cloudera

Data Formats

Chapter 7

201509

Course Chapters

Data Formats

In this chapter you will learn

- **How to select the best data format for your needs**
- **How various Hadoop tools support different data formats**
- **How to define Avro schemas**
- **How Avro schemas can evolve to accommodate changing requirements**
- **How to extract data and metadata from an Avro data file**

Chapter Topics

Data Formats | **Importing and Modeling Structured Data**

- **Selecting a File Format**
- Avro Schemas
- Avro Schema Evolution
- Using Avro with Impala, Hive, and Sqoop
- Using Parquet with Impala, Hive, and Sqoop
- Compression
- Conclusion
- Homework: Select a Format for a Data File

File Storage Formats

- **In previous chapters you saw that alternate file formats were available**
  - E.g., in Hive and Impala

```
CREATE TABLE tablename (colname DATATYPE, ...)
   ROW FORMAT DELIMITED
   FIELDS TERMINATED BY char
   STORED AS format
```

  - E.g., in Sqoop

```
--as-format
```

- **What formats are available?**
- **Which should you choose and why?**

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
- **Verdict: 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
- **Verdict: Good performance, but poor interoperability**

Hadoop File Formats: Avro Data Files

- **Efficient storage due to optimized binary encoding**
- **Widely supported throughout the Hadoop ecosystem**
  - Can also be used outside of Hadoop
- **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
- **Verdict: Excellent interoperability and performance**
  - Best choice for general-purpose storage in Hadoop
- ***More detail in coming slides***

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, Crunch, 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
- **Verdict: Excellent interoperability and performance**
  - Best choice for column-based access patterns

HIDDEN SLIDE RCFile/ORCFile instructor notes

Chapter Topics

Data Formats	Importing and Modeling Structured Data

- Selecting a File Format
- **Avro Schemas**
- Avro Schema Evolution
- Using Avro with Impala, Hive, and Sqoop
- Using Parquet with Impala, Hive, and Sqoop
- Compression
- Conclusion
- Homework: Select a Format for a Data File

Data Serialization

- **To understand Avro, you must first understand *serialization***
  - A way of representing data in memory as a series of bytes
  - Allows you to save data to disk or send it across the network
  - *Deserialization* allows you to read that data back into memory
- **For example, how do you serialize the number 108125150?**
  - 4 bytes when stored as a Java `int`
  - 9 bytes when stored as a Java `String`
- **Many programming languages and libraries support serialization**
  - Such as `Serializable` in Java or `pickle` in Python
- **Backwards compatibility and cross-language support can be challenging**
  - Avro was developed to address these challenges

What is Apache Avro?

- **Avro Data File Format is just one part of the Avro project**
  - But it is the part this course focuses on
- **Avro is an efficient data serialization framework**
  - Top-level Apache project created by Doug Cutting (creator of Hadoop)
  - Widely supported throughout Hadoop and its ecosystem
- **Offers compatibility without sacrificing performance**
  - Data is serialized according to a *schema* you define
  - Read/write data in Java, C, C++, C#, Python, PHP, and other languages
  - Serializes data using a highly-optimized binary encoding
  - Specifies rules for *evolving* your schema over time
- **Avro also supports Remote Procedure Calls (RPC)**
  - Can be used for building custom network protocols
  - Flume uses this for internal communication

Supported Types in Avro Schemas (Simple)

- **A simple type holds exactly one value**

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`

Supported Types in Avro Schemas (Complex)

- **Avro also supports 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

- **The `record` type is the most important**
  - Main use of other types is to define a record's fields

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.data",
 "type":"record",
 "name":"Employee",
 "fields":[
     {"name":"id", "type":"int"},
     {"name":"name", "type":"string"},
     {"name":"title", "type":"string"},
     {"name":"bonus", "type":"int"}]
}
```

Specifying Default Values in the Schema

- **Avro also supports setting a default value in the schema**
  - Used when no value was explicitly set for a field
  - Similar to SQL

```
{"namespace":"com.loudacre.data",
 "type":"record",
 "name":"Invoice",
 "fields":[
    {"name":"id", "type":"int"},
    {"name":"taxcode", "type":"int", "default":"39"},
    {"name":"lang", "type":"string", "default":"EN_US"}]
}
```

The `taxcode` and `lang` fields have default values

Avro Schemas and Null Values

- **Avro checks for null values when serializing the data**
- **Null values are only allowed *when explicitly specified* in the schema**

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

The `title` and `bonus` fields allow null values

Schema Example with Complex Types

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

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

- **It's a good practice to document any ambiguities in a schema**
  - All types (including record) support an optional `doc` attribute

```
{"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 Container Format

- **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

Inspecting Avro Data Files with Avro Tools

- **Avro data files are an efficient way to store data**
  - However, 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
  - Included with CDH 5 and later
  - Available for download from the Avro Web site or Maven repository

```
$ 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
```

Chapter Topics

Data Formats	Importing and Modeling Structured Data

- Selecting a File Format
- Avro Schemas
- **Avro Schema Evolution**
- Using Avro with Impala, Hive, and Sqoop
- Using Parquet with Impala, Hive, and Sqoop
- Compression
- Conclusion
- Homework: Select a Format for a Data File

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 **`SequenceFile`**s 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: A Practical Example (1)

- **Imagine that we have written millions of records with this schema**

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

Schema Evolution: A Practical Example (2)

- **We would like to modernize this based on the schema below**
  - Rename `id` field to `customerId` and change type from `int` to `long`
  - Remove `faxNumber` field
  - Add `prefLang` field
  - Add `email` field

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

Schema Evolution: A Practical Example (3)

- **We could use the new schema to write new data**
  - Applications that use the new schema could read the new data
- **Unfortunately, new applications wouldn't be able to read the *old* data**
  - We must make a few schema changes to improve compatibility

Schema Evolution: A Practical Example (4)

- **If you rename a field, you must specify an alias for the old name(s)**
  - Here, we map the old `id` field to the new `customerId` field

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

Schema Evolution: A Practical Example (5)

- **Newly-added fields will lack values for records previously written**
  - You must specify a default value

```
{"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}
]}
```

Default value for **prefLang** is **en_US**

email is nullable so **null** can be the default

Schema Evolution: Compatible Changes

- **The following changes will not affect existing readers**
  - Adding, changing, or removing a `doc` attribute
  - Changing a field's default value
  - Adding a new field with a default value
  - Removing a field that specified a default value
  - Promoting a field to a wider type (e.g., `int` to `long`)
  - Adding aliases for a field

Schema Evolution: Incompatible Changes

- **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**
  1. Read your old data (using the original schema)
  2. Modify data as needed in your application
  3. Write the new data (using the new schema)
- **Existing readers/writers may need to be updated to use new schema**

Chapter Topics

Data Formats | **Importing and Modeling Structured Data**

- Selecting a File Format
- Avro Schemas
- Avro Schema Evolution
- **Using Avro with Impala, Hive, and Sqoop**
- Using Parquet with Impala, Hive, and Sqoop
- Compression
- Conclusion
- Homework: Select a Format for a Data File

Using Avro with Sqoop

- **Sqoop supports importing data as Avro, or exporting data from existing Avro data files**
- **`sqoop import` saves the schema JSON file in local directory**

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

Using Avro with Impala and Hive (1)

- **Hive and Impala support Avro**
  - Hive supports all Avro types
  - Impala does not support complex types
    - **`enum`**, **`array`**, etc.

Using Avro with Impala and Hive (2)

- **Table creation can include schema inline or in a separate file**

```
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"}
        ]}');
```


Chapter Topics

Data Formats	Importing and Modeling Structured Data

- Selecting a File Format
- Avro Schemas
- Avro Schema Evolution
- Using Avro with Impala, Hive, and Sqoop
- **Using Parquet with Impala, Hive, and Sqoop**
- Compression
- Conclusion
- Homework: Select a Format for a Data File

Using Parquet with Sqoop

- **Sqoop supports importing data as Parquet, or exporting data from existing Parquet data files**
  - Use the **`--as-parquetfile`** option

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

Using Parquet with Hive and Impala (1)

- **Create a new table stored in Parquet format**

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

* **STORED AS PARQUET** supported in Impala, and in Hive 0.13 and later

Using Parquet with Hive and Impala (2)

- **In Impala, use `LIKE PARQUET` to use column metadata from an existing Parquet data file**
- **Example: Create a new table to access existing Parquet format data**

```
CREATE EXTERNAL TABLE ad_data
  LIKE PARQUET '/loudacre/ad_data/datafile1.parquet'
  STORED AS PARQUET
  LOCATION '/loudacre/ad_data/';
```


Chapter Topics

Data Formats	Importing and Modeling Structured Data

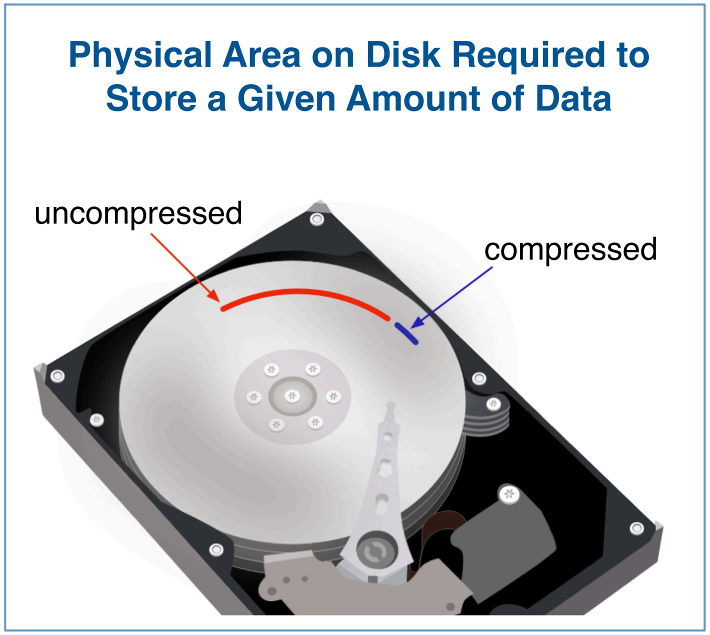
- Selecting a File Format
- Avro Schemas
- Avro Schema Evolution
- Using Avro with Impala, Hive, and Sqoop
- Using Parquet with Impala, Hive, and Sqoop
- **Compression**
- Conclusion
- Homework: Select a Format for a Data File

Performance Considerations

- **You have just learned how different file formats affect performance**
- **Another factor can significantly affect performance: data compression**

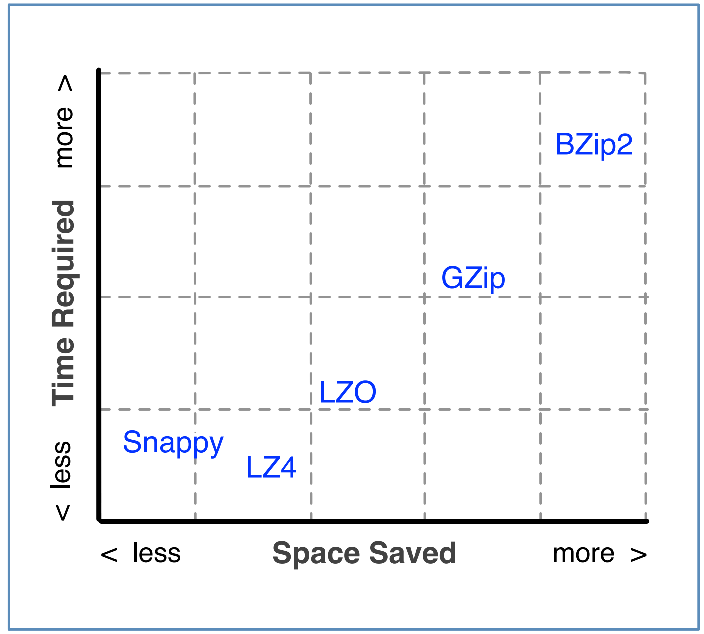
Data Compression

- **Each file format may also support compression**
  - This reduces amount of disk space required to store data
- **Compression is a tradeoff between CPU time and bandwidth/storage space**
  - Aggressive algorithms take a long time, but save more space
  - Less aggressive algorithms save less space but are much faster
- **Can significantly improve performance**
  - Many Hadoop jobs are I/O-bound
  - Using compression allows you to handle more data per I/O operation
  - Compression can also improve the performance of network transfers



Compression Codecs

- **The implementation of a compression algorithm is known as a *codec***
  - Short for compressor/decompressor
- **Many codecs are commonly used with Hadoop**
  - Each has different performance characteristics
  - Not all Hadoop tools are compatible with all codecs
- **Overall, BZip2 saves the most space**
  - But LZ4 and Snappy are much faster
  - Impala supports Snappy but not LZ4
- **For "hot" data, speed matters most**
  - Better to compress by 40% in one second than by 80% in 10 seconds



Using Compression With Sqoop

- **Sqoop – use `--compression-codec` flag**
  - Example

```
--compression-codec
org.apache.hadoop.io.compress.SnappyCodec
```

Using Compression With Impala and Hive

- **Not all file format/compression combinations are supported**
  - Properties and syntax varies
  - See the documentation for a full list of supported formats and codecs for Impala and Hive
  - Caution: Impala queries data in memory – both compressed and uncompressed data are stored in memory

- **Impala example**

```
> CREATE TABLE mytable_parquet LIKE mytable_text
    STORED AS PARQUET;
> set PARQUET_COMPRESSION_CODEC=snappy;
> INSERT INTO mytable_parquet
    SELECT * FROM mytable_text;
```

Chapter Topics

Data Formats	Importing and Modeling Structured Data

- Selecting a File Format
- Avro Schemas
- Avro Schema Evolution
- Using Avro with Impala, Hive, and Sqoop
- Using Parquet with Impala, Hive, and Sqoop
- Compression
- **Conclusion**
- Homework: Select a Format for a Data File

Essential Points (1)

- **Hadoop and its ecosystem support many file formats**
  - May ingest data in one format, but convert to another as needed
- **Selecting the format for your data set involves several considerations**
  - Ingest pattern
  - Tool compatibility
  - Expected lifetime
  - Storage and performance requirements

Essential Points (2)

- **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
- **Compression saves disk storage space and IO times at the cost of CPU time**
  - Hadoop tools support a number of different codecs

Bibliography

The following offer more information on topics discussed in this chapter

- **Avro Getting Started Guide (Java)**
  - `http://tiny.cloudera.com/adcc03a`
- **Avro Specification**
  - `http://tiny.cloudera.com/adcc03b`
- **Parquet**
  - `https://parquet.apache.org`
- **Announcing Parquet 1.0: Columnar Storage for Hadoop**
  - `http://tiny.cloudera.com/adcc03c`

Chapter Topics

Data Formats	Importing and Modeling Structured Data

- Selecting a File Format
- Avro Schemas
- Avro Schema Evolution
- Using Avro with Impala, Hive, and Sqoop
- Using Parquet with Impala, Hive, and Sqoop
- Compression
- Conclusion
- **Homework: Select a Format for a Data File**

Homework: Select a Format for a Data File

- **In this homework assignment you will**
  - Use Sqoop to import the accounts table in Avro format
  - Define an Impala table to access the Avro accounts data
  - Bonus: Save an existing plain text Impala table as Parquet
- **Please refer to the Homework description**