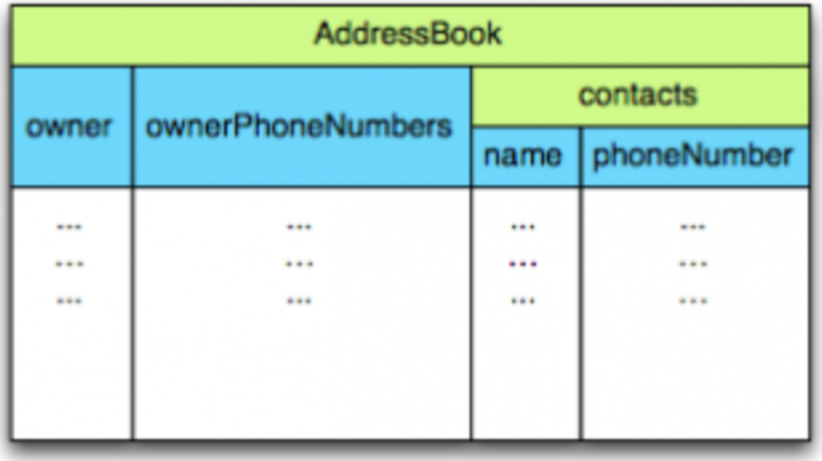
Protocol Buffers definition

Record Schema

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



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



AddressBook			
owner	ownerPhoneNumbers	contacts	
		name	phoneNumber
...	...	...	...
...	...	...	...
...	...	...	...

Examples

```
message1: {
 owner: “Alex”;
 ownerPhoneNumbers: [
  “951-555-7777”, “961-555-9999”
 ],
 contacts: [{
   name: “Chris”;
   phoneNumber: “951-555-6666”;
 }]
}
```

```
message2: {
 owner: null;
 ownerPhoneNu[illegible]ers: [
  “951-555-77[illegible] “[illegible]555-9999”
 ],
 contacts: [{
   name: “Chr[illegible],
   phoneNumbe[illegible]. “951-5[illegible]5-6666”;
 }]
}
```

```
message3: {
 owner: “Joe”;
 ownerPhoneNumbers: [
  “951-555-4444”, “961-555-3333”
 ]
}
```

```
message4: {
 owner: “Olivia”;
 ownerPhoneNumbers: [
  “951-555-2222”
 ],
 contacts: [{
   name: “Chris”;
   phoneNumber: null;
 }]
}
```

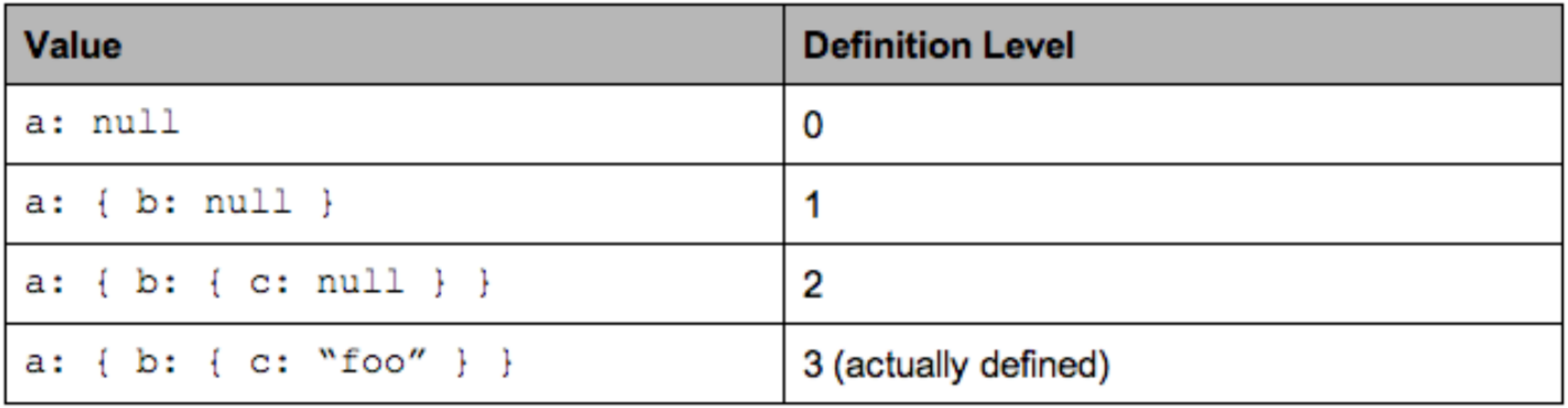
```
message5: {
 owner: “Violet”;
 ownerPhoneNumbers: [
  “961-555-1111”
 ]
}
```

Definition Level

- The nesting level at which a field is null

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

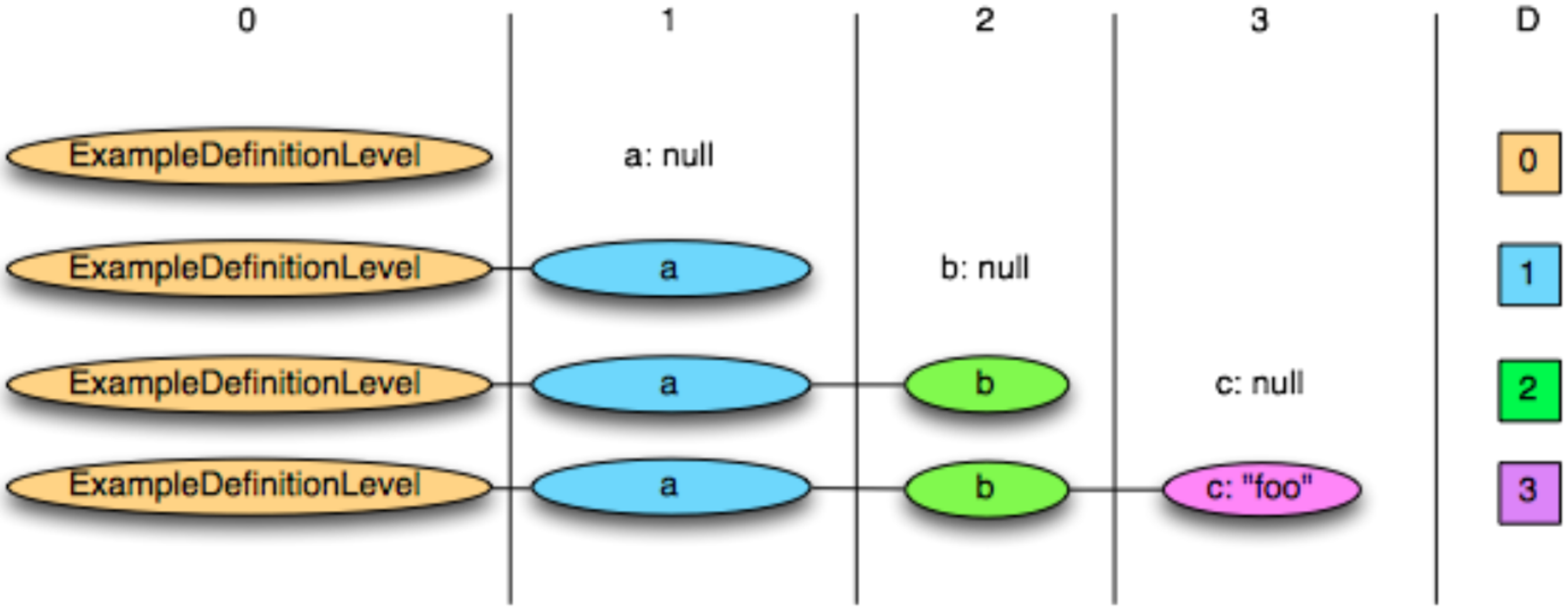
Observation: If no nesting is involved, i.e., one level, this scheme falls back to the 0/1 schema of flat data



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

Definition Level

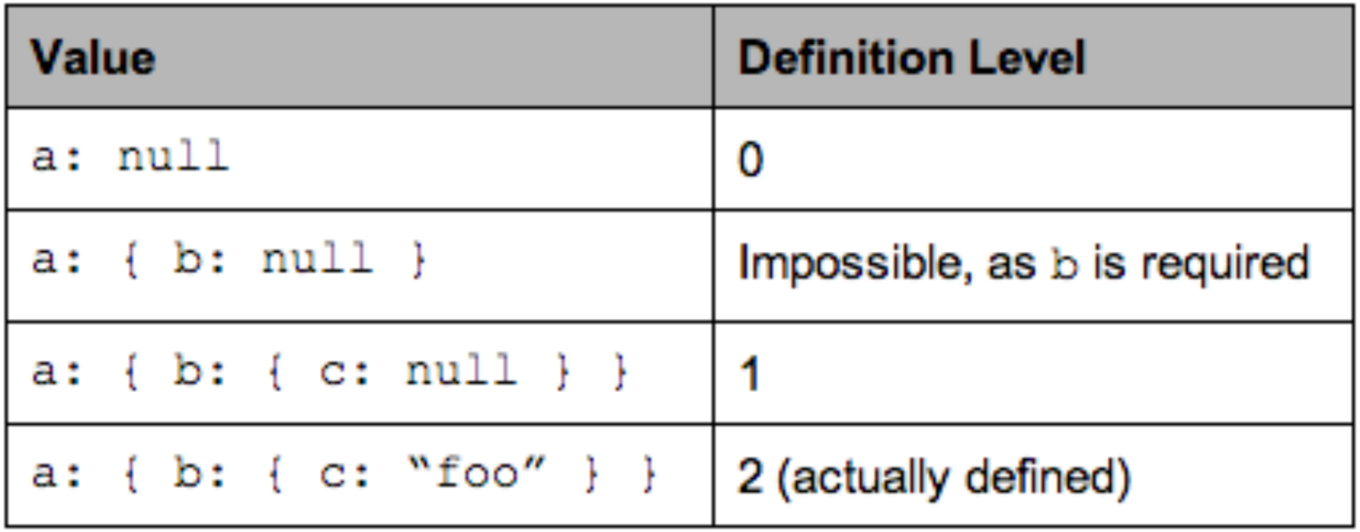
Value	Definition Level
`a: null`	0
`a: { b: null }`	1
`a: { b: { c: null } }`	2
`a: { b: { c: “foo” } }`	3 (actually defined)



Definition Level with Required

- When a field is required (not nullable), then there is one definition level that is not allowed

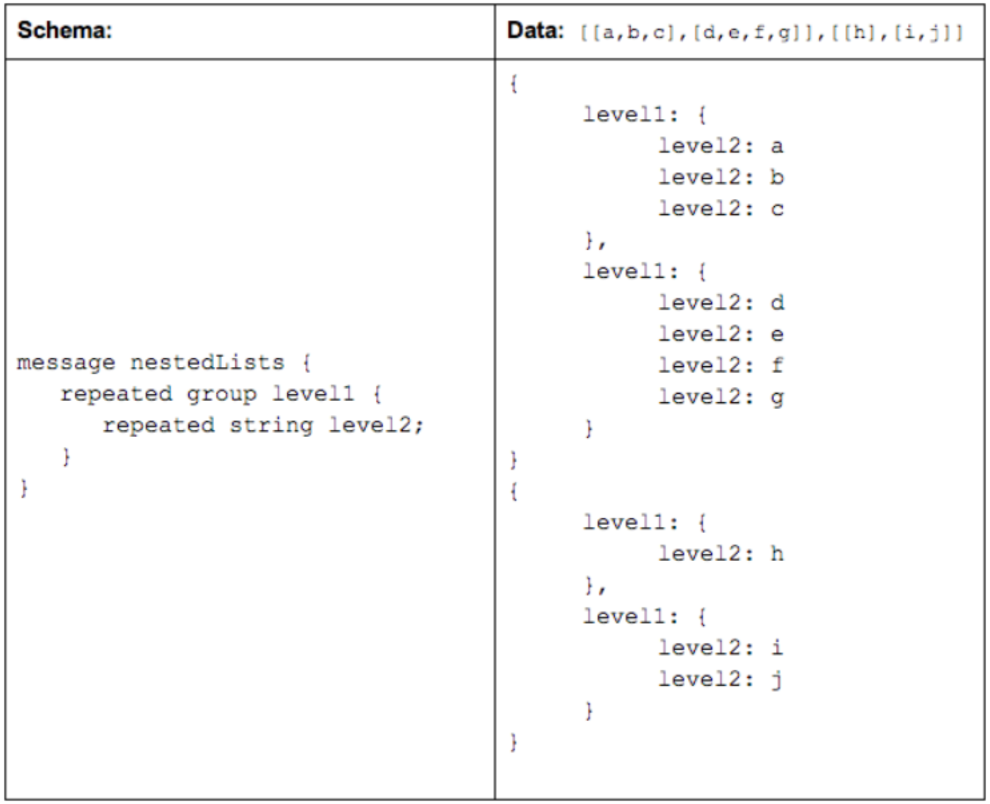
```
message ExampleDefinitionLevel {
  optional group a {
    required group b {
      optional string c;
    }
  }
}
```



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 Level

- The level at which we should create a new list

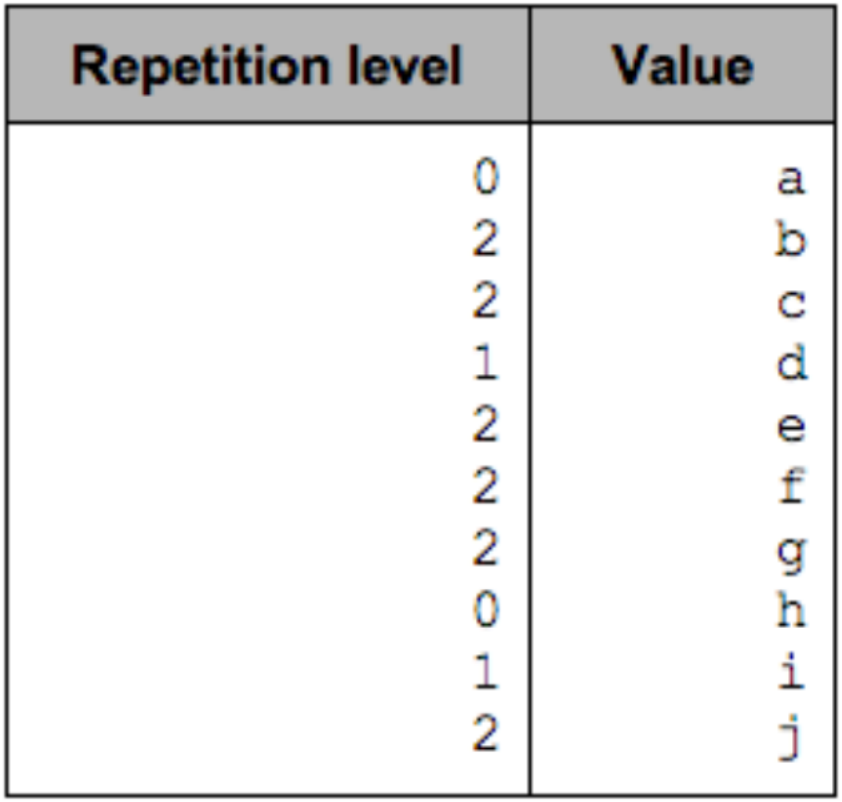


Schema:

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

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

```
{
        level1: {
                level2: a
                level2: b
                level2: c
        },
        level1: {
                level2: d
                level2: e
                level2: f
                level2: g
        }
}
{
        level1: {
                level2: h
        },
        level1: {
                level2: i
                level2: j
        }
}
```

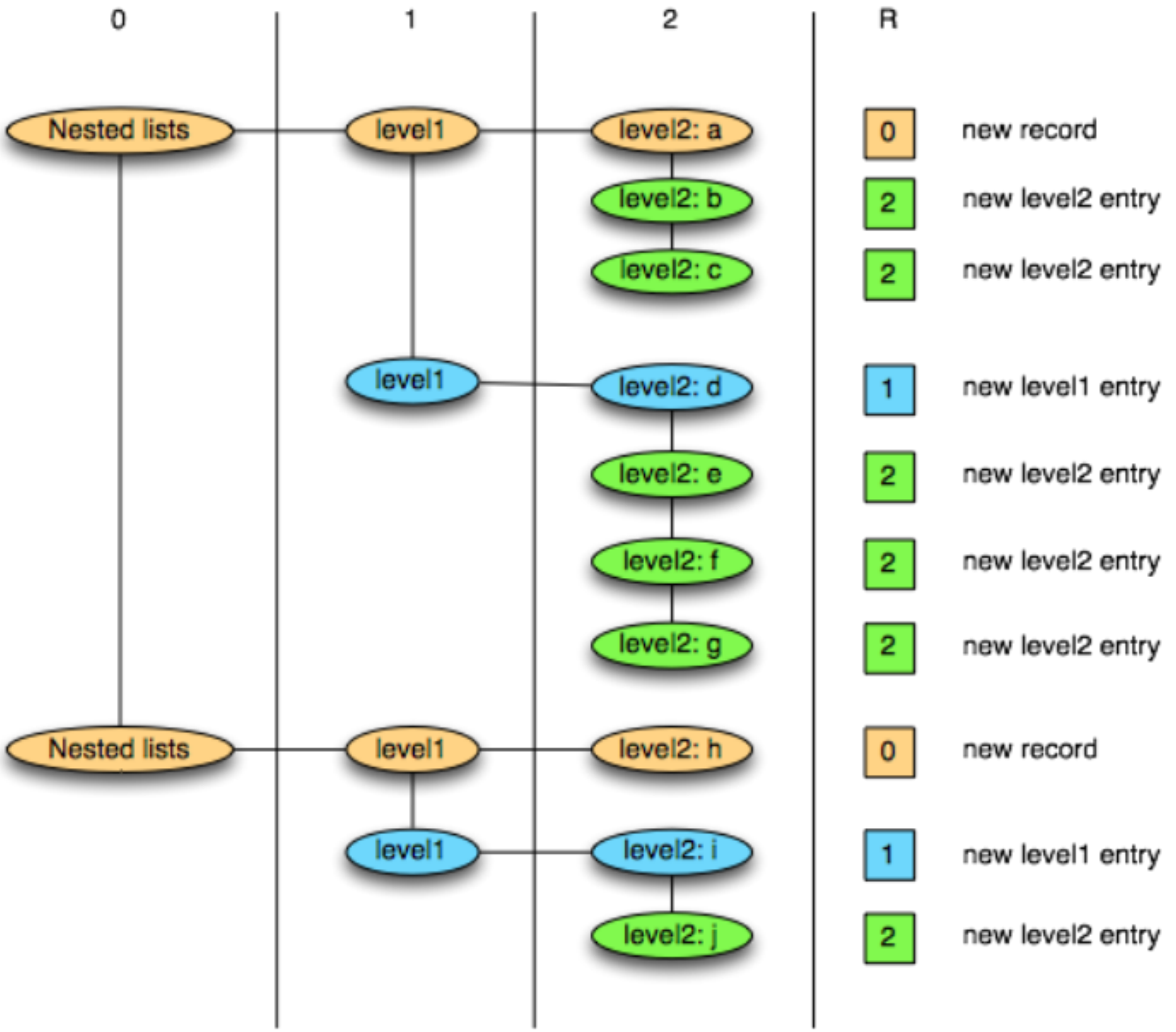


Repetition level	Value
0	a
2	b
2	c
1	d
2	e
2	f
2	g
0	h
1	i
2	j

Repetition Level

- The repetition level marks the beginning of lists and can be interpreted as follows:
  - 0 marks the *first value of every attribute* in each 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 Level



AddressBook Example

Record Schema

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

Attribute	Optional	Max Definition level	Max Repetition level
Owner	No	0 (owner is required)	0 (no repetition)
Owner phone number	Yes	1	1 (repeated)
Contacts.name	No	1 (name is required)	1 (contacts is repeated)
Contacts.Phone number	Yes	2 (phone is optional)	1 (contacts is repeated)

Example

```
DocId: 10
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 in64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    option String Url;}}
```

```
DocId: 20
Links
    Backward: 10
    Backward: 30
    Forward: 80
Name
    Url: ‘http://C’
```

Summary

- Two orthogonal problems in big-data storage
  - File formats (row, column, or hybrid)
  - Indexing (Global and local)
- File formats
  - Row: Flexible but inefficient
  - Column: Efficient for some queries but inflexible
- Indexing
  - Global: Load-balanced partitioning
  - Local: Additional metadata affixed to each block
- Parquet: A common column format for big data

Further Reading

- **Dremel made simple with Parquet [**https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.html]
- Apache Parquet project homepage [http://parquet.apache.org]
- Parquet for MapReduce (works for both Hadoop and Spark) [https://github.com/apache/parquet-mr]