

Serialization Formats

Many serialization algorithms and formats are available to data engineers. While the abundance of options is a significant source of pain in data engineering, they are also a massive opportunity for performance improvements. We've sometimes seen job performance improve by a factor of 100 simply by switching from CSV to Parquet serialization. As data moves through a pipeline, engineers will also manage reserialization—conversion from one format to another. Sometimes data engineers have no choice but to accept data in an ancient, nasty form; they must design processes to deserialize this format and handle exceptions, and then clean up and convert data for consistent, fast downstream processing and consumption.

Row-Based Serialization

As its name suggests, *row-based serialization* organizes data by row. CSV format is an archetypal row-based format. For semistructured data (data objects that support nesting and schema variation), row-oriented serialization entails storing each object as a unit.

CSV: The nonstandard standard

We discussed CSV in [Chapter 7](#). CSV is a serialization format that data engineers love to hate. The term *CSV* is essentially a catchall for delimited text, but there is flexibility in conventions of escaping, quote characters, delimiter, and more.

Data engineers should avoid using CSV files in pipelines because they are highly error-prone and deliver poor performance. Engineers are often required to use CSV format to exchange data with systems and business processes outside their control. CSV is a common format for data archival. If you use CSV for archival, include a complete technical description of the serialization configuration for your files so that future consumers can ingest the data.

XML

Extensible Markup Language (XML) was popular when HTML and the internet were new, but it is now viewed as legacy; it is generally slow to deserialize and serialize for data engineering applications. XML is another standard that data engineers are often forced to interact with as they exchange data with legacy systems and software. JSON has largely replaced XML for plain-text object serialization.

JSON and JSONL

JavaScript Object Notation (JSON) has emerged as the new standard for data exchange over APIs, and it has also become an extremely popular format for data storage. In the context of databases, the popularity of JSON has grown apace with the rise of MongoDB and other document stores. Databases such as Snowflake, BigQuery, and SQL Server also offer extensive native support, facilitating easy data exchange between applications, APIs, and database systems.

JSON Lines (JSONL) is a specialized version of JSON for storing bulk semistructured data in files. JSONL stores a sequence of JSON objects, with objects delimited by line breaks. From our perspective, JSONL is an extremely useful format for storing data right after it is ingested from API or applications. However, many columnar formats offer significantly better performance. Consider moving to another format for intermediate pipeline stages and serving.

Avro

Avro is a row-oriented data format designed for RPCs and data serialization. Avro encodes data into a binary format, with schema metadata specified in JSON. Avro is popular in the Hadoop ecosystem and is also supported by various cloud data tools.

Columnar Serialization

The serialization formats we've discussed so far are row-oriented. Data is encoded as complete relations (CSV) or documents (XML and JSON), and these are written into files sequentially.

With *columnar serialization*, data organization is essentially pivoted by storing each column into its own set of files. One obvious advantage to columnar storage is that it allows us to read data from only a subset of fields rather than having to read full rows at once. This is a common scenario in analytics applications and can dramatically reduce the amount of data that must be scanned to execute a query.

Storing data as columns also puts similar values next to each other, allowing us to encode columnar data efficiently. One common technique involves looking for repeated values and tokenizing these, a simple but highly efficient compression method for columns with large numbers of repeats.

Even when columns don't contain large numbers of repeated values, they may manifest high redundancy. Suppose that we organized customer support messages into a single column of data. We likely see the same themes and verbiage again and again across these messages, allowing data compression algorithms to realize a high ratio. For this reason, columnar storage is usually combined with compression, allowing us to maximize disk and network bandwidth resources.

Columnar storage and compression come with some disadvantages too. We cannot easily access individual data records; we must reconstruct records by reading data from several column files. Record updates are also challenging. To change one field in one record, we must decompress the column file, modify it, recompress it, and write it back to storage. To avoid rewriting full columns on each update, columns are broken into many files, typically using partitioning and clustering strategies that organize data according to query and update patterns for the table. Even so, the overhead for updating a single row is horrendous. Columnar databases are a terrible fit for transactional workloads, so transactional databases generally utilize some form of row- or record-oriented storage.

Parquet

Parquet stores data in a columnar format and is designed to realize excellent read and write performance in a data lake environment. Parquet solves a few problems that frequently bedevil data engineers. Parquet-encoded data builds in schema information and natively supports nested data, unlike CSV. Furthermore, Parquet is portable; while databases such as BigQuery and Snowflake serialize data in proprietary columnar formats and offer excellent query performance on data stored internally, a huge performance hit occurs when interoperating with external tools. Data must be deserialized, reserialized into an exchangeable format, and exported to use data lake tools such as Spark and Presto. Parquet files in a data lake may be a superior option to proprietary cloud data warehouses in a polyglot tool environment.

Parquet format is used with various compression algorithms; speed optimized compression algorithms such as Snappy (discussed later in this appendix) are especially popular.

ORC

Optimized Row Columnar (ORC) is a columnar storage format similar to Parquet. ORC was very popular for use with Apache Hive; while still widely used, we generally see it much less than Apache Parquet, and it enjoys somewhat less support in modern cloud ecosystem tools. For example, Snowflake and BigQuery support Parquet file import and export; while they can read from ORC files, neither tool can export to ORC.

Apache Arrow or in-memory serialization

When we introduced serialization as a storage raw ingredient at the beginning of this chapter, we mentioned that software could store data in complex objects scattered in memory and connected by pointers, or more orderly, densely packed structures such as Fortran and C arrays. Generally, densely packed in-memory data structures were limited to simple types (e.g., INT64) or fixed-width data structures (e.g., fixed-width strings). More complex structures (e.g., JSON documents) could not be densely stored in memory and required serialization for storage and transfer between systems.

The idea of [Apache Arrow](#) is to rethink serialization by utilizing a binary data format that is suitable for both in-memory processing and export.¹ This allows us to avoid the overhead of serialization and deserialization; we simply use the same format for in-memory processing, export over the network, and long-term storage. Arrow relies on columnar storage, where each column essentially gets its own chunks of memory. For nested data, we use a technique called *shredding*, which maps each location in the schema of JSON documents into a separate column.

This technique means that we can store a data file on disk, swap it directly into program address space by using virtual memory, and begin running a query against the data without deserialization overhead. In fact, we can swap chunks of the file into memory as we scan it, and then swap them back out to avoid running out of memory for large datasets.

One obvious headache with this approach is that different programming languages serialize data in different ways. To address this issue, the Arrow Project has created software libraries for a variety of programming languages (including C, Go, Java, JavaScript, MATLAB, Python, R, and Rust) that allow these languages to interoperate with Arrow data in memory. In some cases, these libraries use an interface between the chosen language and low-level code in another language (e.g., C) to read and write from Arrow. This allows high interoperability between languages without extra serialization overhead. For example, a Scala program can use the Java library to write arrow data and then pass it as a message to a Python program.

Arrow is seeing rapid uptake with a variety of popular frameworks such as Apache Spark. Arrow has also spanned a new data warehouse product; [Dremio](#) is a query engine and data warehouse built around Arrow serialization to support fast queries.

Hybrid Serialization

We use the term *hybrid serialization* to refer to technologies that combine multiple serialization techniques or integrate serialization with additional abstraction layers, such as schema management. We cite as examples Apache Hudi and Apache Iceberg.

Hudi

Hudi stands for *Hadoop Update Delete Incremental*. This table management technology combines multiple serialization techniques to allow columnar database performance for analytics queries while also supporting atomic, transactional updates. A typical Hudi application is a table that is updated from a CDC stream from a transactional application database. The stream is captured into a row-oriented serialization format, while the bulk of the table is retained in a columnar format. A query runs over both columnar and row-oriented files to return results for the current state of the table. Periodically, a repacking process runs that combines the row and columnar files into updated columnar files to maximize query efficiency.

Iceberg

Like Hudi, Iceberg is a table management technology. Iceberg can track all files that make up a table. It can also track files in each table snapshot over time, allowing table time travel in a data lake environment. Iceberg supports schema evolution and can readily manage tables at a petabyte scale.