File formats

Motivation

- Tabular data
- Multiple data types
- Optional (null) values
- No nested or repeated values
- Large number of columns

Id	***Name***	***Location***
1	aa	Braga
2	bbb	Porto
3	cc	Porto
4	dddddd	
5	eee	Lisboa
...	...	...

Issues

- Representation of types
  - Compactness and ambiguity
- Data that needs to be moved for:
  - Selection (range scan)
  - Projection
- Compression

Text (CSV)

- Simple to produce and consume
- Schema can be inferred
- Redundancy and verbose representation (numbers)
- Ambiguity in separators and missing fields
- Difficult to page, especially when compressed

Id	***Name***	***Location***
1	aa	Braga
2	bbb	Porto
3	cc	Porto
4	dddddd	
5	eee	Lisboa
...		...

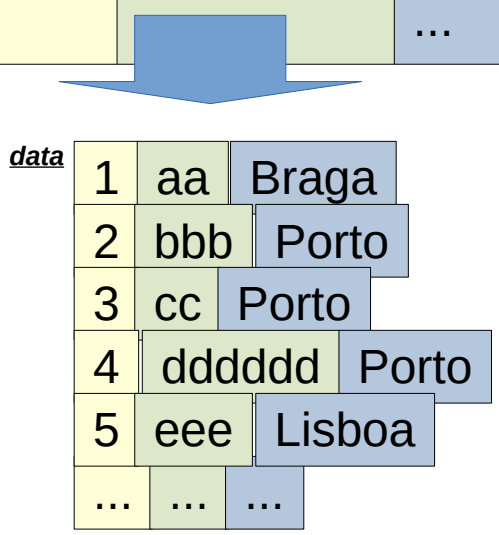
data.csv

```
"1","aa","Braga"
"2","bbb","Porto"
"3","cc","Porto"
"4","dddddd",
"5","eee","Lisboa"
…,…,...
```

Binary rows

- Compact and unambiguous
- Efficient I/U/D
- Can be paged and compressed
  - Not efficient as different data types are interleaved
- All data is read for projections

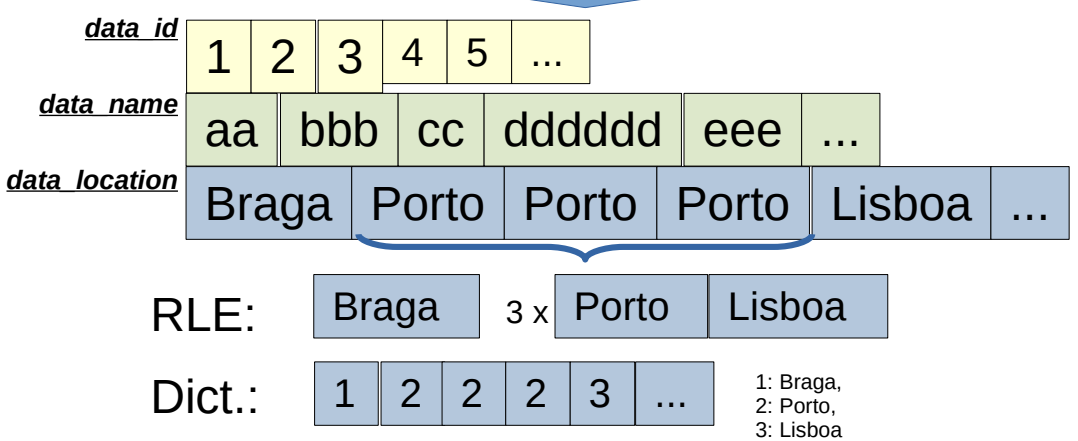
Id	*Name*	*Location*
1	aa	Braga
2	bbb	Porto
3	cc	Porto
4	dddddd	
5	eee	Lisboa
...		...



Columnar

- Efficient projections
- Compressed very efficiently
  - Dictionary and/or
  - Run Length Encoding (RLE)
- Inefficient I/U/D
- Inefficient range scan

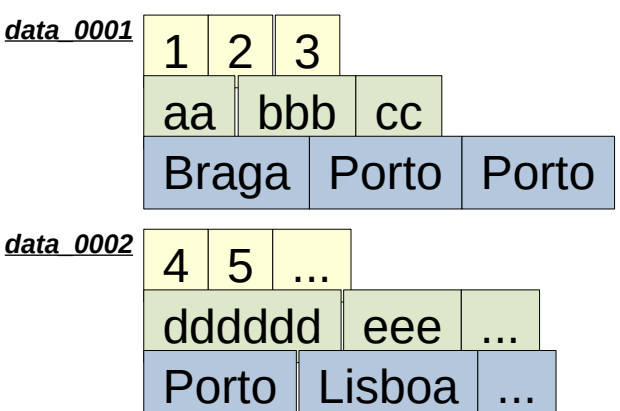
Id	*Name*	*Location*
1	aa	Braga
2	bbb	Porto
3	cc	Porto
4	dddddd	
5	eee	Lisboa
...		...



Hybrid

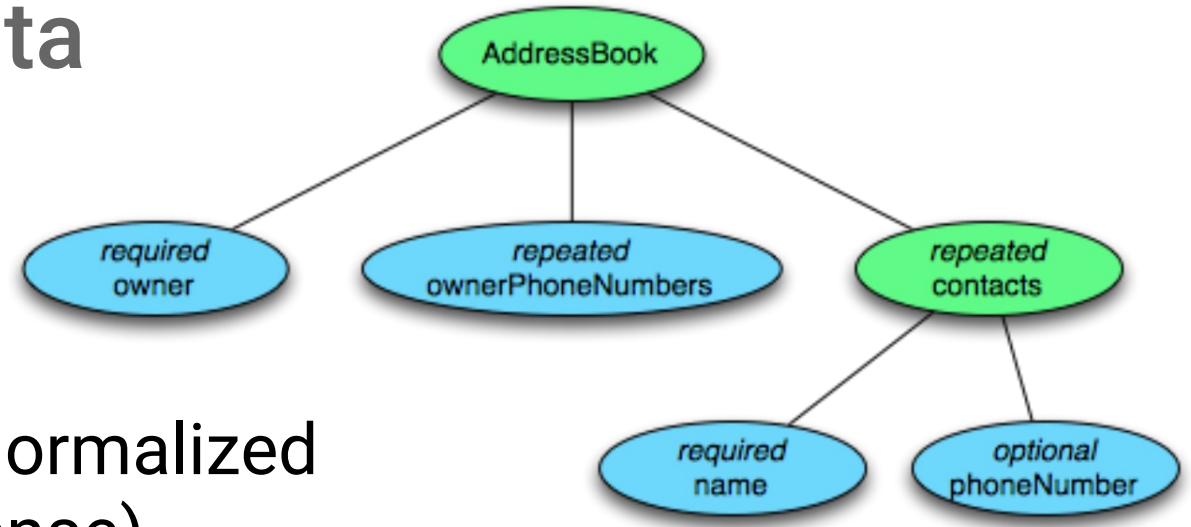
- Columnar segments, that can be accessed and compressed separately
- Good trade-off:
  - I/U/D updates only one segment
  - Range scans can read only some segments
  - Projections can easily skip columns

Id	*Name*	*Location*
1	aa	Braga
2	bbb	Porto
3	cc	Porto
4	dddddd	
5	eee	Lisboa
...		...



Hierarchical file formats

Hierarchical data



- Data that is not normalized (in a relational sense)
  - Nested structures
  - Repeated fields
- Useful as it avoids multiple files and foreign keys

Image from:https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.html

JSON

- Well-known and widely supported
- Row-based
- Not splitable

```
{
  "AddressBook": [
    {
      "owner": "Jason F.",
      "ownerPhoneNumbers": [
        "123456789",
        "987654321"
      ],
      "contacts": [
        { "name":  "John" },
        { "name":  "Joe", "number":  "214365879" }
      ]
    },
    {
      "owner": "Joe G.",
      "ownerPhoneNumbers": [
        "214365879"
      ]
    }
  ]
}
```

Dremel splitting

- One columnar file for each leaf attribute
- How to match records in different columns?
- Avoid additional information: record numbers, keys, …
  - https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.html

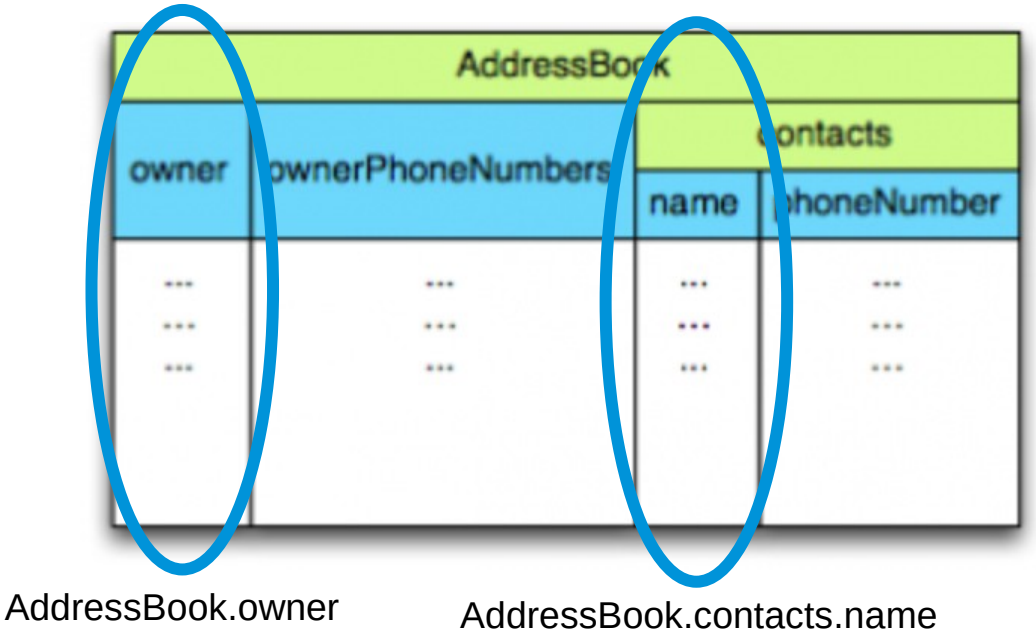


Image from:https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.html

AVRO™ + Parquet

Example: Parquet-Avro

- Avro is a binary format
- Parquet is a hybrid columnar layout
- Parquet-Avro is a hybrid columnar layout using Avro for representing atomic types
- Support for Hadoop Map-Reduce input and output

```
<dependency>
    <groupId>org.apache.parquet</groupId>
    <artifactId>parquet-hadoop-bundle</artifactId>
    <version>1.11.1</version>
</dependency>

<dependency>
    <groupId>org.apache.parquet</groupId>
    <artifactId>parquet-avro</artifactId>
    <version>1.11.1</version>
</dependency>
```

Parquet-Avro Schema

```
message AddressBook {
  required binary owner (STRING);
  required group ownerPhoneNumbers (LIST) {
     repeated int32 array;
  }
  required group contacts (LIST) {
     repeated group contact {
       required binary name (STRING);
       optional int32 phoneNumber;
     }
  }
}
```

```
InputStream is = ...;
String ps = new String(is.readAllBytes());
MessageType mt = MessageTypeParser.parseMessageType(ps);
Schema schema = new AvroSchemaConverter().convert(mt);
```

```
{
  "type": "record",
  "name": "AddressBook",
  "fields": [
    {
      "name": "owner",
      "type": "string"
    },
    {
      "name": "ownerPhoneNumbers",
      "type": {
        "type": "array",
        "items": "int"
      }
    },
    {
      "name": "contacts",
      "type": {
        "type": "array",
        "items": {
          "type": "record",
          "name": "contact",
          "fields": [
            {
              "name": "name",
              "type": "string"
            },
            {
              ...
```

Documentation:
https://github.com/apache/parquet-format/blob/master/LogicalTypes.md

Parquet-Avro Output and Input

```
job.setOutputFormatClass(AvroParquetOutputFormat.class);
AvroParquetOutputFormat.setSchema(job, schema);
FileOutputFormat.setOutputPath(job, new Path("..."));


public static class ToParquetMapper extends Mapper<..., ..., Void, GenericRecord> {
   …
}



job.setInputFormatClass(AvroParquetInputFormat.class);
AvroParquetInputFormat.addInputPath(job, new Path("..."));

AvroParquetInputFormat.setRequestedProjection(job, schema);


public static class FromParquetMapper extends Mapper<Void, GenericRecord, ..., ...> {
   …
}
```

optional!

GenericRecord in Parquet-Avro

- Creating a record:

```
GenericRecord record = new GenericData.Record(schema);
```

- Reading and writing a field:

```
String owner = record.get("owner");
record.put("owner", owner);
```

- Getting a nested schema. for nested records:

```
Schema array_elem_schema = schema.getField("contacts").schema().getElementType();
```

Multiple inputs

- AvroParquetInputFormat is not compatible with multiple inputs (e.g., a shuffle join), unless all files have the same schema
  - But hierarchical data also makes it less likely that a multi-input job is needed...
- Workaround: Use separate map-only jobs to perform the map portion of a multi-input job
- Store map outputs in temporary files:
  - Best format for this is SequenceFile with SequenceFileOutputFormat / SequenceFileInputFormat
- Use a multi-input job that reads temporary files and performs the reduce