Technical Disclosure Commons

Defensive Publications Series

September 2022

Deserializing and Exposing In-memory Columnar OLAP/OLTP Data

Prashant Mishra

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation
Mishra, Prashant, "Deserializing and Exposing In-memory Columnar OLAP/OLTP Data", Technical Disclosure Commons, (September 18, 2022)
https://www.tdcommons.org/dpubs_series/5378

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.

Deserializing and Exposing In-memory Columnar OLAP/OLTP Data

ABSTRACT

This disclosure describes high-performance techniques of using a client library to access data from OLAP or OLTP databases. Data is transported between the database and the client library in a columnar in-memory format such as Apache Arrow over a binary protocol such as RPC. RPC is significantly faster than REST when receiving data due to the use of protocol Buffers and HTTP/2. In addition, Apache Arrow is zero-copy and does efficient in-memory computations. The stream of data is processed in separate parser threads, thus maximizing concurrency and parallel processing. This makes the client library an order of magnitude faster than the legacy REST based implementations. The client library deserializes columnar in-memory data into rows such that callers of the client can retrieve data in a familiar backward-compatible format, e.g., java.sql.ResultSet (if the client library is written in Java), which acts as a data abstraction layer.

KEYWORDS

- Online analytical system (OLAP)
- Online transaction processing (OLTP)
- Serialization
- Deserialization
- SERDES
- Java database connectivity (JDBC)
- Remote procedure call (RPC)
- Columnar data
- In-memory data

BACKGROUND

Online analytical processing (OLAP) is a computing technique that enables client applications to swiftly and at scale extract data and execute multi-dimensional analytical queries on databases. Such queries are typically used in business intelligence and reporting applications. Online transaction processing (OLAP) databases are frequently used to carry out very large numbers of day-to-day transactions, e.g., online booking, ticket-booking, etc. Client libraries typically access OLAP or online transaction processing (OLTP) databases via a representational state transfer (REST) API. For example, such an API is essentially a JavaScript object notation (JSON) payload over a secure connection (e.g., a HTTPS connection). This is suboptimal for high-throughput applications.

DESCRIPTION

This disclosure describes high-performance techniques of using a client library to access data from OLAP or OLTP databases. The client library can be written in Java or other languages. Compared to legacy techniques that use REST endpoints, data access performance is improved by an order of magnitude even as concurrency is optimized and JDBC (or equivalent) data abstraction layer for data consumption is retained. The client library deserializes columnar, in-memory data into rows. The client library uses a separate parser thread for deserializing columnar, in-memory data into rows and it can use one parser thread per stream of data, thus maximizing concurrency and parallel processing of multiple data streams.

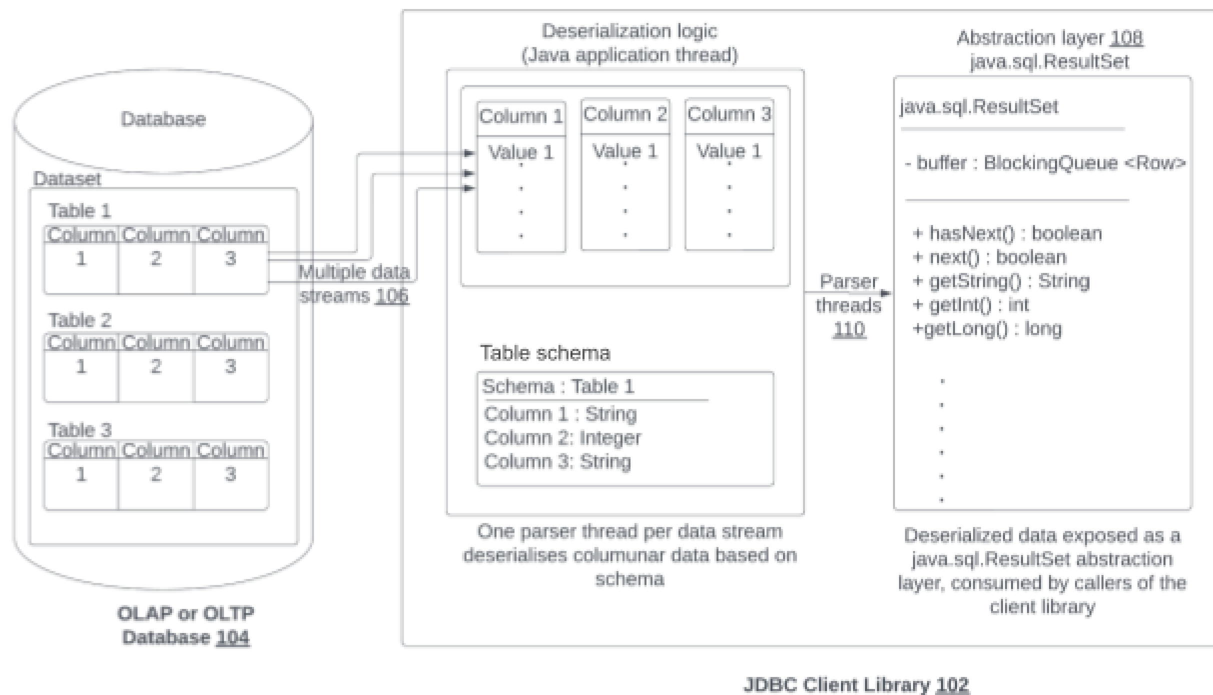

Fig. 1: Deserializing and exposing in-memory columnar OLAP/OTAP data

Fig. 1 illustrates deserializing and exposing in-memory columnar OLAP/OLTP data. A client library (102) reads data from an OLAP or OLTP database (104) in an in-memory columnar data format such as Apache Arrow and exposes it as JDBC at the abstraction layer. The client library can be written in Java or other languages. Multiple streams of columnar data (106) are transferred from the OLAP/OLTP database to the client library in a binary protocol such as RPC (remote procedure call). If the data is ordered or sorted, just one stream is read. The client library translates the data from columnar, in-memory data format to a conventional, in-memory, row-based representation using multiple concurrent parser threads (110) to maximize throughput. The client library can use one parser thread per stream of in-memory columnar data.

The client library uses the table schema of the database to identify different column labels in the input columnar in-memory data. A schema of a given query includes a definition of the columns used. It has information such as the data type of the columns along with other metadata

about the columns (such as the column's nullability) for the columns used. The schema is also used to map a given column with its index number.

Based on the schema of the database, the parser thread deserializes columnar data into rows and populates a shared blocking queue with a certain number of records. The deserialized data is exposed, e.g., as a java.sql.ResultSet (if the client library is written in Java), which acts as an abstraction layer (108). The deserialized data is consumed by the callers of the client library. The shared blocking queue enables synchronization of the production of data (e.g., imported from the database) and the consumption of data (e.g., consumed by the callers of the client library).

The abstraction layer can have a number of function calls, e.g., hasNext(), next(), getString(), getInt(), getLong(), etc., to enable the caller of the client library to access database entries as they transform to rows. For example, the hasNext() function, which returns a Boolean, can be used to query if the next row is now available for consumption. If hasNext() returns true, then the function next() can be used to retrieve the latest row from the blocking queue. If hasNext() returns false, then all rows thus far received from the database have been read, e.g., no new rows are available for the given query. However, if hasNext() returns true, and all the rows from the blocking queue has already been read but there are more record(s) yet to be received from the database, the consumer waits at the next() call until the next row arrives

RPC is significantly faster than REST when receiving data mainly due to the use of protocol buffers and HTTP/2. In addition, Apache Arrow is zero-copy and does efficient in-memory computations. This makes the client library an order of magnitude faster than legacy REST based implementations. By deserializing columns to rows, callers of the client library can retrieve data in a familiar backward-compatible format.

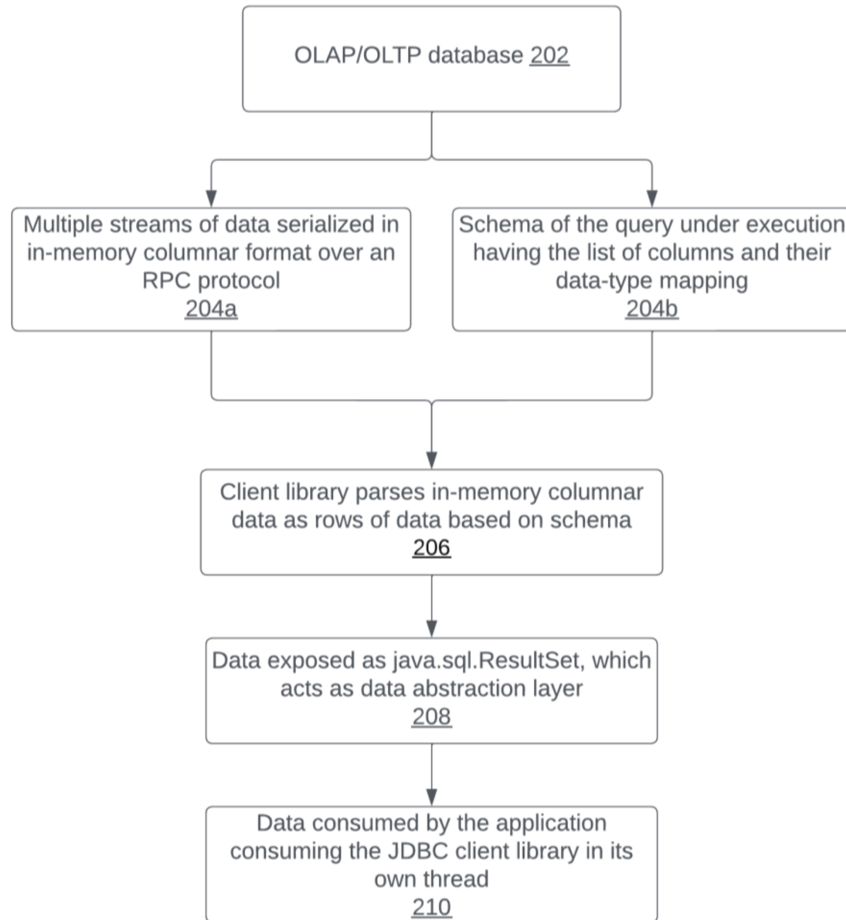

Fig. 2: Workflow

Fig. 2 illustrates the workflow. An OLAP/OLTP database (202) includes datasets with tables defined by their schema, which can include a list of columns, their data-type mapping, etc. Based on the schema of the query under execution (204b), multiple streams of data serialized in in-memory columnar format are transferred from the OLAP/OLTP database to the client library over a binary protocol such as RPC (204a).

If the data is ordered or sorted, just one stream is used. The client library parses in-memory columnar data into rows of data based on the schema (206) of the query. Data is exposed, e.g., as java.sql.ResultSet (if the client library is written in Java), which acts as a data abstraction layer (208). Data is consumed by the application in its own thread (210). The

techniques apply across cloud client libraries and database products regardless of programming language.

CONCLUSION

This disclosure describes high-performance techniques of using a client library to access data from OLAP or OLTP databases. Data is transported between the database and the client library in a columnar in-memory format such as Apache Arrow over a binary protocol such as RPC. RPC is significantly faster than REST when receiving data due to the use of protocol Buffers and HTTP/2. In addition, Apache Arrow is zero-copy and does efficient in-memory computations. The stream of data is processed in separate parser threads, thus maximizing concurrency and parallel processing. This makes the client library an order of magnitude faster than the legacy REST based implementations. The client library deserializes columnar in-memory data into rows such that callers of the client can retrieve data in a familiar backward-compatible format, e.g., java.sql.ResultSet (if the client library is written in Java), which acts as a data abstraction layer.