MEP 13 -- Support Apache Arrow As In-Memory Data Format

Current state: "Rejected"

ISSUE: #7210

PRs:

Keywords: arrow/column-based/row-based

Released: Milvus 2.0

Summary

Milvus 2.0 is a cloud-native and multi-language vector database, we use gRPC and pulsar to communicate among SDK and components.

In consideration of the data size, especially when inserting and search result returning, Milvus takes a lot of CPU cycles to do serialization and deserialization.

In this enhancement proposal, we suggest to adopt Apache Arrow as Milvus in-memory data format. Since in the field of big data, Apache Arrow has been a

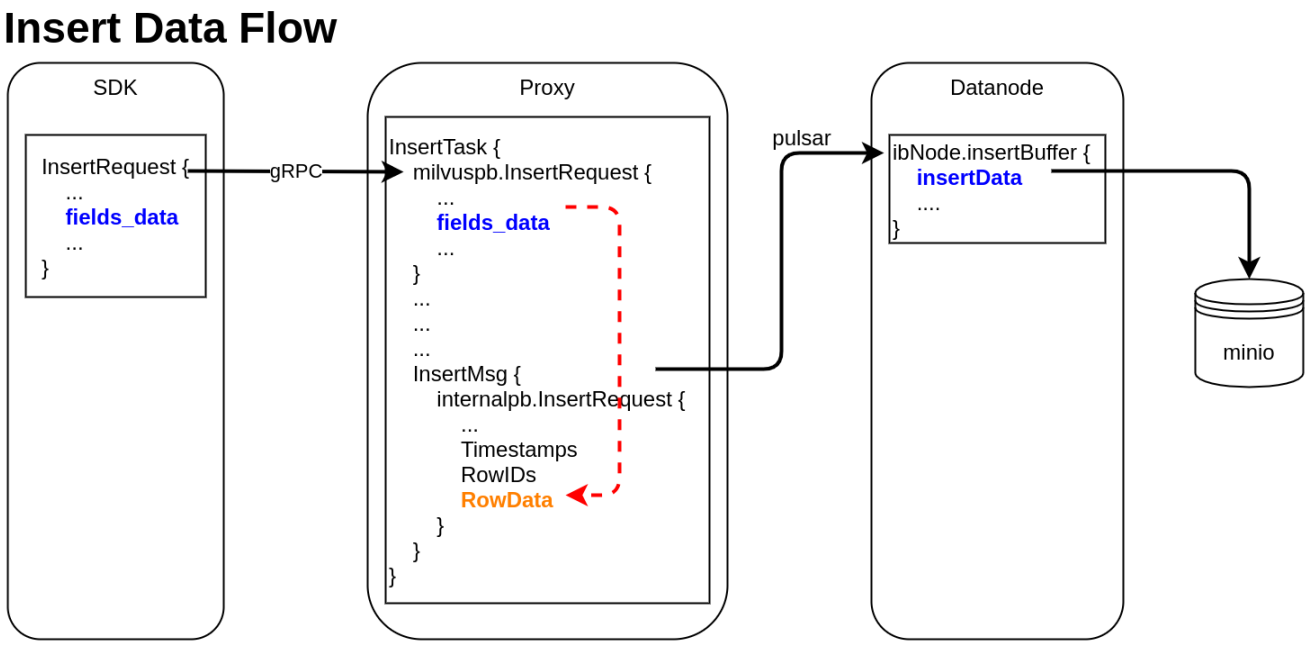
factor standard for in-memory analytics. It specifies a standardized language-independent columnar memory format.

Motivation(required)

From a data perspective, Milvus mainly includes 2 data flows:

1. Insert data flow
2. Search data flow

Insert Data Flow



BLUE - Column-based data structure

ORANGE - Row-based data structure

RED DASHED LINE - Data format conversion

Insert data flow includes following steps:

1. pymilvus creates a data insert request with type *milvuspb.InsertRequest* (client/prepare.py::bulk_insert_param)

```
// grpc-proto/milvus.proto
message InsertRequest {
  common.MsgBase base = 1;
  string db_name = 2;
  string collection_name = 3;
  string partition_name = 4;
  repeated schema.FieldData fields_data = 5;        // fields' data
  repeated uint32 hash_keys = 6;
  uint32 num_rows = 7;
}
```

   Data is inserted into *fields_data* by column, *schemapb.FieldData* is defined as following:

```
// grpc-proto/schema.proto
message ScalarField {
  oneof data {
    BoolArray bool_data = 1;
    IntArray int_data = 2;
    LongArray long_data = 3;
    FloatArray float_data = 4;
    DoubleArray double_data = 5;
    StringArray string_data = 6;
    BytesArray bytes_data = 7;
  }
}

message VectorField {
  int64 dim = 1;
  oneof data {
    FloatArray float_vector = 2;
    bytes binary_vector = 3;
  }
}

message FieldData {
  DataType type = 1;
  string field_name = 2;
  oneof field {
    ScalarField scalars = 3;
    VectorField vectors = 4;
  }
  int64 field_id = 5;
}
```

2. *milvuspb.InsertRequest* is serialized and send via gRPC
3. Proxy receives *milvuspb.InsertRequest*, creates *InsertTask* for it, and adds this task into execution queue (internal/proxy/impl.go::Insert)
4. *InsertTask* is executed, the **column-based** data stored in *InsertTask.req* is converted to **row-based** format, and saved into another internal message with type *internalpb.InsertRequest* (internal/proxy/task.go::transferColumnBasedRequestToRowBasedData)

```
// internal/proto/internal.proto
message InsertRequest {
  common.MsgBase base = 1;
  string db_name = 2;
  string collection_name = 3;
  string partition_name = 4;
  int64 dbID = 5;
  int64 collectionID = 6;
  int64 partitionID = 7;
  int64 segmentID = 8;
  string channelID = 9;
  repeated uint64 timestamps = 10;
  repeated int64 rowIDs = 11;
  repeated common.Blob row_data = 12;  // row-based data
}
```

rowID and timestamp are added for each row data

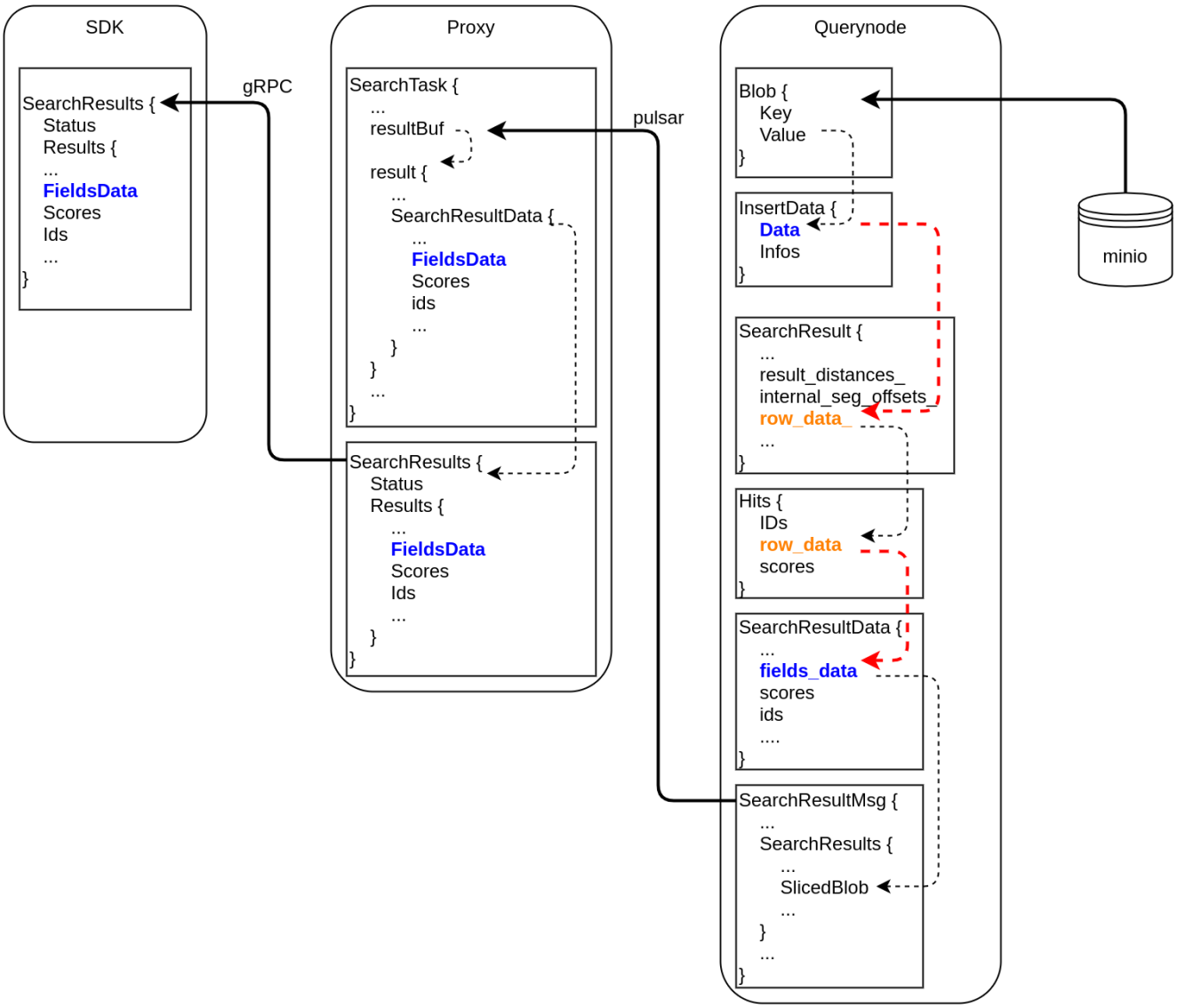
5. *Proxy* encapsulates *internalpb.InsertRequest* into *InsertMsg*, and send it to pulsar channel
6. *Datanode* receives *InsertMsg* from pulsar channel, restore data to **column-based** into structure *InsertData* (internal/datanode /flow_graph_insert_buffer_node.go::insertBufferNode::Operate)

```
type InsertData struct {
    Data  map[FieldID]FieldData // field id to field data
    Infos []BlobInfo
}
```

7. *InsertData* is written into Minio with parquet format (internal/datanode/flow_graph_insert_buffer_node.go::flushSegment)

Search Data Flow

Search Data Flow



BLUE - Column-based data structure

ORANGE - Row-based data structure

RED DASHED LINE - Data format conversion

Search data flow includes following steps:

1. *querynode* reads segment's binlog files from Minio, and saves them into structure Blob (internal/querynode/segment_loader.go:: loadSegmentFieldsData)

```
type Blob struct {
    Key   string    // binlog file path
    Value []byte    // binlog file data
}
```

   The data in Blob is deserialized, raw-data in it is saved into structure *InsertData*
2. *querynode* invokes search engine to get *SearchResult* (internal/query_node/query_collection.go::search)

```
// internal/core/src/common/Types.h
struct SearchResult {
 ...
 public:
    int64_t num_queries_;
    int64_t topk_;
    std::vector<float> result_distances_;

 public:
    void* segment_;
    std::vector<int64_t> internal_seg_offsets_;
    std::vector<int64_t> result_offsets_;
    std::vector<std::vector<char>> row_data_;
};
```

At this time, only "result_distances_" and "internal_seg_offsets_" of "SearchResult" are filled into data.

querynode reduces all *SearchResult* returned by segment, fetches all other fields' data, and saves them into "row_data_" with **row-based** format. (internal/query_node/query_collection.go::reduceSearchResultsAndFillData)

3. querynode organizes SearchResult again, and save them into structure milvus.Hits

```
// internal/proto/milvus.proto
message Hits {
  repeated int64 IDs = 1;
  repeated bytes row_data = 2;
  repeated float scores = 3;
}
```

4. **Row-based** data saved in *milvus.Hits* is converted to **column-based** data, and saved into *schemapb.SearchResultData* (internal/query_node /query_collection.go::translateHits)

```
// internal/proto/schema.proto
message SearchResultData {
  int64 num_queries = 1;
  int64 top_k = 2;
  repeated FieldData fields_data = 3;
  repeated float scores = 4;
  IDs ids = 5;
  repeated int64 topks = 6;
}
```

5. *schemapb.SearchResultData* is serialized, encapsulated as *internalpb.SearchResults*, saved into *SearchResultMsg*, and send into pulsar channel (internal/query_node/query_collection.go::search)

```
// internal/proto/internal.proto
message SearchResults {
  common.MsgBase base = 1;
  common.Status status = 2;
  string result_channelID = 3;
  string metric_type = 4;
  repeated bytes hits = 5;  // search result data

  // schema.SearchResultsData inside
  bytes sliced_blob = 9;
  int64 sliced_num_count = 10;
  int64 sliced_offset = 11;

  repeated int64 sealed_segmentIDs_searched = 6;
  repeated string channelIDs_searched = 7;
  repeated int64 global_sealed_segmentIDs = 8;
}
```

6. *Proxy* collects all *SearchResultMsg* from *querynode*s, gets *schemapb.SearchResultData* by deserialization, then gets *milvuspb.SearchResults* by reducing, finally send back to SDK visa gRPC. (internal/proxy/task.go::SearchTask::PostExecute)

```
// internal/proto/milvus.proto
message SearchResults {
  common.Status status = 1;
  schema.SearchResultData results = 2;
}
```

7. SDK receives milvuspb.SearchResult

In above 2 data flows, we can see frequent format conversion between column-based data and row-based data (marked as RED dashed line).

If we use Arrow as all in-memory data format, we can:

- omit the serialization and deserialization between SDK and proxy
- remove all format conversion between column-based data and row-based data
- use Parquet as binlog file format, and write from arrow data directly

Proposal Benefit Analysis (optional)

Arrow memory usage, test following 3 scenarios used in Milvus:

Data Type	Raw Data Size (Byte)	Array Array Buffer Size (Byte)
int64	80000	80000
FixedSizeList (float32, dim = 128)	5120000	5160064
string (len = 512)	5120000	5160064

For Scalar data, Arrow Array uses memory as same as raw data;

for vector data or string, Arrow Array uses few more memory than raw data (about 4 bytes for each row).

Give an example to illustrate the problems we will encounter if using Arrow.

During insert, after *Proxy* receives data, it will encapsulate the data into *InsertMsg* by line, and then send it to *Datanode* through *Pulsar*.

Splitting by line is based on two reasons:

1. Each collection has two or more physical channels. Data insertion performance can be improved by inserting multiple channels at the same time.
2. *Pulsar* limits the size of each *InsertMsg*

We tried 4 solutions, each has its own problems:

- Solution-1

  After the *Proxy* receives the inserted data, it only creates one Arrow RecordBatch, encapsulates the data into *InsertMsg* by line, and then sends it to *Datanode* through *Pulsar*.

  PROBLEM: There is no interface to read data item by item from Arrow RecordBatch. RecordBatch has a NewSlice interface, but the return value of NewSlice cannot do anything except print.

- Solution-2

  After the *Proxy* receives the inserted data, it creates multiple Arrow RecordBatch in advance according to the size limit of *Pulsar* for *InsertMsg* . The data is serialized according to the RecordBatch, inserted into the *InsertMsg*, and then sent to the *Datanode* through *Pulsar*. *Datanode* combines multiple RecordBatch into one complete RecordBatch.

  PROBLEM: Multiple RecordBatch can only be logically restored to one ArrowTable, but each column of data is physically discontinuous, so subsequent columnar operations cannot be performed.

- Solution-3

  After the *Proxy* receives the inserted data, create multiple Arrow Array by field, instead of RecordBatch.

  PROBLEM: The primitive unit of serialized data in Arrow is **RecordBatch**. Arrow does not provide interface to serialize Arrow Array.

- Solution-4

After the *Proxy* receives the inserted data, it creates multiple RecordBatch in advance according to the size limit of the *Pulsar* for *InsertMsg*. The data is serialized according to the RecordBatch and inserted into the *InsertMsg*, and then sent to the *Datanode* through *Pulsar*. The *Data node* receives multiple RecordBatch, fetches the data from each column, and regenerates a new RecordBatch.

PROBLEM: There seems no advantages comparing with current implementation.

Summarize some limitations in the use of Arrow:

1. Arrow data can only be serialized and deserialized by unit of RecordBatch;
2. Cannot copy out row data from RecordBatch;
3. RecordBatch must be regenerated after sending via pulsar.

Arrow is suitable for data analysis scenario (data is sealed and read only).

In Milvus, we need do data split and concatenate.

Arrow seems not a good choice for Milvus.

Design Details(required)

~~We divide this MEP into 2 stages, all compatibility changes will be achieved in Stage 1 before Milvus 2.0.0, other internal changes can be left later.~~

~~Stage 1~~

1. ~~Update InsertRequest in milvus.proto, change Insert to use Arrow format~~
2. ~~Update SearchRequest/Hits in milvus.proto, and SearchResultData in schema.proto, change Search to use Arrow format~~
3. ~~Update QueryResults in milvus.proto, change Query to use Arrow format~~

~~Stage 2~~

1. ~~Update Storage module to use GoArrow to write Parquet from Arrow, or read Arrow from Parquet directly, remove C++ Arrow.~~
2. ~~Remove all internal row-based data structure, including "RowData" in internalpb.InsertRequest, "row_data" in milvuspb.Hits, "row_data_" in C++ SearchResult.~~
3. ~~Optimize search result flow~~

Test Plan(required)

Pass all CI flows

References(optional)

https://arrow.apache.org/docs/

Arrow Test Code (Go)

```
import (
        "bytes"
        "fmt"
        "testing"

        "github.com/apache/arrow/go/arrow"
        "github.com/apache/arrow/go/arrow/array"
        "github.com/apache/arrow/go/arrow/ipc"
        "github.com/apache/arrow/go/arrow/memory"
)

const (
        _DIM = 4
)

var pool = memory.NewGoAllocator()
```

```
func CreateArrowSchema() *arrow.Schema {
	fieldVector := arrow.Field{
		Name: "field_vector",
		Type: arrow.FixedSizeListOf(_DIM, arrow.PrimitiveTypes.Float32),
	}
	fieldVal := arrow.Field{
		Name: "field_val",
		Type: arrow.PrimitiveTypes.Int64,
	}
	schema := arrow.NewSchema([]arrow.Field{fieldVector, fieldVal}, nil)
	return schema
}

func CreateArrowRecord(schema *arrow.Schema, iValues []int64, vValues []float32) array.Record {
	rb := array.NewRecordBuilder(pool, schema)
	defer rb.Release()
	rb.Reserve(len(iValues))

	rowNum := len(iValues)
	for i, field := range rb.Schema().Fields() {
		switch field.Type.ID() {
		case arrow.INT64:
			vb := rb.Field(i).(*array.Int64Builder)
			vb.AppendValues(iValues, nil)
		case arrow.FIXED_SIZE_LIST:
			lb := rb.Field(i).(*array.FixedSizeListBuilder)
			valid := make([]bool, rowNum)
			for i := 0; i < rowNum; i++ {
				valid[i] = true
			}
			lb.AppendValues(valid)
			vb := lb.ValueBuilder().(*array.Float32Builder)
			vb.AppendValues(vValues, nil)
		}
	}

	rec := rb.NewRecord()

	return rec
}

func WriteArrowRecord(schema *arrow.Schema, rec array.Record) []byte {
	defer rec.Release()
	blob := make([]byte, 0)
	buf := bytes.NewBuffer(blob)

	// internal/arrdata/ioutil.go
	writer := ipc.NewWriter(buf, ipc.WithSchema(schema), ipc.WithAllocator(pool))
	defer writer.Close()

	//ShowArrowRecord(rec)
	if err := writer.Write(rec); err != nil {
		panic("could not write record: %v" + err.Error())
	}

	err := writer.Close()
	if err != nil {
		panic(err.Error())
	}

	return buf.Bytes()
}

func ReadArrowRecords(schema *arrow.Schema, blobs [][]byte) array.Record {
	iValues := make([]int64, 0)
	vValues := make([]float32, 0)
	for _, blob := range blobs {
		buf := bytes.NewReader(blob)

		reader, err := ipc.NewReader(buf, ipc.WithSchema(schema), ipc.WithAllocator(pool))
		if err != nil {
```

```
                        panic("create reader fail: %v" + err.Error())
                }
                defer reader.Release()

                rec, err := reader.Read()
                if err != nil {
                        panic("read record fail: %v" + err.Error())
                }
                defer rec.Release()

                for _, col := range rec.Columns() {
                        switch col.DataType().ID() {
                        case arrow.INT64:
                                arr := col.(*array.Int64)
                                iValues = append(iValues, arr.Int64Values()...)
                        case arrow.FIXED_SIZE_LIST:
                                arr := col.(*array.FixedSizeList).ListValues().(*array.Float32)
                                vValues = append(vValues, arr.Float32Values()...)
                        }
                }
        }
        ret := CreateArrowRecord(schema, iValues, vValues)
        ShowArrowRecord(ret)

        return ret
}

func ReadArrowRecordsToTable(schema *arrow.Schema, blobs [][]byte) array.Table {
        recs := make([]array.Record, 0)
        for _, blob := range blobs {
                buf := bytes.NewReader(blob)

                reader, err := ipc.NewReader(buf, ipc.WithSchema(schema), ipc.WithAllocator(pool))
                if err != nil {
                        panic("create reader fail: %v" + err.Error())
                }
                defer reader.Release()

                rec, err := reader.Read()
                if err != nil {
                        panic("read record fail: %v" + err.Error())
                }
                defer rec.Release()

                recs = append(recs, rec)
        }
        table := array.NewTableFromRecords(schema, recs)
        ShowArrowTable(table)

        return table
}

func ShowArrowRecord(rec array.Record) {
        fmt.Printf("\n============================\n")
        fmt.Printf("Schema: %v\n", rec.Schema())
        fmt.Printf("NumCols: %v\n", rec.NumCols())
        fmt.Printf("NumRows: %v\n", rec.NumRows())
        //rowNum := int(rec.NumRows())
        for i, col := range rec.Columns() {
                fmt.Printf("Column[%d] %q: %v\n", i, rec.ColumnName(i), col)
        }
}

func ShowArrowTable(tbl array.Table) {
        fmt.Printf("\n============================\n")
        fmt.Printf("Schema: %v\n", tbl.Schema())
        fmt.Printf("NumCols: %v\n", tbl.NumCols())
        fmt.Printf("NumRows: %v\n", tbl.NumRows())
        for i := 0; i < int(tbl.NumCols()); i++ {
                col := tbl.Column(i)
                fmt.Printf("Column[%d] %s: %v\n", i, tbl.Schema().Field(i).Name, col.Data().Chunks())
```

```
        }
}

func TestArrowIPC(t *testing.T) {
        schema := CreateArrowSchema()
        rec0 := CreateArrowRecord(schema, []int64{0}, []float32{0,0,0,0})
        rec1 := CreateArrowRecord(schema, []int64{1,2,3}, []float32{1,1,1,1,2,2,2,2,3,3,3,3})
        blob0 := WriteArrowRecord(schema, rec0)
        blob1 := WriteArrowRecord(schema, rec1)
        ReadArrowRecords(schema, [][]byte{blob0, blob1})
        ReadArrowRecordsToTable(schema, [][]byte{blob0, blob1})
}
```