

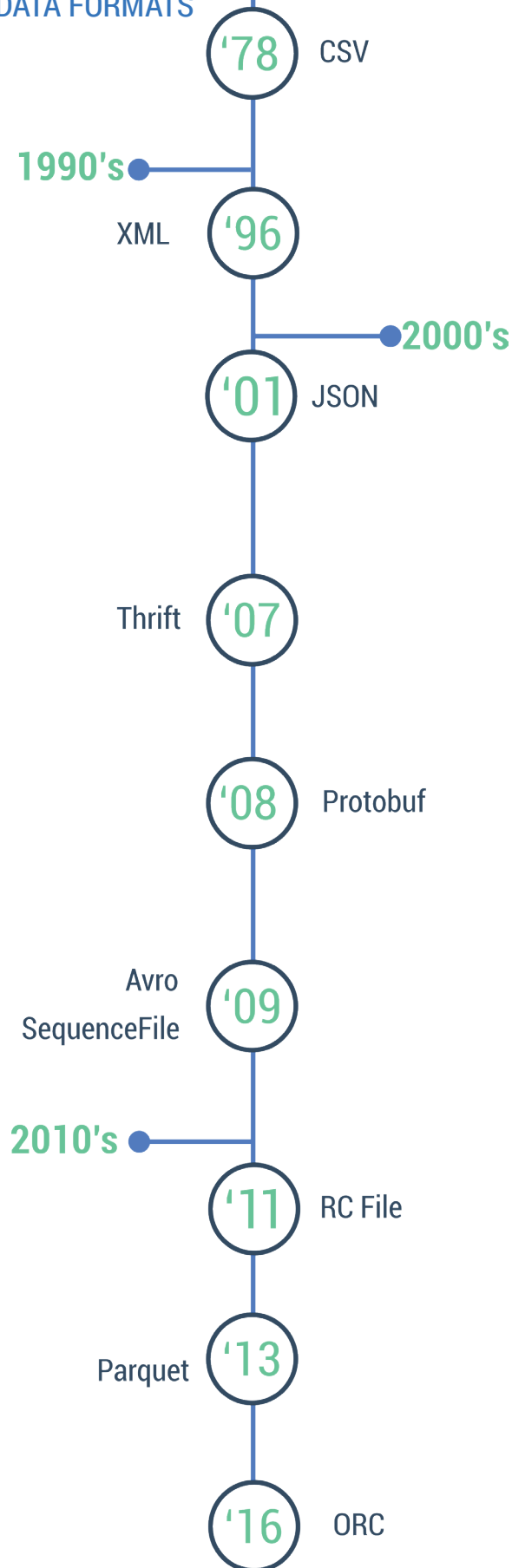
An Introduction to Big Data Formats

Understanding Avro, Parquet, and ORC

TABLE OF CONTENTS

| | |
|-------------------------------------|----|
| TABLE OF CONTENTS | 2 |
| INTRODUCTION | 3 |
| FORMAT EVALUATION FRAMEWORK | 4 |
| ROW VS. COLUMN | 4 |
| SCHEMA EVOLUTION | 6 |
| SPLITABILITY | 7 |
| COMPRESSION | 7 |
| THE FORMATS | 8 |
| AVRO | 8 |
| PARQUET | 9 |
| ORC | 9 |
| COMPARISON: WHICH FORMAT TO CHOOSE? | 10 |
| CONCLUSION | 12 |

A TIMELINE OF BIG DATA FORMATS



AN INTRODUCTION TO BIG DATA FORMATS

The goal of this whitepaper is to provide an introduction to the popular big data file formats Avro, Parquet, and ORC. We aim to understand their benefits and disadvantages as well as the context in which they were developed. By illuminating when and why to use the different formats, we hope to help you choose the format that is right for the job. The right data format is essential to achieving optimal performance and desired business outcomes.

If you're not a database expert, the choices and nuances of big data formats can be overwhelming. Increasingly, analysts, data scientists, engineers and business users need to know these formats in order to make decisions and understand workflows.

What readers can expect from this paper:

- The analyst and data scientist may gain more insight into why different formats emerged, and some of the trade-offs required when choosing a format
- The data engineer might better understand the evolution of data formats and ideal use cases for each type
- The business user will be able to understand why their analysts and engineers may prefer certain formats—and what "Avro," "Parquet," and "ORC" mean!

A Big Data Format Evaluation Framework

Evaluation Framework: Row vs. Column

HOW TO CHOOSE THE RIGHT DATA FORMAT

Before we dive into the ins and outs of different big data formats, it is helpful to establish a framework to use when evaluating them. In this paper we will use a framework that takes into account the considerations engineers and analysts might have when selecting a format, and helps to introduce basic concepts for non-technical readers. It is not meant to be comprehensive and indeed, your specific use case might need to consider other variables.

At its core, this evaluation framework has four key considerations: row or column, schema management, splitability, and compression. Let's explain each of these in turn.

CONSIDERATION ONE: ROW VS. COLUMN

Perhaps the most important consideration when selecting a big data format is whether a row or column-based format is best suited to your objectives. At the highest level, column-based storage is most useful when performing analytics queries that require only a subset of columns examined over very large data sets. If your queries require access to all or most of the columns of each row of data, row-based storage will be better suited to your needs. Let's examine these two types.

To help illustrate the differences between row and column-based data, consider this table of basic transaction data. For each transaction, we have the customer name, the product ID, sale amount, and the date.

Exhibit A

EXAMPLE: SAMPLE TRANSACTION DATA

| Customer Name | Product ID | Sale Amount | Transaction Date |
|---------------|------------|-------------|------------------|
| Emma | Prod 1 | 100.00 | 2018-04-02 |
| Liam | Prod 2 | 79.99 | 2018-04-02 |
| Noah | Prod 3 | 19.99 | 2018-04-01 |
| Olivia | Prod 2 | 79.99 | 2018-04-03 |

Let's first consider the case where the transaction data is stored in a row-based format. In this format, every row in the set has all the columns contained in the data's schema. Row-based storage is the simplest form of data table and is used in many applications, from web log files to highly-structured database systems like MySQL and Oracle.

In a database, this data would be stored by row, as follows:

```
Emma, Prod1, 100.00, 2018-04-02; Liam, Prod2, 79.99, 2018-04-02; Noah, Prod3, 19.99, 2018-04-01; Olivia, Prod2, 79.99, 2018-04-03
```

Storing data in row format is ideal when you need to access one or more entries and all or many columns for each entry.

To process this data, a computer would read this data from left to right, starting at the first row and then reading each subsequent row.

Storing data in this format is ideal when you need to access one or more entries and all or many columns for each entry. For example, let's say you're presenting customer transaction history to an account manager. The account manager needs to view all the records from her clients (e.g., the four transactions shown above), and many columns (e.g., the customer, product, price, and date columns above). That strongly suggests using row-based storage. Row-based data is most useful when you want to use many of the fields associated with an entry—and you need to access many entries.

Column-based data formats, as you might imagine, store data by column. Using our transaction data as an example, in a columnar database this data would be stored as follows:

```
Emma , Liam , Noah , Olivia ; Prod1 , Prod2 , Prod3 ; Pr  
od2 ; 100.00 , 79.99 , 19.99 , 79.99 ; 2018-04-02 , 2018-04-02 ,  
2018-04-01 , 2018-04-03
```

In columnar formats, data is stored sequentially by column, from top to bottom—not by row, left to right. Having data grouped by column makes it more efficient to easily focus computation on specific columns of data. Reading only relevant columns of data saves compute costs as irrelevant columns are ignored. Having the data stored sequentially by column allows for faster scan of the data because all relevant values are stored next to each other. There is no need to search for values within the rows. Column-based storage is also ideal for sparse data sets where you may have empty values.

Continuing with our transaction data, let's imagine a company with thousands of transactions. What is the easiest way to find the highest value sale by date? It's really easy to find the answer if you think about it from a columnar perspective—you just need to know the "Sale Amount" and "Transaction Date." If you can fetch those two columns, you can perform the operation and get the answer.

If you were to do this row-wise, then the database would have to fetch all the rows and with each row, all the columns. It would unnecessarily incur the overhead of fetching columns that were not needed for the final result. When you need to analyze select columns in the data, columnar becomes the clear choice.

Exhibit B - Please refer to Exhibit A for data details.

ROW VS. COLUMN COMPARISON

ROW-BASED REPRESENTATION



COLUMN-BASED REPRESENTATION



To compare the difference, imagine the data values from the four columns of our table represented as colored boxes, as illustrated above. To analyze "Sale Amount" (orange) and "Transaction Date" (navy blue) in a row-based format, a computer would need to read a lot of unnecessary data (the blue and green boxes) across the whole data set. That requires more time, and higher compute costs.

By contrast, the column-based representation allows a computer to skip right to the relevant data, and only read the orange and navy blue boxes. It can ignore all the blue and green values, reducing the workload and increasing efficiency. Storing the data by column allows a computer to easily skip these entries, and bypass reading the entire row. This makes computation and compression more efficient.

Evaluation Framework: Schema Evolution

CONSIDERATION TWO: SCHEMA EVOLUTION

When we talk about "schema" in a database context, we are really talking about its organization—the tables, columns, views, primary keys, relationships, etc. When we talk about schemas in the context of an individual dataset or data file, it's helpful to simplify schema further to the individual attribute level (column headers in the simplest use case). The schema will store the definition of each attribute and its type. Unless your data is guaranteed to never change, you'll need to think about schema evolution, or how your data schema changes over time. How will your file format manage fields that are added or deleted?

One of the most important considerations when selecting a data format is how it manages schema evolution. When evaluating schema evolution specifically, there are a few key questions to ask of any data format:

- How easy is it to update a schema (such as adding a field, removing or re-naming a field)?
- How will different versions of the schema "talk" to each other?

- Is it human-readable? Does it need to be?
- How fast can the schema be processed?
- How does it impact the size of data?

We'll answer these questions for each file format in the next section.

Evaluation Framework: Splitability

CONSIDERATION THREE: SPLITABILITY

By definition, big data is BIG. Datasets are commonly composed of hundreds to thousands of files, each of which may contain thousands to millions of records or more. Furthermore, these file-based chunks of data are often being generated continuously. Processing such datasets efficiently usually requires breaking the job up into parts that can be farmed out to separate processors. In fact, large-scale parallelization of processing is key to performance. Your choice of file format can critically affect the ease with which this parallelization can be implemented. For example, if each file in your dataset contains one massive XML structure or JSON record, the files will not be "splittable", i.e. decomposable into smaller records that can be handled independently.

All of the big-data formats that we'll look at in this paper support splitability, depending on the type of transactions or queries you want to perform. Most of them play a significant role in the MapReduce ecosystem, which drives the need for breaking large chunks of data into smaller, more processable ones.

Row-based formats, such as Avro, can be split along row boundaries, as long as the processing can proceed with one record at a time. If groups of records related by some particular column value are required for processing, out-of-the box partitioning may be more challenging for row-based data stored in random order.

A column-based format will be more amenable to splitting into separate jobs if the query calculation is concerned with a single column at a time. The columnar formats we discuss in this paper are row-columnar, which means they take a batch of rows and store that batch in columnar format. These batches then become split boundaries.

Evaluation Framework: Compression

CONSIDERATION FOUR: COMPRESSION

Data compression reduces the amount of information needed for the storage or transmission of a given set of data. It reduces the resources required to store and transmit data, typically saving time and money. Compression uses encoding for frequently repeating data to achieve this reduction, done at the source of the data before it is stored and/or transmitted. Simply reducing the size of a data file can be referred to as data compression.

Columnar data can achieve better compression rates than row-based data. Storing values by column, with the same type next to each other, allows you to do more efficient compression on them than if you're storing rows of data. For example, storing all dates together in memory allows for more efficient

compression than storing data of various types next to each other—such as string, number, date, string, date.

While compression may save on storage costs, it is important to also consider compute costs and resources. Chances are, at some point you will want to decompress that data for use in another application. Decompression is not free—it incurs compute costs. If and how you compress the data will be a function of how you want to optimize the compute costs vs. storage costs for your given use case.

The Formats

UNDERSTANDING THE FORMATS

For the purpose of this paper, we will focus on the most popular big data formats: Avro, Parquet, and ORC. Each evolved to better address a particular use case.

Avro

APACHE AVRO: A ROW BASED FORMAT

Apache Avro was released by the Hadoop working group in 2009. It is a row-based format that is highly splittable. The innovative, key feature of Avro is that the schema travels with data. The data definition is stored in JSON format while the data is stored in binary format, minimizing file size and maximizing efficiency. Avro features robust support for schema evolution by managing added fields, missing fields, and fields that have changed. This allows old software to read the new data and new software to read the old data—a critical feature if your data has the potential to change.

We understand this intuitively—as soon as you've finished what you're sure is the master schema to end all schemas, someone will come up with a new use case and request to add a field. This is especially true for big, distributed systems in large corporations. With Avro's capacity to manage schema evolution, it's possible to update components independently, at different times, with low risk of incompatibility. This saves applications from having to write if-else statements to process different schema versions, and saves the developer from having to look at old code to understand old schemas. Because all versions of the schema are stored in a human-readable JSON header, it's easy to understand all the fields that you have available.

Avro can support many different programming languages. Because the schema is stored in JSON while the data is in binary, Avro is a relatively compact option for both persistent data storage and wire transfer. Avro is typically the format of choice for write-heavy workloads given its easy to append new rows.

Parquet

APACHE PARQUET: A COLUMN BASED FORMAT

Launched in 2013, Parquet was developed by Cloudera and Twitter (and inspired by Google's Dremel query system) to serve as an optimized columnar data store on Hadoop. Because data is stored by columns, it can be highly compressed and splittable (for the reasons noted above). Parquet is commonly used with Apache Impala, an analytics database for Hadoop. Impala is designed for low latency and high concurrency queries on Hadoop.

The column metadata for a Parquet file is stored at the end of the file, which allows for fast, one-pass writing. Metadata can include information such as, data types, compression/encoding scheme used (if any), statistics, element names, and more.

Parquet is especially adept at analyzing wide datasets with many columns. Each Parquet file contains binary data organized by "row group." For each row group, the data values are organized by column. This enables the compression benefits that we described above. Parquet is a good choice for read-heavy workloads.

Generally, schema evolution in the Parquet file type is not an issue and is supported. However, not all systems that prefer Parquet support schema evolution optimally. For example, consider a columnar store like Impala. It is hard for that data store to support schema evolution, as the database needs to have two versions of schema (old and new) for a table.

ORC

APACHE ORC: A ROW-COLUMNAR BASED FORMAT

Optimized Row Columnar (ORC) format was first developed at Hortonworks to optimize storage and performance in Hive, a data warehouse for summarization, query and analysis that lives on top of Hadoop. Hive is designed for queries and analysis, and uses the query language HiveQL (similar to SQL).

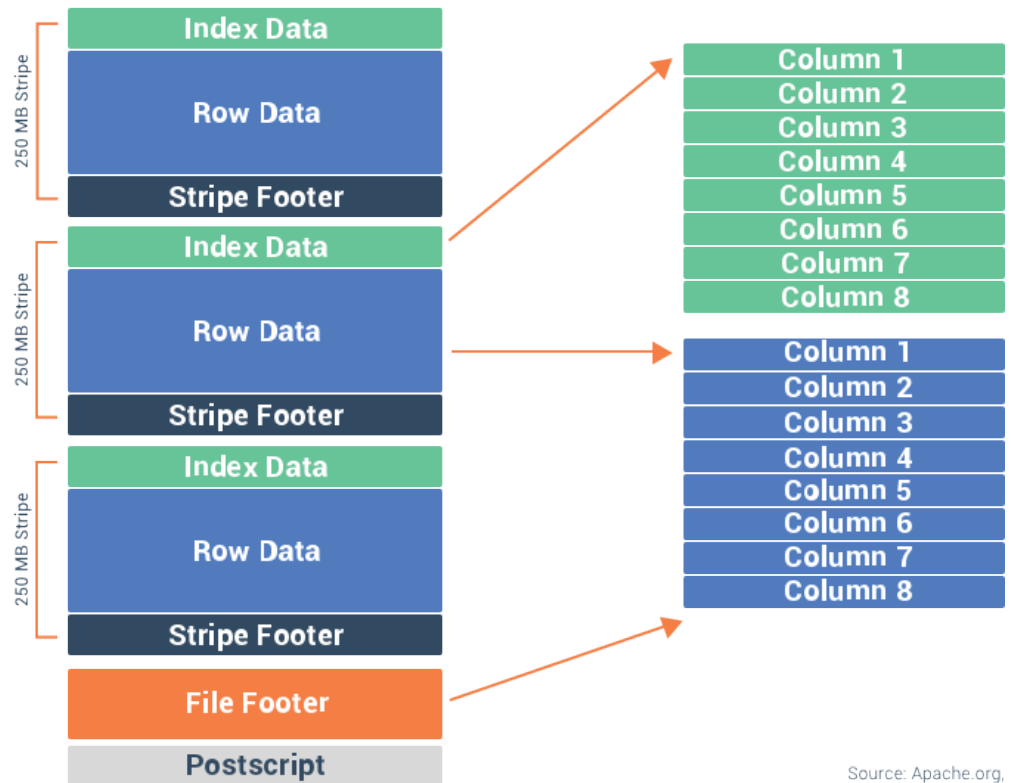
ORC files are designed for high performance when Hive is reading, writing, and processing data. ORC stores row data in columnar format. This row-columnar format is highly efficient for compression and storage. It allows for parallel processing across a cluster, and the columnar format allows for skipping of unneeded columns for faster processing and decompression. ORC files can store data more efficiently without compression than compressed text files. Like Parquet, ORC is a good option for read-heavy workloads.

This advanced level of compression is possible because of its index system. ORC files contain "stripes" of data, or 10,000 rows. These stripes are the data building blocks and independent of each other, which means queries can skip to

the stripe that is needed for any given query. Within each stripe, the reader can focus only on the columns required. The footer file includes descriptive statistics for each column within a stripe such as count, sum, min, max, and if null values are present.

Exhibit C

ORC FILE STRUCTURE



Source: Apache.org,
<https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC>

ORC is designed to maximize storage and query efficiency. According to the Apache Foundation, "Facebook uses ORC to save tens of petabytes in their data warehouse and demonstrated that ORC is significantly faster than RC File or Parquet."

Similar to Parquet, schema evolution is supported by the ORC file format, but its efficacy is dependent on what the data store supports. Recent advances have been made in Hive that allow for appending columns, type conversion, and name mapping.

Comparison










WHICH FORMAT TO CHOOSE?

Now that we've described in some detail how Avro, Parquet, and ORC function, it can be useful to compare them. Referring back to the evaluation framework we outlined at the beginning of the paper, we compared Avro, Parquet and ORC against their support for splitability, compression, and schema evolution. Based on analysis of each format's capabilities, its relative score is noted below.

Depending on the nature of your data set and analytic objectives, you will likely value some of those features more than others. Because all of these formats are Apache open source projects, they are constantly being updated to support new features and functionality. It's always worthwhile to consult the latest release for specifics.

Exhibit C

BIG DATA FORMATS COMPARISON

| | Avro | Parquet | ORC |
|---------------------------|--|--|--|
| Schema Evolution Support |  |  |  |
| Compression |  |  |  |
| Splitability |  |  |  |
| Most Compatible Platforms | Kafka, Druid | Impala, Arrow Drill, Spark | Hive, Presto |
| Row or Column | Row | Column | Column |
| Read or Write | Write | Read | Read |

Source: Nexla analysis, April 2018

The first determination will likely be if your data is better suited to be stored by row or by column. Transactional data, event-level data, and use cases for which you will need to leverage many columns are best-suited to row-based data. If that's the case, Avro is likely the best choice. Avro is typically the choice for more write-heavy workloads, since its row-based format is easier to append (similar to a traditional database structure).

However, if you know your data is best suited to a columnar format, the question will become Parquet or ORC. In addition to the relative importance of splitability, compression, and schema evolution support, consideration must be given to your existing infrastructure. ORC maximizes performance on Hive. Both formats offer benefits and it will likely come down to which system you have access to and are most familiar with. Columnar formats are the choice for read-heavy workloads, owing to the efficiency gains from splitability and compression we discussed in this paper.

Conclusion

CHOOSE THE RIGHT FORMAT FOR THE JOB

In this paper we've discussed a helpful framework for evaluating the big data formats Avro, Parquet, and ORC, an overview of how each format was developed, and their strengths.

In an ideal world, you'd always choose the data format that was right for your use case and infrastructure. However, sometimes we don't get to decide how we receive the data we need to work with. Data may be coming in any format—CSV, JSON, XML, or one of the big data formats we discussed. Converting data from the incoming format to the one optimally suited for a specific processing need can be a laborious process. It may include detecting, evolving, or modifying schemas, combining or splitting files, and applying partitioning. This is in addition to managing the difference in frequency of incoming data to the desired frequency of output. All things considered, converting data formats can significantly increase workloads.

Nexla makes these data format conversions easy. Point Nexla to any source—such as a datastore with Avro files—and Nexla can extract, transform, and convert the data into the preferred format. Companies use this capability to convert JSON CloudTrail logs into Parquet for use in Amazon Athena, or ingest Avro event data to process into database tables. Perhaps your system outputs data into Avro but you have a machine learning project that could benefit from Parquet. No matter how you're getting the data, with Nexla you can easily create the pipeline to convert it into the format that works for you.

About Nexla:

Nexla is a scalable Data Operations platform that can manage inter-company data collaboration, securely and in real-time. Nexla automates DataOps so companies can quickly derive value from their data, with minimal engineering required. Our secure platform runs in the cloud or on-premise. It allows business users to send, receive, transform, and monitor data in their preferred format via an easy to use web interface.