A journey with Apache Arrow (part 1)

Basics and common transformations

Author: Laurent Querel - F5 Inc.

Apache Arrow is a technology widely adopted in big data, analytics, and machine learning applications. This article discusses our journey with Arrow, specifically its application to telemetry, and the challenges we encountered while optimizing the OpenTelemetry protocol to significantly reduce bandwidth costs. The promising results we achieved inspired us to share our insights. This article specifically focuses on transforming data from an XYZ format into an efficient Arrow representation that optimizes both compression ratio, transport, and data processing. Our benchmarks thus far have shown promising results, with compression ratio improvements ranging from 1.5x to 6x, depending on the data type (metrics, logs, traces) and distribution. The approaches presented for addressing these challenges may be applicable to other Arrow domains as well. This article serves as the first installment in a two-part series.

What is Apache Arrow

Apache Arrow is an open-source project offering a standardized, language-agnostic in-memory format for representing structured and semi-structured data. This enables data sharing and zero-copy data access between systems, eliminating the need for serialization and deserialization when exchanging datasets between varying CPU architectures and programming languages. Furthermore, Arrow features an extensive set of high-performance, parallel, and vectorized kernel functions designed for efficiently processing massive amounts of columnar data. These features make Arrow an appealing technology for big data processing, data transport, analytics, and machine learning applications. The growing number of products and open-source projects that have adopted Apache Arrow at their core or offer Arrow support reflects the widespread recognition and appreciation of its benefits (refer to this article for an in-depth overview of the Arrow

ecosystem and adoption). Over 11,000 GitHub users support this project, and 840+ are contributors who make this project an undeniable success.

Very often people ask about the differences between Arrow and Apache Parquet or other columnar file formats. Arrow is designed and optimized for in-memory processing, while Parquet is tailored for disk-based storage. In reality, these technologies are complementary, with bridges existing between them to simplify interoperability. In both cases, data is represented in columns to optimize access, data locality and compressibility. However, the tradeoffs differ slightly. Arrow prioritizes data processing speed over the optimal data encoding. Complex encodings that don't benefit from SIMD instruction sets are generally not natively supported by Arrow, unlike formats such as Parquet. Storing data in Parquet format and processing and transporting it in Arrow format has become a prevalent model within the big data community.

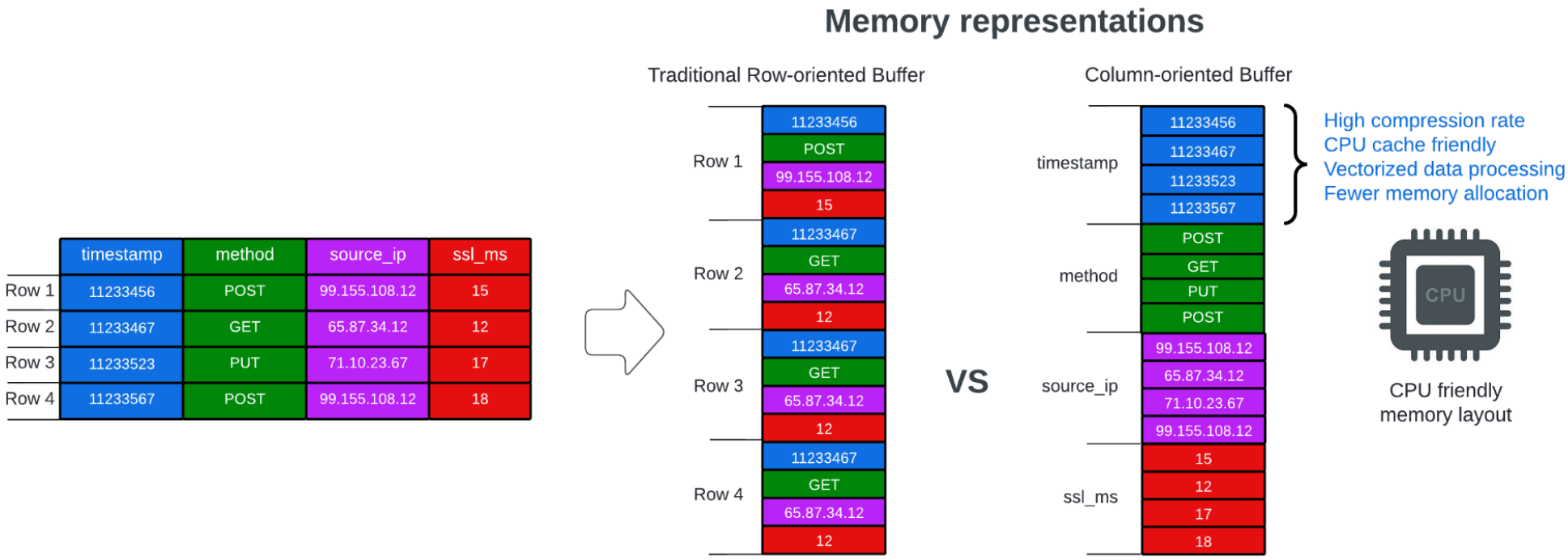

Fig 1: Memory representations: row vs columnar data.

Figure 1 illustrates the differences in memory representation between row-oriented and column-oriented approaches. The column-oriented approach groups data from the same column in a continuous memory area, which facilitates parallel processing (SIMD) and enhances compression performance.

Why are we interested in Apache Arrow

At F5, we've adopted OpenTelemetry (OTel) as the standard for all telemetry across our products, such as BIGIP and NGINX. These products may generate large volumes of metrics and logs for various reasons, from performance evaluation to forensic purposes. The data produced by these systems is typically centralized and processed in dedicated systems. Transporting and processing this data accounts for a significant portion of the cost associated with telemetry pipelines. In this context, we became interested in Apache Arrow. Instead of reinventing yet another telemetry solution, we decided to invest in the OpenTelemetry project, working on improvements to the protocol to significantly increase its efficiency with high telemetry data volumes. We collaborated with Joshua MacDonald from Lightstep to integrate these optimizations into an experimental OTel collector and are currently in discussions with the OTel technical committee to finalize a code donation.

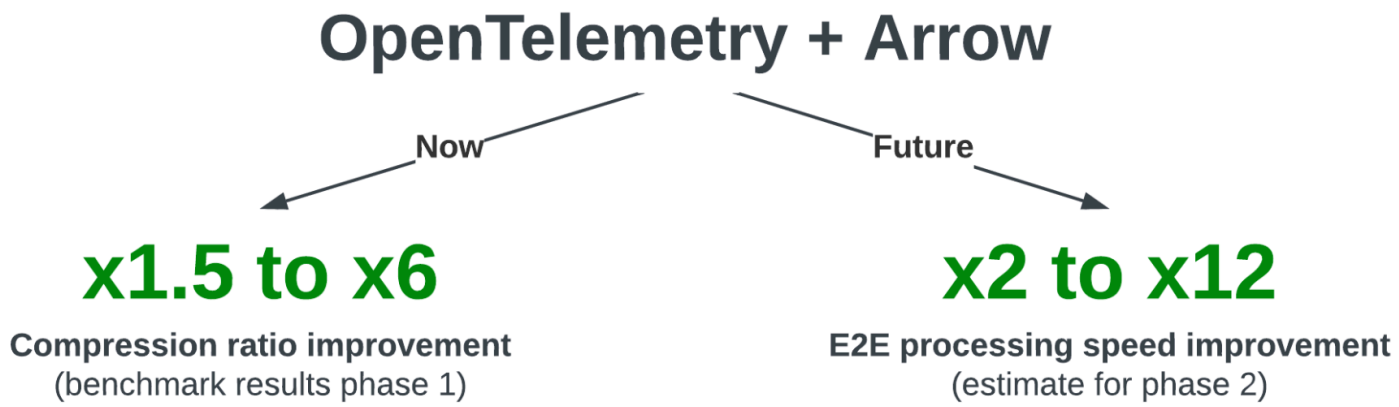

Fig 2: Performance improvement in the OpenTelemetry Arrow experimental project.

This project has been divided into two phases. The first phase, which is nearing completion, aims to enhance the protocol's compression ratio. The second phase, planned for the future, focuses on improving end-to-end performance by incorporating Apache Arrow throughout

all levels, eliminating the need for conversion between old and new protocols. The results so far are promising, with our benchmarks showing compression ratio improvements ranging from x1.5 to x6, depending on the data type (metrics, logs, traces) and distribution. For the second phase, our estimates suggest that data processing acceleration could range from x2 to x12, again depending on the data's nature and distribution. For more information, we encourage you to review the specifications and the reference implementation.

Arrow relies on a schema to define the structure of data batches that it processes and transports. The subsequent sections will discuss various techniques that can be employed to optimize the creation of these schemas.

How to leverage Arrow to optimize network transport cost

Apache Arrow is a complex project with a rapidly evolving ecosystem, which can sometimes be overwhelming for newcomers. Fortunately the Arrow community has published three introductory articles (articles 1, 2, and 3) that we recommend for those interested in exploring this technology.

This article primarily focuses on transforming data from an XYZ format into an efficient Arrow representation that optimizes both compression ratio and data processing. There are numerous approaches to this transformation, and we will examine how these methods can impact compression ratio, CPU usage, and memory consumption during the conversion process, among other factors.

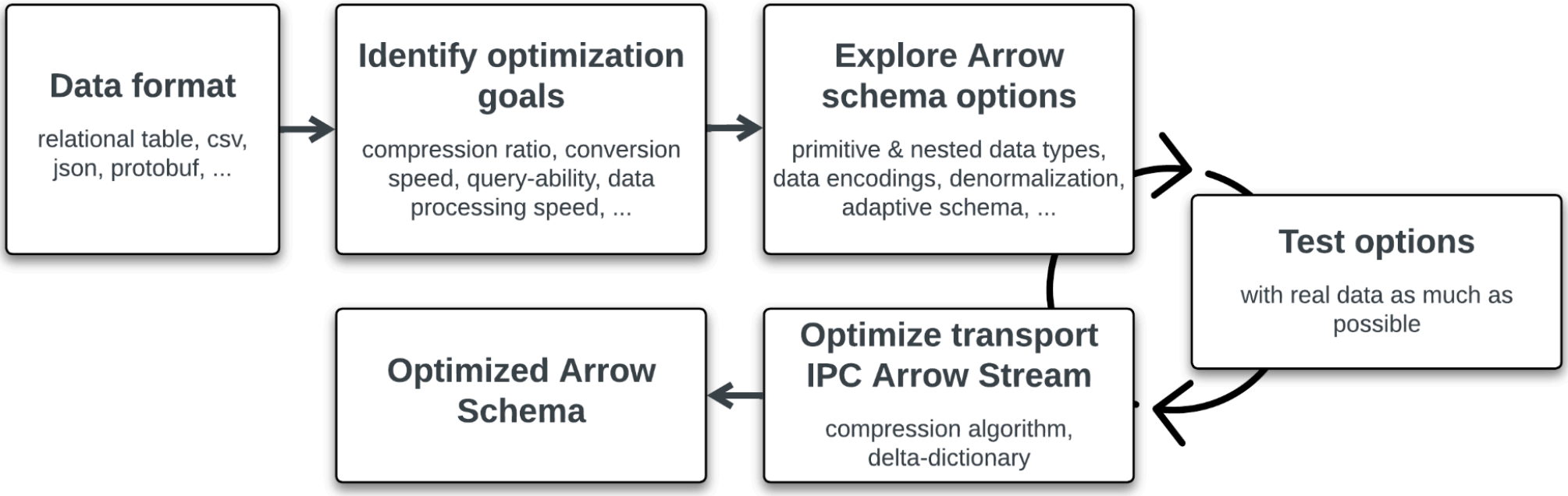

Fig 3: Optimization process for the definition of an Arrow schema.

The complexity of your initial model significantly impacts the Arrow mapping choices you need to make. To begin, it's essential to identify the properties you want to optimize for your specific context. Compression rate, conversion speed, memory consumption, speed and ease of use of the final model, compatibility, and extensibility are all factors that can influence your final mapping decisions. From there, you must explore multiple alternative schemas.

The choice of the Arrow type and data encoding for each individual field will affect the performance of your schema. There are various ways to represent hierarchical data or highly dynamic data models, and multiple options need to be evaluated in coordination with the

configuration of the transport layer. This transport layer should also be carefully considered. Arrow supports compression mechanisms and dictionary deltas that may not be active by default.

After several iterations of this process, you should arrive at an optimized schema that meets the goals you initially set. It's crucial to compare the performance of your different approaches using real data, as the distribution of data in each individual field may influence whether you use dictionary encoding or not. We will now examine these choices in greater detail throughout the remainder of this article.

Arrow data type selection

The principles of selecting an Arrow data type are quite similar to those used when defining a data model for databases. Arrow supports a wide range of data types. Some of these types are supported by all implementations, while others are only available for languages with the strongest Arrow community support (see this page for a comparison matrix of the different implementations). For primitive types, it is generally preferable to choose the type that offers the most concise representation and is closest to the semantics of your initial field. For example, while it's possible to represent a timestamp with an int64, it's more advantageous to use the native Arrow Timestamp type. This choice isn't due to a more efficient binary representation, but rather because it will be easier to process and manipulate in your pipeline. Query engines such as DataFusion offer dedicated timestamp handling functions for columns of this type. The same choices can be made for primitive types such as date, time, duration, and interval. However, if your project requires maximum compatibility, it may be crucial in some cases to favor types with universal support instead of the most optimal type in terms of memory occupation.

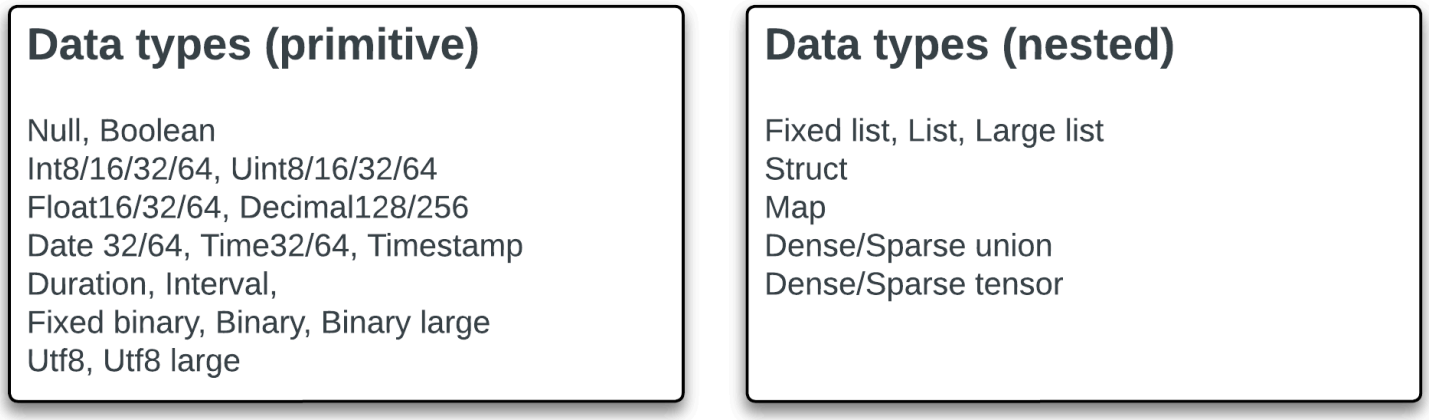

Fig 4: Data types supported by Apache Arrow.

When selecting the Arrow data type, it's important to consider the size of the data before and after compression. It's quite possible that the size after compression is the same for two different types, but the actual size in memory may be two, four, or even eight times larger (e.g., uint8 vs. uint64). This difference will impact your ability to process large batches of data and will also significantly influence the speed of processing these data in memory (e.g., cache optimization, SIMD instruction efficiency).

It's also possible to extend these types using an extension type mechanism that builds upon one of the currently supported primitive types while adding specific semantics. This extension mechanism can simplify the use of this data in your own project, while remaining transparent to intermediate systems that will interpret this data as a basic primitive type.

There are some variations in the encoding of primitive types, which we will explore next.

Data encoding

Another crucial aspect of optimizing your Arrow schema is analyzing the cardinality of your data. Fields that can have only a limited number of values will typically be more efficiently represented with a dictionary encoding.

The maximum cardinality of a field determines the data type characteristics of your dictionary. For instance, for a field representing the status code of an HTTP transaction, it's preferable to use a dictionary with an index of type ‘uint8’ and a value of type ‘uint16’ (notation: ‘Dictionary<uint8, uint16>’). This consumes less memory because the main array will be of type ‘[]uint8’. Even if the range of possible values is greater than 255, as long as the number of distinct values does not exceed 255, the representation remains efficient. Similarly, the representation of a ‘user-agent’ will be more efficient with a dictionary of type ‘Dictionary<uint16, string>’ (see figure 5). In this case, the main array will be of type ‘uint16’, allowing a compact representation in memory and during transfers at the cost of an indirection during reverse conversion.

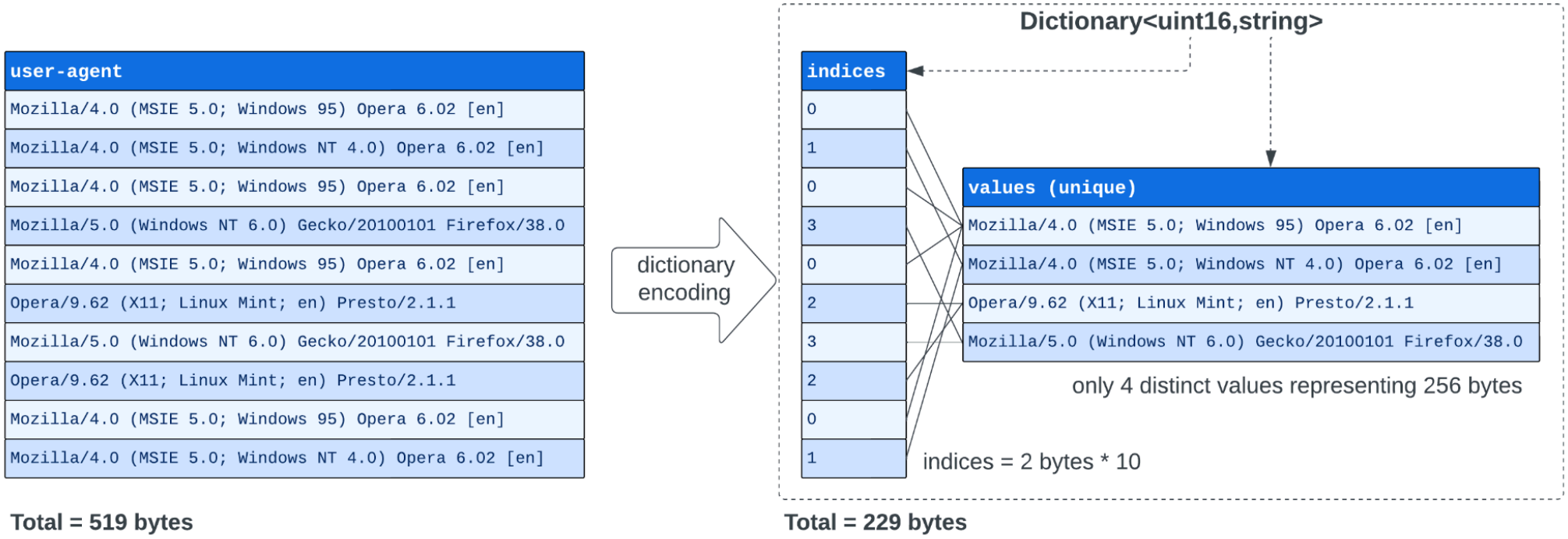

Fig 5: Dictionary encoding.

Dictionary encoding is highly flexible in Apache Arrow, allowing the creation of encodings for any Arrow primitive type. The size of the indices can also be configured based on the context.

In general, it is advisable to use dictionaries in the following cases:

- Representation of enumerations
- Representation of textual or binary fields with a high probability of having redundant values.

- Representation of fields with cardinalities known to be below 2^16 or 2^32.

Sometimes, the cardinality of a field is not known a priori. For example, a proxy that transforms a data stream from a row-oriented format into a series of columnar-encoded batches (e.g., OpenTelemetry collector) may not be able to predict in advance whether a field will have a fixed number of distinct values. Two approaches are possible:

1. a conservative approach using the largest data type (e.g., ‘int64’, ‘string’, etc., instead of dictionary),
2. an adaptive approach that modifies the schema on the fly based on the observed cardinality of the field(s). In this second approach, without cardinality information, one can optimistically start by using a ‘Dictionary<uint8, original-field-type>’ dictionary, then detect a potential dictionary overflow during conversion, and change the schema to a ‘Dictionary<uint16, original-field-type>’ in case of an overflow. This technique of automatic management of dictionary overflows will be presented in greater detail in a future article.

Recent advancements in Apache Arrow include the implementation of run-end encoding, a technique that efficiently represents data with sequences of repeated values. This encoding method is particularly beneficial for handling data sets containing long stretches of identical values, as it offers a more compact and optimized representation.

In conclusion, dictionary encoding not only occupies less space in memory and during transfers but also significantly improves the compression ratio and data processing speed. However, this type of representation requires indirection when extracting the initial values (although this isn’t always necessary, even during some data processing operations). Additionally, it is important to manage dictionary index overflow, especially when the encoded field doesn't have a well-defined cardinality.

Hierarchical data

Basic hierarchical data structures translate relatively well into Arrow. However, as we will see, there are some complications to handle in more general cases (see figure 6). While Arrow schemas do support nested structures, maps, and unions, some components of the Arrow ecosystem do not fully support them, which can make these Arrow data types unsuitable for certain scenarios. Additionally, unlike most

languages and formats, such as Protobuf, Arrow doesn't support the concept of a recursively defined schema. An Arrow schema is static in its definition, and the depth of its nested elements must be known in advance. There are multiple strategies to work around this limitation and we'll explore these in the following sections.

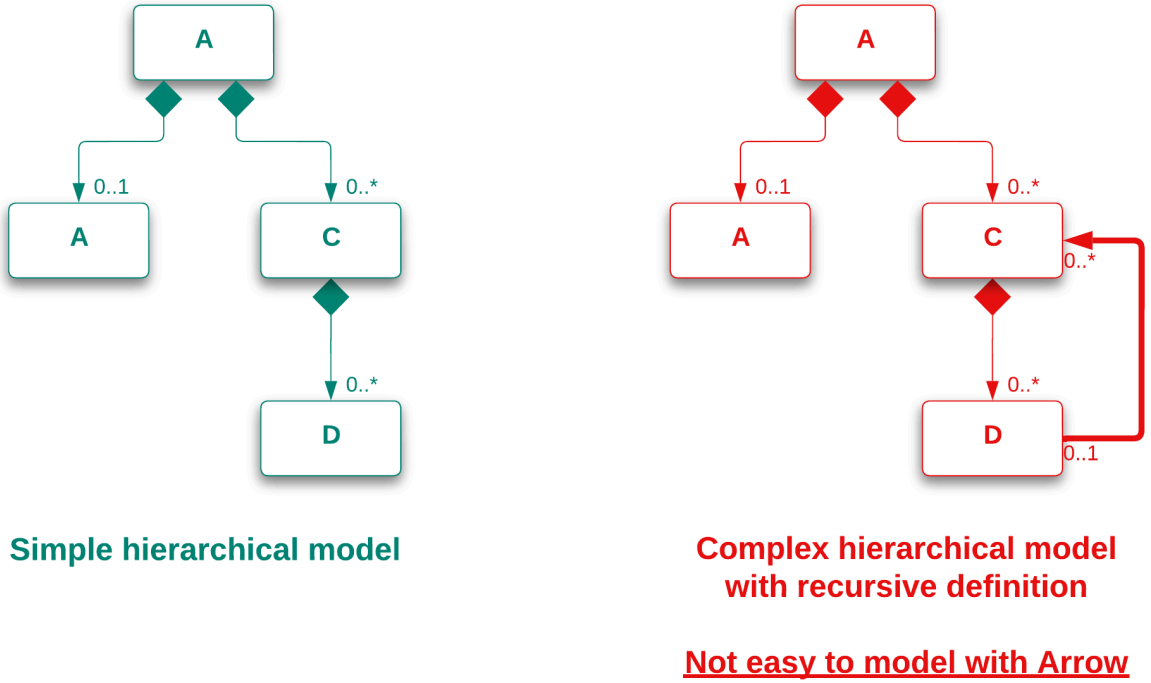

Fig 6: simple vs complex data model

Natural representation

The most straightforward and intuitive approach to representing a simple hierarchical data model is to use Arrow's list, map, and union data types. However, it's important to note that some of these data types are not fully supported throughout the entire Arrow ecosystem. For example, the conversion of unions to Parquet is not directly supported and requires a transformation step (see denormalization & flattening representation to decompose a sparse union into a nullable struct and type ids column). Similarly, lists and maps are not yet supported in DataFusion version 20 (nested structures are partially supported).

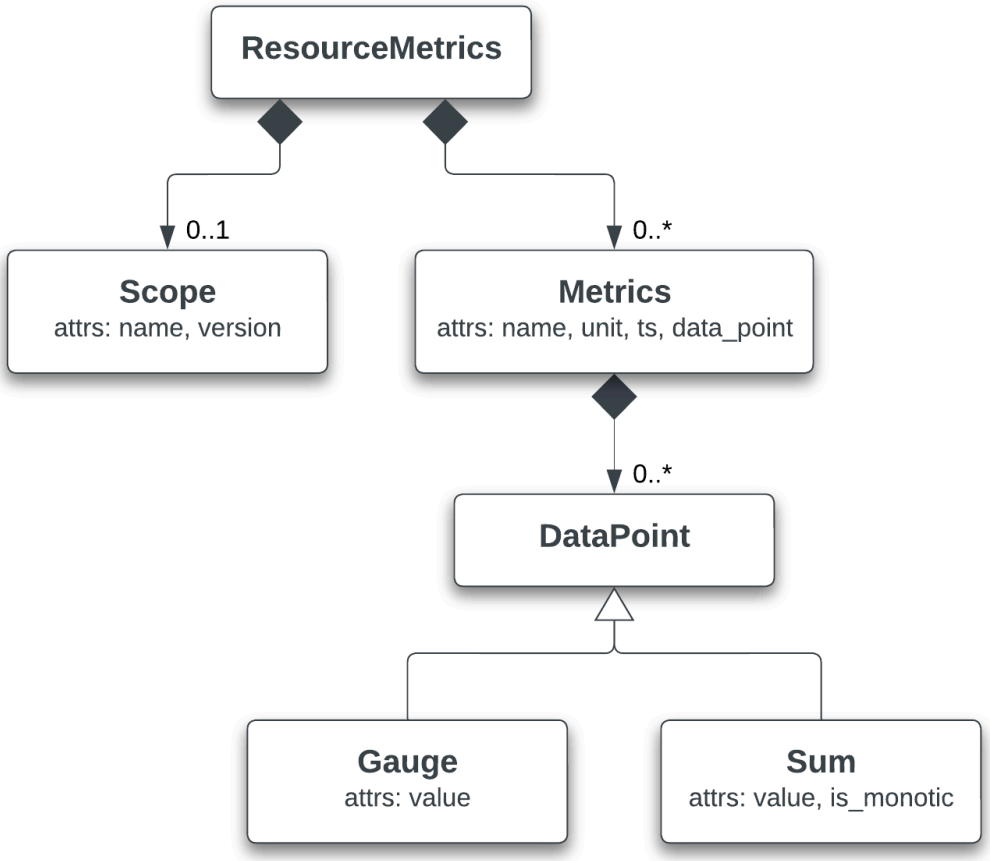

Fig 7: initial data model

The following example is a Go program snippet of an Arrow schema using these different data types to represent the model above.

```
import "github.com/apache/arrow/go/v11/arrow"

const (
  GaugeMetricCode arrow.UnionTypeCode = 0
  SumMetricCode   arrow.UnionTypeCode = 1
)

var (
  // uint8Dictionary represent a Dictionary<Uint8, String>
  uint8Dictionary = &arrow.DictionaryType{
     IndexType: arrow.PrimitiveTypes.Uint8,
     ValueType: arrow.BinaryTypes.String,
  }
  // uint16Dictionary represent a Dictionary<Uint16, String>
  uint16Dictionary = &arrow.DictionaryType{
     IndexType: arrow.PrimitiveTypes.Uint16,
     ValueType: arrow.BinaryTypes.String,
  }

  Schema = arrow.NewSchema([]arrow.Field{
     {Name: "resource_metrics", Type: arrow.ListOf(arrow.StructOf([]arrow.Field{
        {Name: "scope", Type: arrow.StructOf([]arrow.Field{
           // Name and Version are declared as dictionaries (Dictionary<Uint16, String>)).
           {Name: "name", Type: uint16Dictionary},
           {Name: "version", Type: uint16Dictionary},
        }...)},
        {Name: "metrics", Type: arrow.ListOf(arrow.StructOf([]arrow.Field{
           {Name: "name", Type: uint16Dictionary},
           {Name: "unit", Type: uint8Dictionary},
```

```
            {Name: "timestamp", Type: arrow.TIMESTAMP},
            {Name: "metric_type", Type: arrow.UINT8},
            {Name: "data_point", Type: arrow.ListOf(arrow.StructOf([]arrow.Field{
               {Name: "metric", Type: arrow.DenseUnionOf(
                  []arrow.Field{
                     {Name: "gauge", Type: arrow.StructOf([]arrow.Field{
                        {Name: "data_point", Type: arrow.FLOAT64},
                     }...)},
                     {Name: "sum", Type: arrow.StructOf([]arrow.Field{
                        {Name: "data_point", Type: arrow.FLOAT64},
                        {Name: "is_monotonic", Type: arrow.BOOL},
                     }...)},
                  },
                  []arrow.UnionTypeCode{GaugeMetricCode, SumMetricCode},
               )},
            }...))},
         }...))},
      }...))},
   }, nil)
)
```

In this pattern, we use a union type to represent an inheritance relationship. There are two types of Arrow union that are optimized for different cases. The dense union type has a relatively succinct memory representation but doesn't support vectorizable operations, making it less efficient during the processing phase. Conversely, a sparse union supports vectorization operations, but comes with a memory overhead directly proportional to the number of variants in the union. Dense and sparse unions have quite similar compression rates, with

sometimes a slight advantage for sparse unions. In addition, sparse unions with a large number of variants should generally be avoided, as they can lead to excessive memory consumption. For more details on the memory representation of unions, you can consult this page.

In certain scenarios, it may be more idiomatic to represent the inheritance relationship using multiple schemas (i.e., one schema per subtype), thereby avoiding the use of the union type. However, applying this approach to the aforementioned model may not be optimal, as the data preceding the inheritance relationship (i.e., `ResourceMetrics`, `Scope`, and `Metrics`) could potentially be duplicated numerous times. If the relationships between `ResourceMetrics`, `Metrics`, and `DataPoint` were 0..1 (zero-to-one) relationships, then the multi-schema approach would likely be the simplest and most idiomatic solution.

Denormalization & Flattening representations

If the `List` type is not supported in your telemetry pipeline, you can denormalize your data model. This process is often used in the database world to remove a join between two tables for optimization purposes. In the Arrow world, denormalization is employed to eliminate the `List` type by duplicating some data. Once transformed, the previous Arrow schema becomes.

```
Schema = arrow.NewSchema([]arrow.Field{
  {Name: "resource_metrics", Type: arrow.StructOf([]arrow.Field{
     {Name: "scope", Type: arrow.StructOf([]arrow.Field{
        // Name and Version are declared as dictionaries (Dictionary<Uint16, String>)).
        {Name: "name", Type: uint16Dictionary},
        {Name: "version", Type: uint16Dictionary},
     }...)},
     {Name: "metrics", Type: arrow.StructOf([]arrow.Field{
        {Name: "name", Type: uint16Dictionary},
        {Name: "unit", Type: uint8Dictionary},
        {Name: "timestamp", Type: arrow.TIMESTAMP},
        {Name: "metric_type", Type: arrow.UINT8},
        {Name: "data_point", Type: arrow.StructOf([]arrow.Field{
```

```
            {Name: "metric", Type: arrow.DenseUnionOf(
               []arrow.Field{
                  {Name: "gauge", Type: arrow.StructOf([]arrow.Field{
                     {Name: "value", Type: arrow.FLOAT64},
                  }...)},
                  {Name: "sum", Type: arrow.StructOf([]arrow.Field{
                     {Name: "value", Type: arrow.FLOAT64},
                     {Name: "is_monotonic", Type: arrow.BOOL},
                  }...)},
               },
               []arrow.UnionTypeCode{GaugeMetricCode, SumMetricCode},
            )},
         }...)},
      }...)},
   }...)},
}, nil)
```

List types are eliminated at all levels. The initial semantics of the model are preserved by duplicating the data of the levels below each data point value. The memory representation will generally be much larger than the previous one, but a query engine that does not support the `List` type will still be able to process this data. Interestingly, once compressed, this way of representing data may not necessarily be larger than the previous approach. This is because the columnar representation compresses very well when there is redundancy in the data.

If the union type is not supported by some components of your pipeline, it is also possible to eliminate them by merging the union variants (the nested structure ‘metric’ is removed, see below).

```
Schema = arrow.NewSchema([]arrow.Field{
   {Name: "resource_metrics", Type: arrow.StructOf([]arrow.Field{
      {Name: "scope", Type: arrow.StructOf([]arrow.Field{
```

```
        // Name and Version are declared as dictionaries (Dictionary<Uint16, String>)).
        {Name: "name", Type: uint16Dictionary},
        {Name: "version", Type: uint16Dictionary},
     }...)},
     {Name: "metrics", Type: arrow.StructOf([]arrow.Field{
        {Name: "name", Type: uint16Dictionary},
        {Name: "unit", Type: uint8Dictionary},
        {Name: "timestamp", Type: arrow.TIMESTAMP},
        {Name: "metric_type", Type: arrow.UINT8},
        {Name: "data_point", Type: arrow.StructOf([]arrow.Field{
           {Name: "value", Type: arrow.FLOAT64},
           {Name: "is_monotonic", Type: arrow.BOOL},
        }...)},
     }...)},
  }...)},
}, nil)
```

The final schema has evolved into a series of nested structures, where the fields of the union variants are merged into one structure. The trade-off of this approach is similar to that of sparse union - the more variants, the higher the memory occupation. Arrow supports the concept of bitmap validity to identify null values (1 bit per entry) for various data types, including those that do not have a unique null representation (e.g., primitive types). The use of bitmap validity makes the query part easier, and query engines such as DataFusion know how to use if efficiently, Columns with numerous nulls typically compress quite efficiently since the underlying arrays are generally initialized with 0's. Upon compression, these extensive sequences of 0's result in high compression efficiency, despite the memory overhead before compression in the case of sparse unions. Consequently, it is essential to select the appropriate trade-off based on your specific context.

In some extreme situations where nested structures are not supported, a flattening approach can be used to address this problem.

```
Schema = arrow.NewSchema([]arrow.Field{
  {Name: "scope_name", Type: uint16Dictionary},
  {Name: "scope_version", Type: uint16Dictionary},
  {Name: "metrics_name", Type: uint16Dictionary},
  {Name: "metrics_unit", Type: uint8Dictionary},
  {Name: "metrics_timestamp", Type: arrow.TIMESTAMP},
  {Name: "metrics_metric_type", Type: arrow.UINT8},
  {Name: "metrics_data_point_value", Type: arrow.FLOAT64},
  {Name: "metrics_data_point_is_monotonic", Type: arrow.BOOL},
}, nil)
```

The terminal fields (leaves) are renamed by concatenating the names of the parent structures to provide proper scoping. This type of structure is supported by all components of the Arrow ecosystem. This approach can be useful if compatibility is a crucial criterion for your system. However, it shares the same drawbacks as other alternative denormalization models.

The Arrow ecosystem is evolving rapidly, so it is likely that support for List, Map, and Union data types in query engines will improve quickly. If kernel functions are sufficient or preferable for your application, it is usually possible to utilize these nested types.

Adaptive/Dynamic representation

Some data models can be more challenging to translate into an Arrow schema, such as the following Protobuf example. In this example, a collection of attributes is added to each data point. These attributes are defined using a recursive definition that most languages and formats, like Protobuf, support (see the ‘AnyValue’ definition below). Unfortunately, Arrow (like most classical database schemas) does not support such recursive definition within schemas.

```
syntax = "proto3";
```

```
message Metric {
 message DataPoint {
   repeated Attribute attributes = 1;
   oneof value {
     int64 int_value = 2;
     double double_value = 3;
   }
 }

 enum MetricType {
   UNSPECIFIED = 0;
   GAUGE = 1;
   SUM = 2;
 }

 message Gauge {
   DataPoint data_point = 1;
 }

 message Sum {
   DataPoint data_point = 1;
   bool is_monotonic = 2;
 }

 string name = 1;
 int64 timestamp = 2;
 string unit = 3;
```

```
 MetricType type = 4;
 oneof metric {
   Gauge gauge = 5;
   Sum sum = 6;
 }
}

message Attribute {
 string name = 1;
 AnyValue value = 2;
}

// Recursive definition of AnyValue. AnyValue can be a primitive value, a list
// of AnyValues, or a list of key-value pairs where the key is a string and
// the value is an AnyValue.
message AnyValue {
 message ArrayValue {
   repeated AnyValue values = 1;
 }
 message KeyValueList {
   message KeyValue {
     string key = 1;
     AnyValue value = 2;
   }
   repeated KeyValue values = 1;
 }
```

```
 oneof value {
   int64 int_value = 1;
   double double_value = 2;
   string string_value = 3;
   ArrayValue list_value = 4;
   KeyValueList kvlist_value = 5;
 }
}
```

If the definition of the attributes were non-recursive, it would have been possible to directly translate them into an Arrow Map type.

To address this kind of issue and further optimize Arrow schema definitions, one can employ an adaptive and iterative method that automatically constructs the Arrow schema based on the data being translated. With this approach, fields are automatically dictionary-encoded according to their cardinalities, unused fields are eliminated, and recursive structures are represented in a specific manner. Another solution involves using a multi-schema approach, in which attributes are depicted in a separate Arrow Record, and the inheritance relation is represented by a self-referential relationship. These strategies will be covered in more depth in a future article. For those eager to learn more, the first method is utilized in the reference implementation of the OTel Arrow Adapter.

Data transport

Unlike to Protobuf, an Arrow schema is generally not known a priori by the two parties participating in an exchange. Before being able to exchange data in Arrow format, the sender must first communicate the schema to the receiver, as well as the contents of the dictionaries used in the data. Only after this initialization phase has been completed can the sender transmit batches of data in Arrow format. This process, known as Arrow IPC Stream, plays an essential role transporting Arrow data between systems. Several approaches can be employed to communicate these Arrow IPC Streams. The simplest method is to use Arrow Flight, which encapsulates Arrow IPC streams in a gRPC-based protocol. However, it is also possible to use your own implementation for specific contexts. Regardless of the solution you

choose, it is crucial to understand that the underlying protocol must be stateful to take full advantage of the Arrow IPC stream approach. To achieve the best compression rates, it is vital to send schemas and dictionaries only once in order to amortize the cost and minimize data redundancy between batches. This necessitates a transport that supports stream-oriented communications, such as gRPC.

Using a stateless protocol is possible for large batches because the overhead of the schema will be negligible compared to the compression gains achieved using dictionary encoding and columnar representation. However, dictionaries will have to be communicated for each batch, making this approach generally less efficient than a stream-oriented approach.

Arrow IPC Stream also supports the concept of "delta dictionaries," which allows for further optimization of batch transport. When a batch adds data to an existing dictionary (at the sender's end), Arrow IPC enables sending the delta dictionary followed by the batch that references it. On the receiver side, this delta is used to update the existing dictionary, eliminating the need to retransmit the entire dictionary when changes occur. This optimization is only possible with a stateful protocol.

To fully leverage the column-oriented format of Apache Arrow, it is essential to consider sorting and compression. If your data model is simple (i.e., flat) and has one or more columns representing a natural order for your data (e.g., timestamp), it might be beneficial to sort your data to optimize the final compression ratio. Before implementing this optimization, it is recommended to perform tests on real data since the benefits may vary. In any case, using a compression algorithm when sending your batches is advantageous. Arrow IPC generally supports the ZSTD compression algorithm, which strikes an excellent balance between speed and compression efficiency, especially for column-oriented data.

Lastly, some implementations (e.g., Arrow Go) are not configured by default to support delta dictionaries and compression algorithms. Therefore, it is crucial to ensure that your code employs these options to maximize data transport efficiency.

Experiments

If your initial data is complex, it is advisable to conduct your own experiments to optimize the Arrow representation according to your data and goals (e.g., optimizing the compression ratio or enhancing the query-ability of your data in Arrow format). In our case, we developed an overlay for Apache Arrow that enables us to carry out these experiments with ease, without having to deal with the intrinsic complexity of Arrow APIs. However, this comes at the expense of a slower conversion phase compared to using Arrow APIs directly. While this library is not currently public, it may become available if there is sufficient interest.

We also employed a "black box optimization" approach, which automatically finds the best combination to meet the objectives we aimed to optimize (refer to "Optimize your applications using Google Vertex AI Vizier" for a description of this approach).

Conclusion and next steps

Essentially, the key concept behind Apache Arrow is that it eliminates the need for serialization and deserialization, enabling zero-copy data sharing. Arrow achieves this by defining a language-agnostic, in-memory format that remains consistent across various implementations. Consequently, raw memory bytes can be transmitted directly over a network without requiring any serialization or deserialization, significantly enhancing data processing efficiency.

Converting a data model to Apache Arrow necessitates adaptation and optimization work, as we have begun to describe in this article. Many parameters must be considered, and it is recommended to perform a series of experiments to validate the various choices made during this process.

Handling highly dynamic data with Arrow can be challenging. Arrow requires the definition of a static schema, which can sometimes make representing this type of data complex or suboptimal, especially when the initial schema contains recursive definitions. This article has discussed several approaches to address this issue. The next article will be dedicated to a hybrid strategy that involves adapting the Arrow

schema on-the-fly to optimize memory usage, compression ratio, and processing speed based on the data being represented. This approach is quite unique and deserves a separate article.