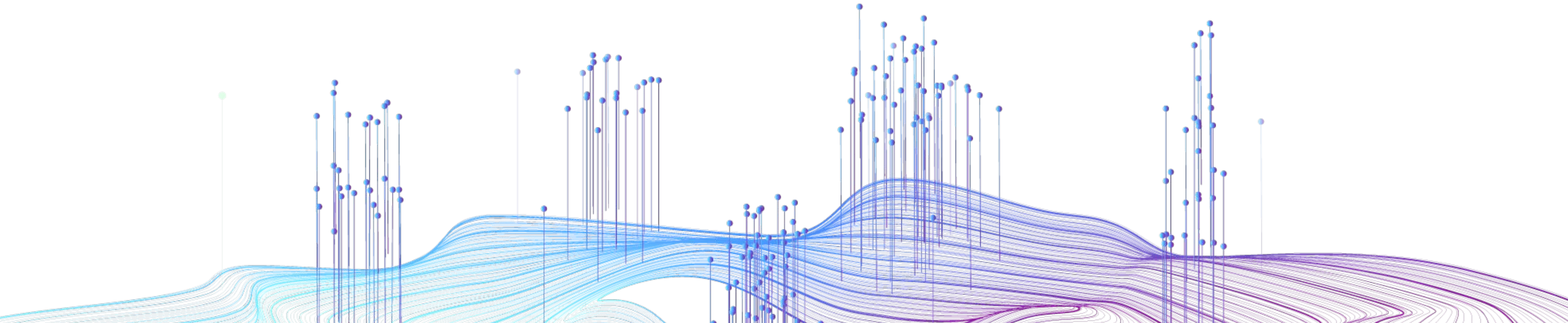
VOLTRON DATA



Apache Arrow and Go: A Match Made in Data

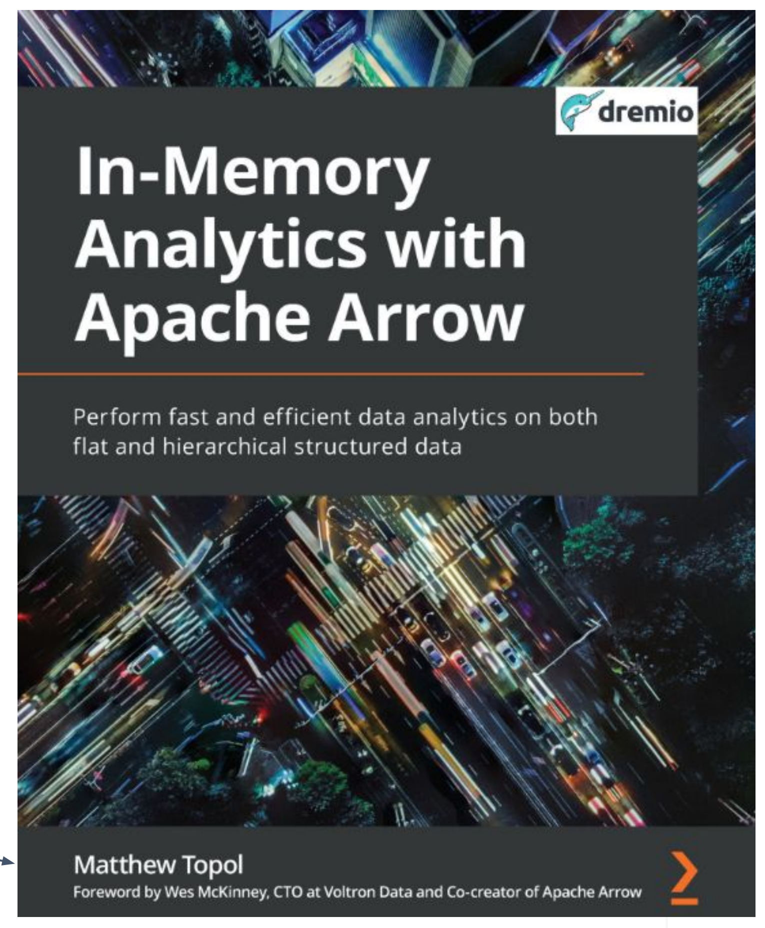
October 3rd, 2022

Presented by:

Matthew Topol

ARROW

Who am I?



Email

matt@voltrondata.com

Author Of

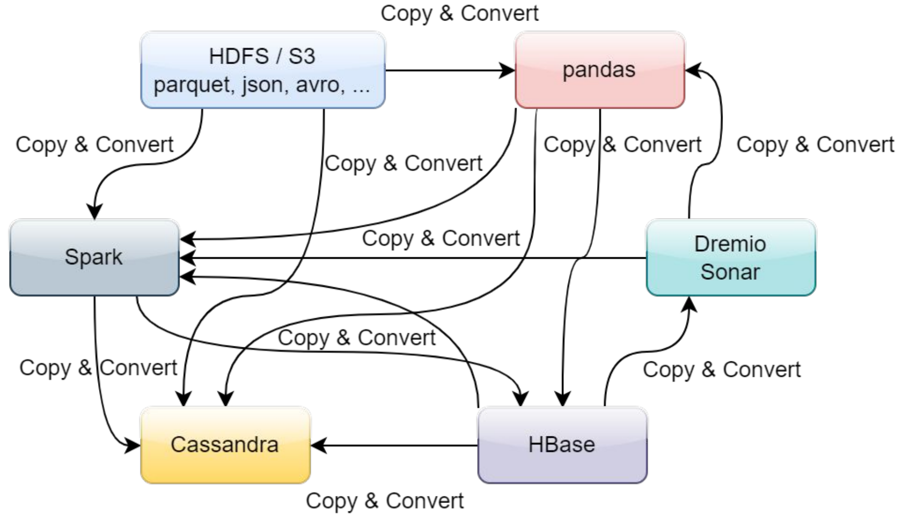
“In-Memory Analytics With Apache Arrow”

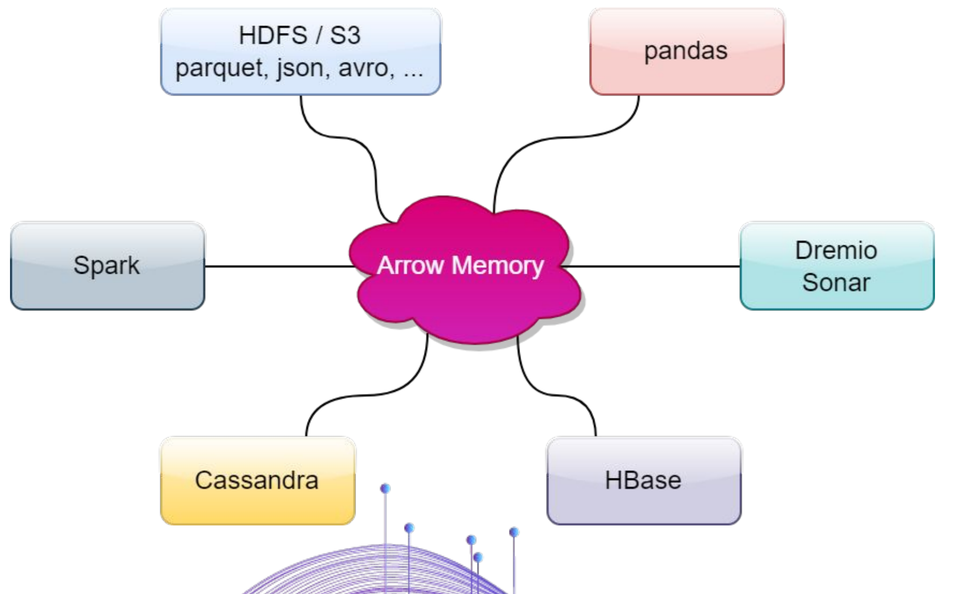
Staff Software Engineer at Voltron Data
Apache Arrow Contributor

The Rundown

- Quick Primer on Apache Arrow
- Why Go?
- Simple Code Examples
- Example Code Walkthrough: A Streaming Data Pipeline
- What else can we do?
- More Resources
- Q/A

A quick primer on





High Performance, In-Memory Columnar Format

No Data Serialization / Deserialization required!

Polyglot! Implementations in many languages

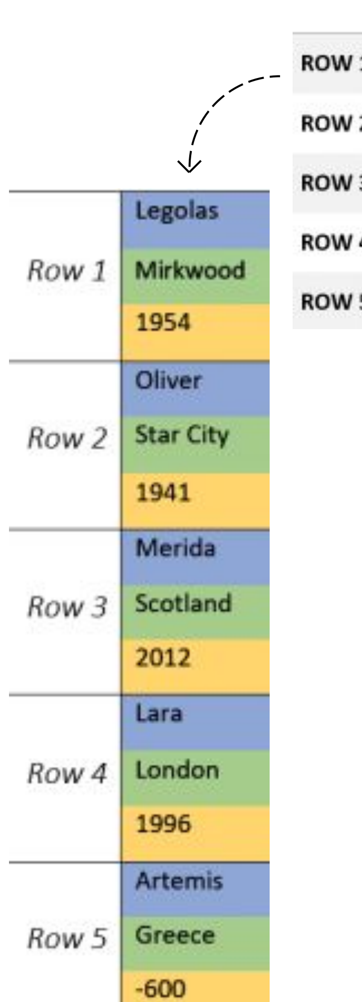
Go, C++, Rust, Python, R, Java, Julia, MATLAB, and more...

What is Columnar?

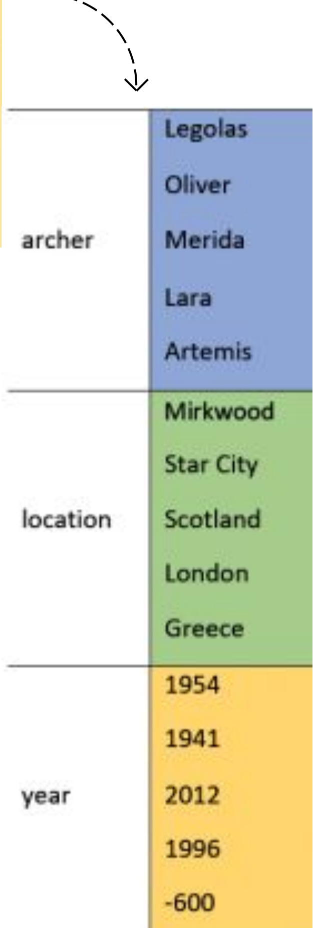
Table of Data

	ARCHER	LOCATION	YEAR
ROW 1	Legolas	Mirkwood	1954
ROW 2	Oliver	Star City	1941
ROW 3	Merida	Scotland	2012
ROW 4	Lara	London	1996
ROW 5	Artemis	Greece	-600

Row Oriented Memory Buffer



Arrow Columnar Memory Buffer



Why Columnar?

Memory Locality
I/O
Vectorization

A **Less I/O, lower memory usage, fewer page faults**

Get All Archers in Europe:

Only need two columns! (Archer, Location)

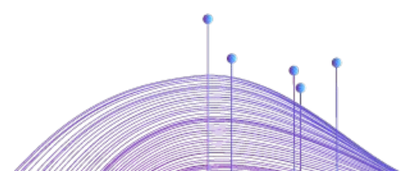
1. *Spin through Locations for indexes*
2. *Get Archers at those indexes*

B **Significantly faster computation!**

Calculate mean for Year column:

Only need the one column! (Year)

1. *Vectorized operations require contiguous memory*
2. *Our column is already contiguous memory!*



But why Golang??

Simple and easy to learn

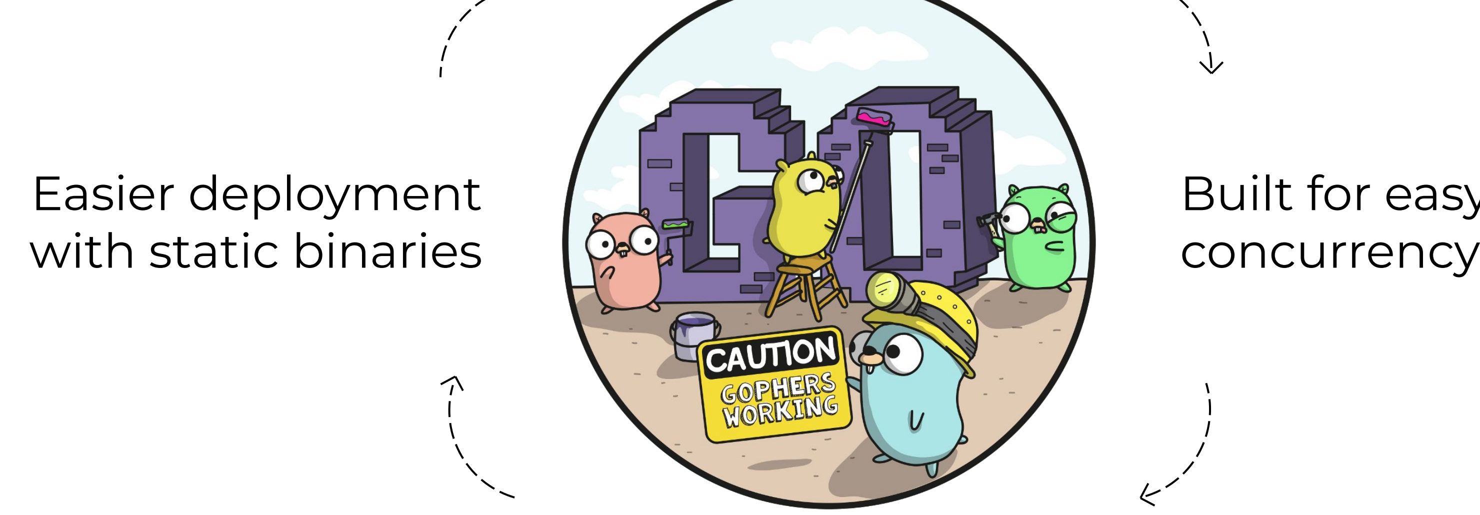
Go is fast!

Built for easy concurrency

Great multi-core usage and scaling

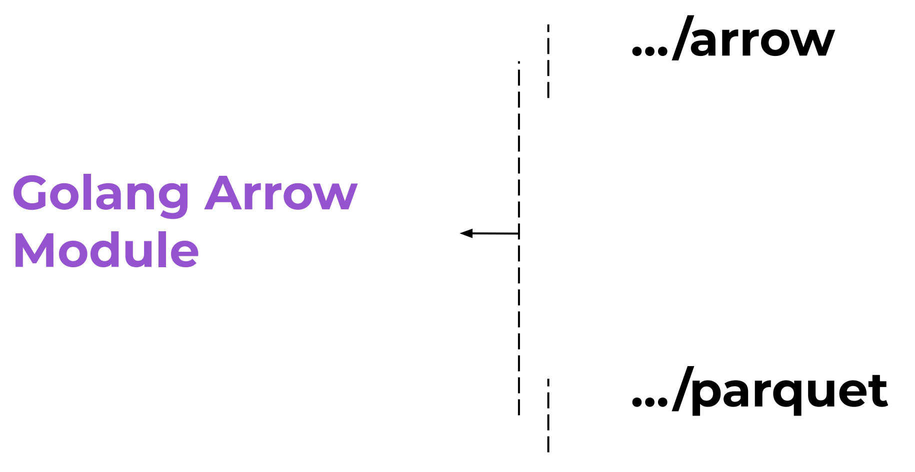
Excellent memory usage characteristics

Easier deployment with static binaries



github.com/apache/arrow/go/v9

v10 should be released in the next couple weeks!!



CSV, JSON, and Arrow IPC reader/writers

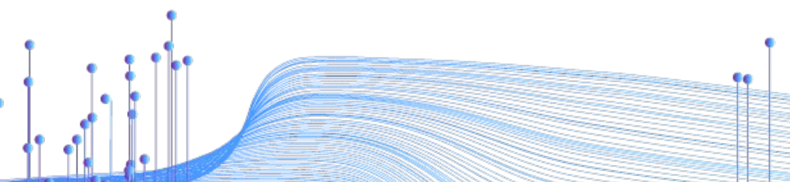
Arrow Flight and Flight SQL client and server

Supports multiple architectures (AMD64, ARM64, s390x, etc.) and leverages SIMD/NEON

Low memory usage, high performance reader/writer

Contains *pqarrow* package for easy interoperability between Parquet and Arrow

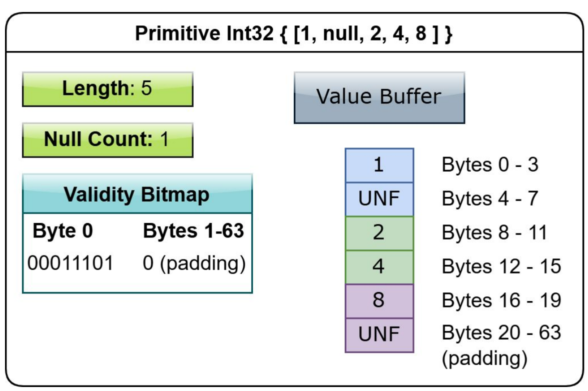
Supports multiple architectures (AMD64, ARM64, s390x, etc.) and leverages SIMD/NEON

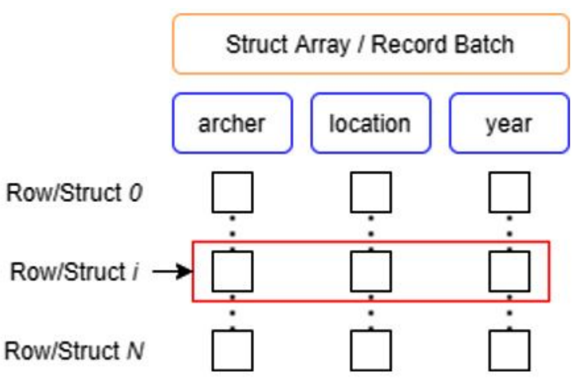


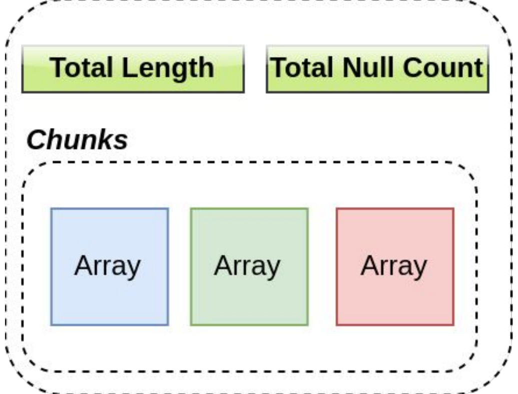
Let’s start exploring!

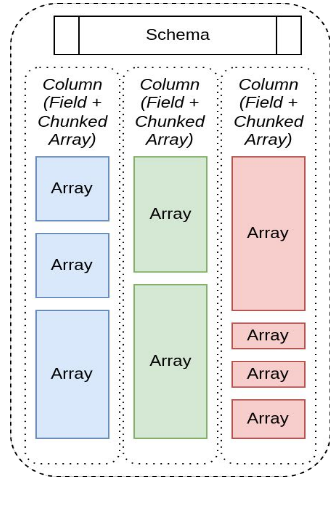
The Go Arrow and Parquet libraries

But first... Some Terminology and Types









Array (arrow.Array)

Logical Data type, length, null count and 1 or more Buffers of data

Record Batch *(arrow.Record)*

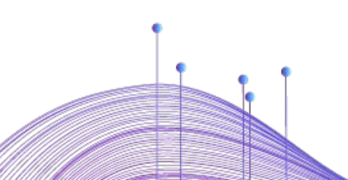
Collection of Arrays with the same length and a Schema (Collection of Fields)

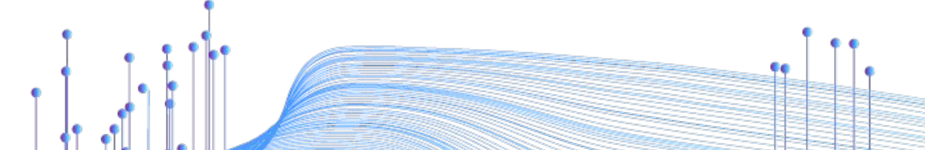
Chunked Array *(arrow.Chunked)*

Sequence of arrays with the same data type, total length and total null count

Table *(arrow.Table)*

Collection of Columns (Chunked Array + Field) with the same total length and a schema



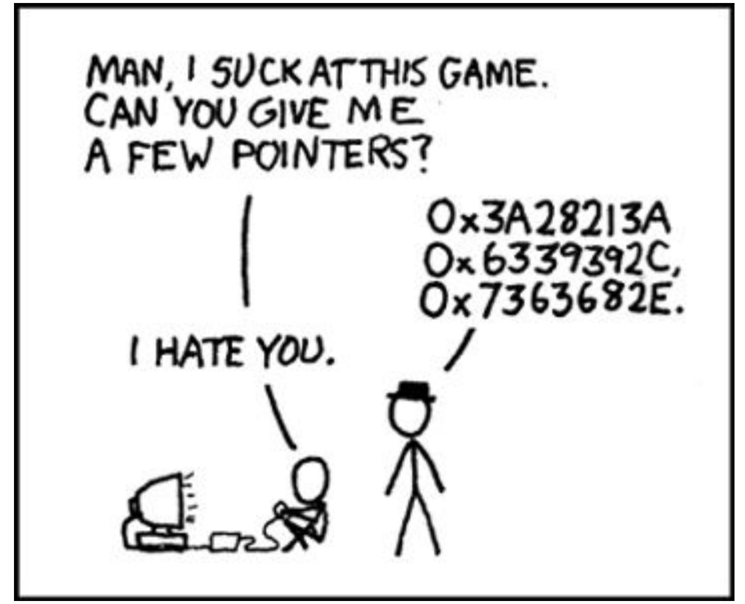


```go
import (
    "fmt"

    "github.com/apache/arrow/go/v10/arrow"
    "github.com/apache/arrow/go/v10/arrow/array"
    "github.com/apache/arrow/go/v10/arrow/memory"
)

func Example_buildInt64() {
    bldr := array.NewInt64Builder(memory.DefaultAllocator)
    defer bldr.Release() // <-- Notice this!

    bldr.Append(25)
    bldr.AppendNull()
    // nil bool slice means ALL valid
    bldr.AppendValues([]int64{1, 2, 3, 4}, nil)
    // otherwise bool slice indicates nulls with false
    bldr.AppendValues([]int64{5, 0, 6, 7},
        []bool{true, false, true, true})

    arr := bldr.NewArray()
    defer arr.Release() // <-- Notice!
    fmt.Println(arr)

    // Output:
    // [25 (null) 1 2 3 4 5 (null) 6 7]
}
```

Simple Example

Build an Int64 Array

Memory Handling



Retain / Release

Reference counting is used to track usage of buffers

Manage ownership and eagerly try to free memory

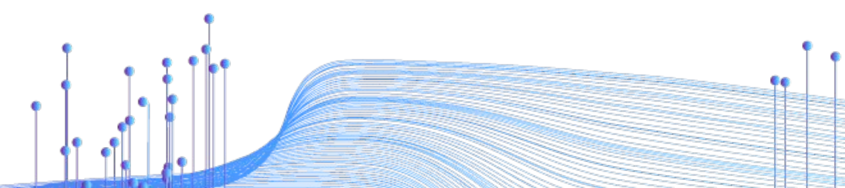
Ties into Allocator interface for custom handling

memory.Allocator

Interface for custom memory allocation, default just uses `make([]byte, …)`

Only Three methods: Allocate, Reallocate, Free

CheckedAllocator for tracking memory usage

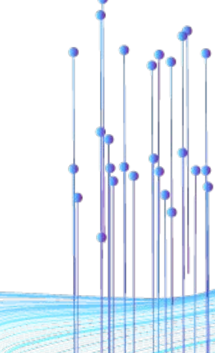


```go
func buildStruct() {
	archerType := arrow.StructOf(
		arrow.Field{Name: "archer", Type: arrow.BinaryTypes.String},
		arrow.Field{Name: "location", Type: arrow.BinaryTypes.String},
		arrow.Field{Name: "year", Type: arrow.PrimitiveTypes.Int16},
	)

	archers := []string{"Legolas", "Oliver", "Merida", "Lara", "Artemis"}
	locations := []string{"Murkwood", "Star City", "Scotland", "London", "Greece"}
	years := []int16{1954, 1941, 2012, 1996, -600}

	bldr := array.NewStructBuilder(memory.DefaultAllocator, archerType)
	defer bldr.Release()

	// notice we don't have to separately release these
	// they are owned by the struct builder!
	archerBldr := bldr.FieldBuilder(0).(*array.StringBuilder)
	locBldr := bldr.FieldBuilder(1).(*array.StringBuilder)
	yearBldr := bldr.FieldBuilder(2).(*array.Int16Builder)

	for i := range archers {
		bldr.Append(true) // <-- Valid struct
		archerBldr.Append(archers[i])
		locBldr.Append(locations[i])
		yearBldr.Append(years[i])
	}

	bldr.Append(false) // == bldr.AppendNull()

	arr := bldr.NewStructArray()
	defer arr.Release() // new array! need to release!

	fmt.Println(arr)
}
```

Struct Builder

Multiple field builders

Builder for each Array type and even a ***RecordBuilder*** which is similar to the ***StructBuilder***



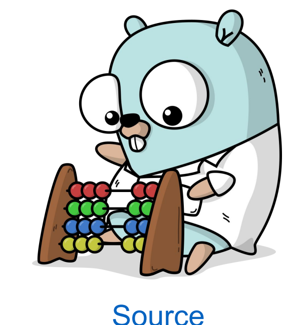
Reading and Writing Data

Multiple formats supported!

Arrow Record Reader/Writer

CSV

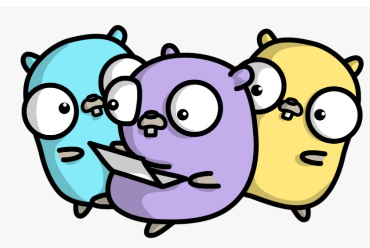
- Can provide an explicit schema or infer types
- Specify `null` values, delimiter, line endings
- Can control Record Batch chunk size

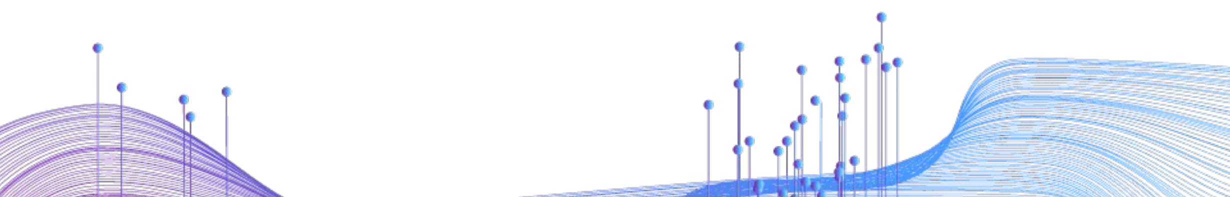


Source

Parquet

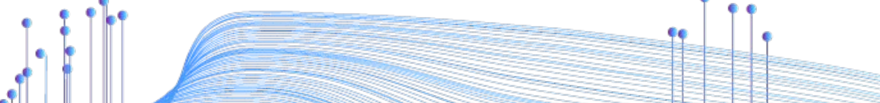
- Highly efficient Columnar storage
- Often Zero-Copy converting to Arrow
- Can easily read columns and row groups in parallel





Sample Usage

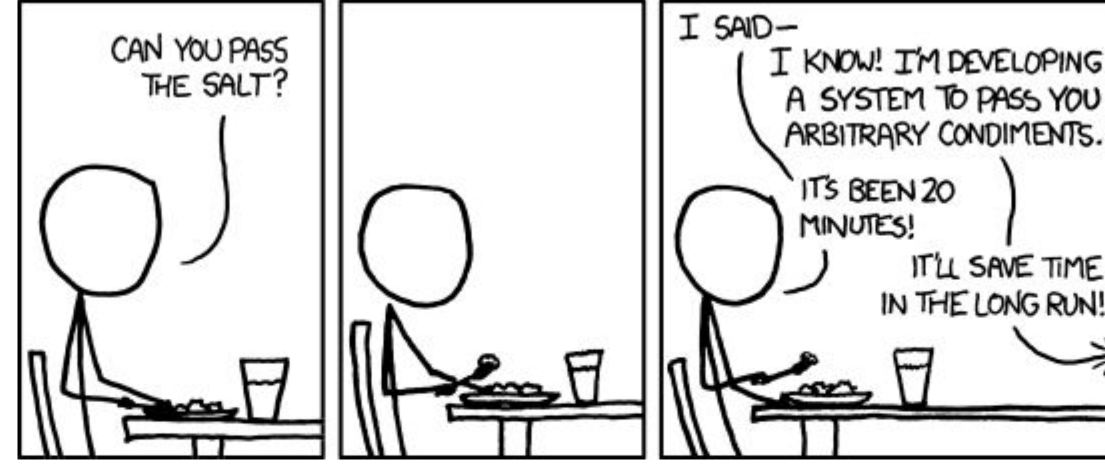
"The Movies Dataset"



Example: A Streaming Data Pipeline

Yes, it's contrived. But it's informative!

1. Read CSV Data
2. Transform / Add / Replace Columns
3. Write out Parquet File



Example: The Sample Data

Kaggle: “The Movies Dataset”

https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset

Source

Most columns are easy

bool, int, float, string

CSV reader can handle nulls for us

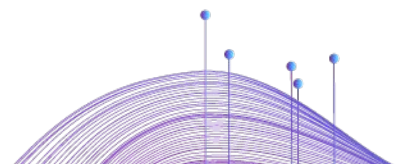
Infer the column types

Zero-copy transfer to new `arrow.Record`

Some columns we want to manipulate

String column values that are JSON strings converted into Lists for easier processing

Any other streaming transformations you’d like…

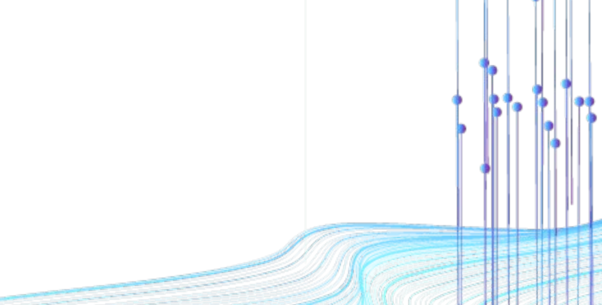


```go
ch := make(chan arrow.Record, 20)
go func() {
	// close the channel when done to signal
	// future pipeline steps
	defer close(ch)
	f, err := os.Open("movies_metadata.csv")
	if err != nil {
		panic(err) // <-- should handle better
	}
	defer f.Close()

	// infer the types and schema from the header line
	// and first line of data.
	rdr := csv.NewInferringReader(f, csv.WithChunk(5000),
		// strings can be null, and these are the values
		// to consider as "null"
		csv.WithNullReader(true, "", "null", "[]"),
		// there's a header line in the data, use it
		csv.WithHeader(true))

	for rdr.Next() {
		rec := rdr.Record()
		rec.Retain()
		ch <- rec
	}

	if rdr.Err() != nil {
		panic(rdr.Err())
	}
}()
```

Reading CSV Data

Stream Records via Channels

Low Memory usage, easy parallelism with Golang



```go
// we need to know the fields we're expecting in this JSON string
// for this example, we'll hardcode it
bldr := array.NewListBuilder(memory.DefaultAllocator, arrow.StructOf(
    arrow.Field{Name: "id", Type: arrow.PrimitiveTypes.Int32},
    arrow.Field{Name: "name", Type: arrow.BinaryTypes.String},
))
defer bldr.Release()

var outSchema *arrow.Schema
for rec := range ch {
    genresCol := rec.Column(3).(*array.String)

    bldr.Reserve(int(rec.NumRows()))
    for i := 0; i < genresCol.Len(); i++ {
        if genresCol.IsNull(i) {
            bldr.AppendNull()
            continue
        }

        // convert single quotes to dbl quotes
        // we want valid JSON
        vals := reID.ReplaceAllString(re.ReplaceAllString(genresCol.Value(i),
            `"${key}": "${dbl}${sngl}"`), `"id": $1`)
        if err := bldr.UnmarshalJSON([]byte("[" + vals + "]")); err != nil {
            panic(err)
        }
    }

    cols := rec.Columns()
    // modify the slice of arrays
    // new record doesn't copy the columns!
    cols[3].Release()
    genreCol := bldr.NewArray()
    cols[3] = genreCol

    // if we don't know the entire schema beforehand, we can just copy the existing
    // schema and replace the field for the column we're altering
    if outSchema == nil {
        fieldList := make([]arrow.Field, rec.NumCols())
        copy(fieldList, rec.Schema().Fields())
        fieldList[3].Type = bldr.Type()
        meta := rec.Schema().Metadata()
        outSchema = arrow.NewSchema(fieldList, &meta)
    }

    ch2 <- array.NewRecord(outSchema, cols, rec.NumRows())
    rec.Release()
}
```

Manipulating the Column

Let’s dig into this a bit

Trust me, it’s easier than it looks!
Follow along for the next few slides...

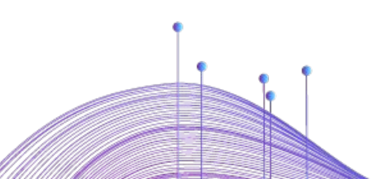
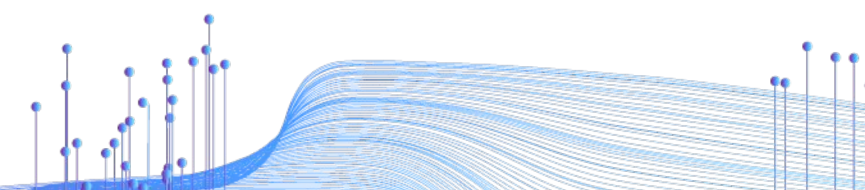
First: A ListBuilder

`` `[{'id': 123, 'name': 'Comedy'}, {'id': 456, 'name': 'Drama'}]` ``

```go
// we need to know the fields we're expecting in this JSON string
// for this example, we'll hardcode it
bldr := array.NewListBuilder(memory.DefaultAllocator, arrow.StructOf(
    arrow.Field{Name: "id", Type: arrow.PrimitiveTypes.Int32},
    arrow.Field{Name: "name", Type: arrow.BinaryTypes.String},
))
defer bldr.Release()
```

Builders are reusable

Create a List Column of Structs

Next: Build Replacement Column

Example is just one column, but could be any number of columns in parallel

```
for rec := range ch {
    genresCol := rec.Column(3).(*array.String)

    bldr.Reserve(int(rec.NumRows()))
    for i := 0; i < genresCol.Len(); i++ {
        if genresCol.IsNull(i) {
            bldr.AppendNull()
            continue
        }

        // convert single quotes to dbl quotes
        // we want valid JSON
        vals := reID.ReplaceAllString(re.ReplaceAllString(genresCol.Value(i),
            `"${key}": "${dbl}${sngl}"`), `"id": $1`)
        if err := bldr.UnmarshalJSON([]byte("[" + vals + "]")); err != nil {
            panic(err)
        }
    }

    cols := rec.Columns()
    // modify the slice of arrays
    // new record doesn't copy the columns!
    cols[3].Release()
    genreCol := bldr.NewArray()
    cols[3] = genreCol
```

Grab column we want

Could find index via Schema with `FieldIndices` method

Parse JSON directly

UnmarshalJSON on a builder parses the JSON and adds the values to the builder

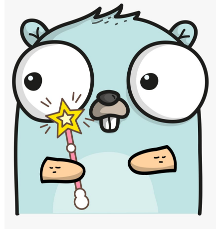
Next: Send the New Record

It's a pointer! There's no copying!

```
    // if we don't know the entire schema beforehand, we can just copy the existing
    // schema and replace the field for the column we're altering
    if outSchema == nil {
        fieldList := make([]arrow.Field, rec.NumCols())
        copy(fieldList, rec.Schema().Fields())
        fieldList[3].Type = bldr.Type()
        meta := rec.Schema().Metadata()
        outSchema = arrow.NewSchema(fieldList, &meta)
    }

    ch2 <- array.NewRecord(outSchema, cols, rec.NumRows())
    rec.Release()
}
```

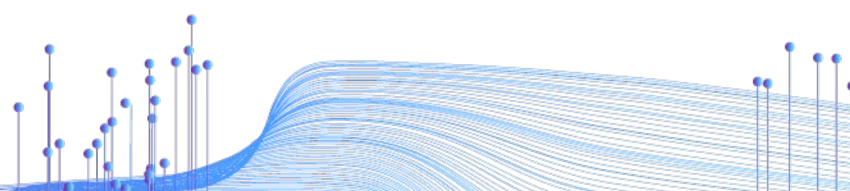
Create the Output Schema

Check if we have it already so we only create it once.

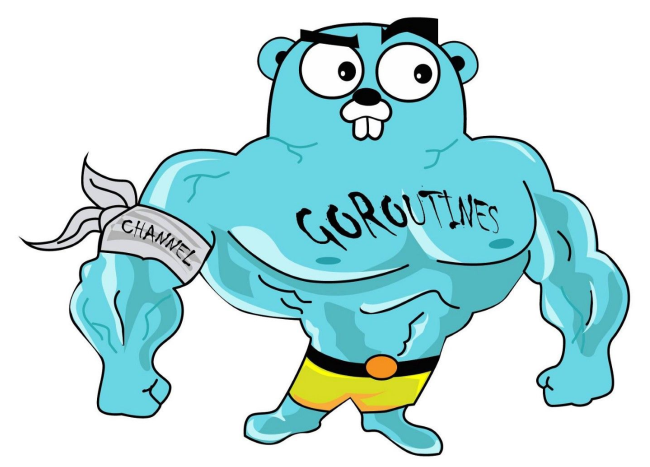


Send the New Record

Pass the new record to a different channel, continuing the pipeline

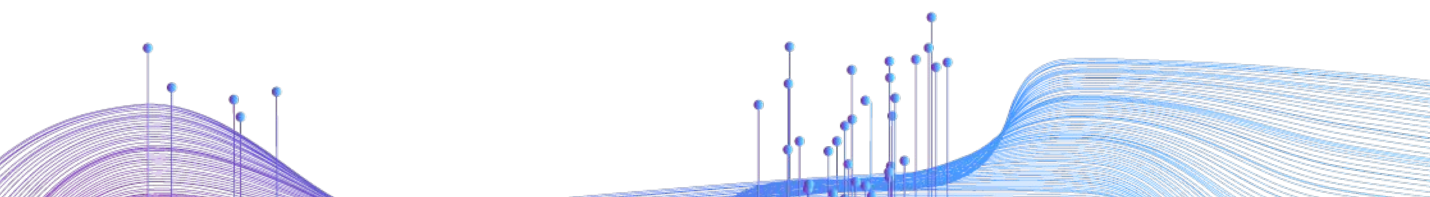


Improvement: Parallelize

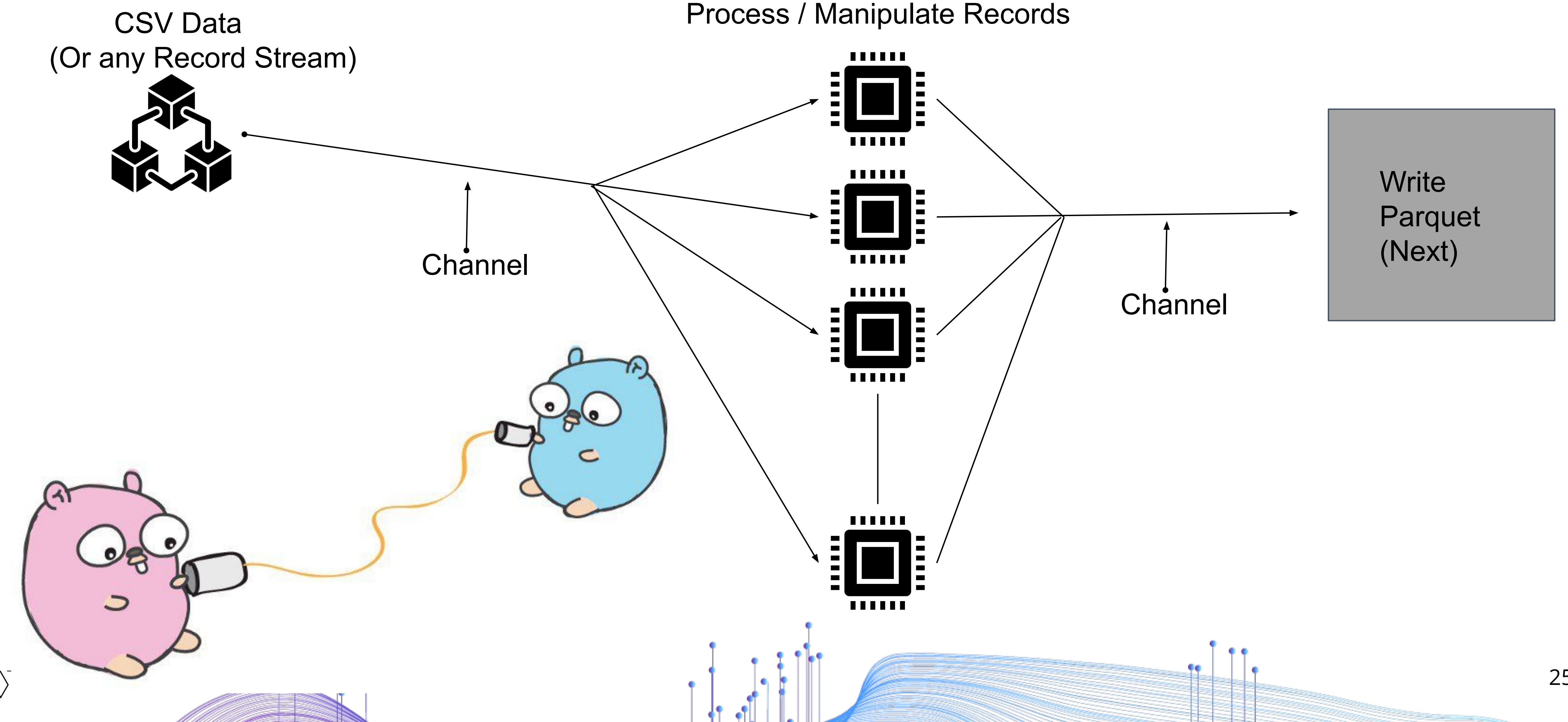



```go
var wg sync.WaitGroup
const nworkers = 10
wg.Add(nworkers)
for i := 0; i < nworkers; i++ {
    go func() {
        defer wg.Done()
        // put record manipulation here
    }()
}

go func() {
    wg.Wait()
    close(ch2)
}()
```

Goroutines and Channels for extremely easy parallel patterns such as fan-out/fan-in



Recap: Pipeline So far…

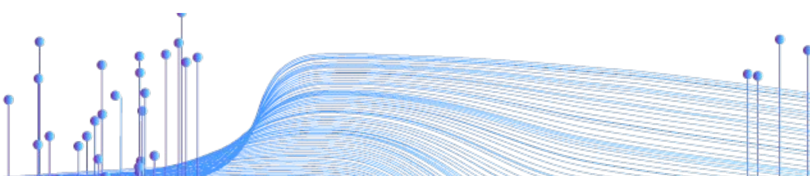


```go
pqout, err := os.Create("movies_metadata.parquet")
if err != nil {
    panic(err)
}

firstRec := <-ch2

wr, err := pqarrow.NewFileWriter(firstRec.Schema(), pqout,
    parquet.NewWriterProperties(
        parquet.WithCompression(compress.Codecs.Snappy),
        parquet.WithCompressionFor("overview", compress.Codecs.Zstd),
        parquet.WithDictionaryDefault(false),
        parquet.WithDictionaryFor("original_language", true),
        parquet.WithDictionaryFor("status", true),
        parquet.WithEncodingFor("id", parquet.Encodings.DeltaBinaryPacked),
        parquet.WithDataPageVersion(parquet.DataPageV2),
        parquet.WithVersion(parquet.V2_LATEST),
    ), pqarrow.DefaultWriterProps())
if err != nil {
    panic(err)
}
defer wr.Close()
firstRec.Release()

for rec := range ch2 {
    wr.Write(rec)
    rec.Release()
}
```

Write a Parquet File

Columnar file storage

Optimized Arrow -> Parquet conversion



Reader and Writer use io Interfaces

Easy reading and writing of data regardless of location

(S3, ADLS, HDFS, etc.)

Parquet

Reader requires `io.ReaderAt` and `io.Seeker`

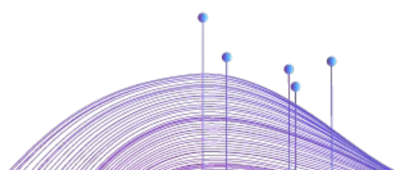
Writer only needs `io.Writer`, great for Streams

Can read Parquet data and metadata directly or convert directly to/from Arrow

CSV

Only needs `io.Reader` and `io.Writer`

Control memory usage via Chunk options



What about between processes?

https://arrow.apache.org/docs/format/Flight.html

Efficient Data Transportation

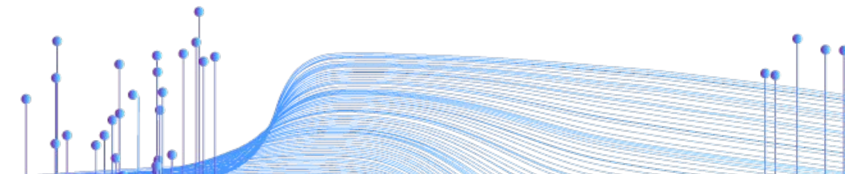
Arrow IPC

- Communicate record batches locally or remotely
- File and streaming formats
- Can mmap for efficiency

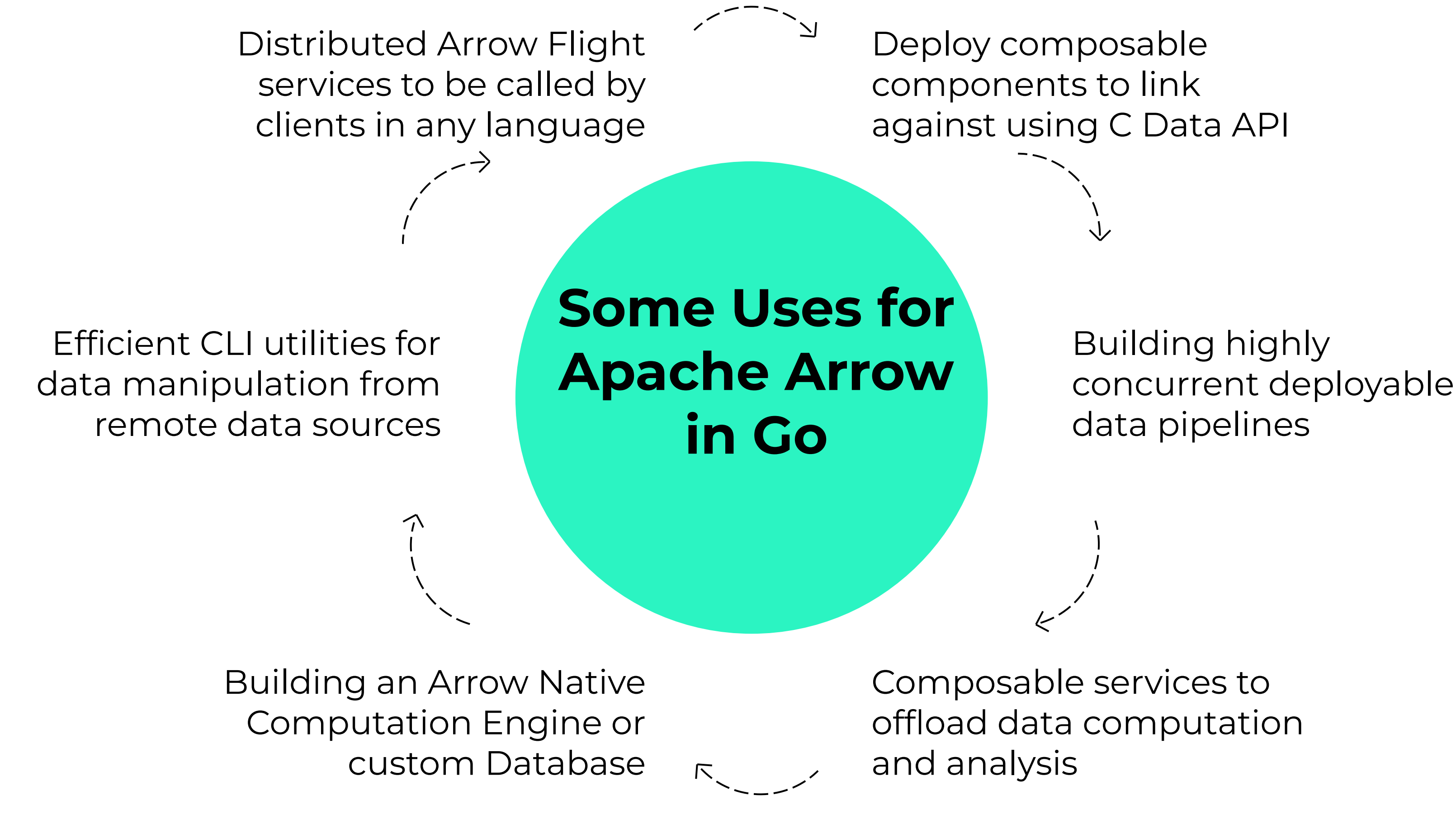
Source

Arrow Flight RPC
Arrow Flight SQL

- Protobuf + Arrow IPC streams
- Standardized Protocol for many clients
- >20x faster than ODBC



What else can it do?



Want more examples?

More on Apache Arrow: https://arrow.apache.org/docs/

Or get my book!

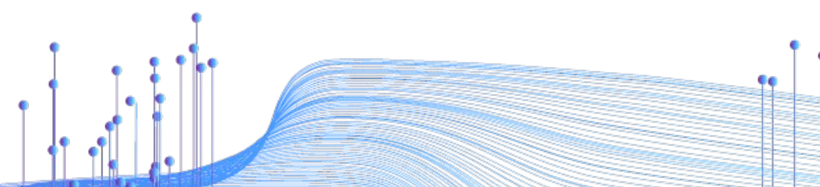
Examples in multiple languages: Python / C++ / Go

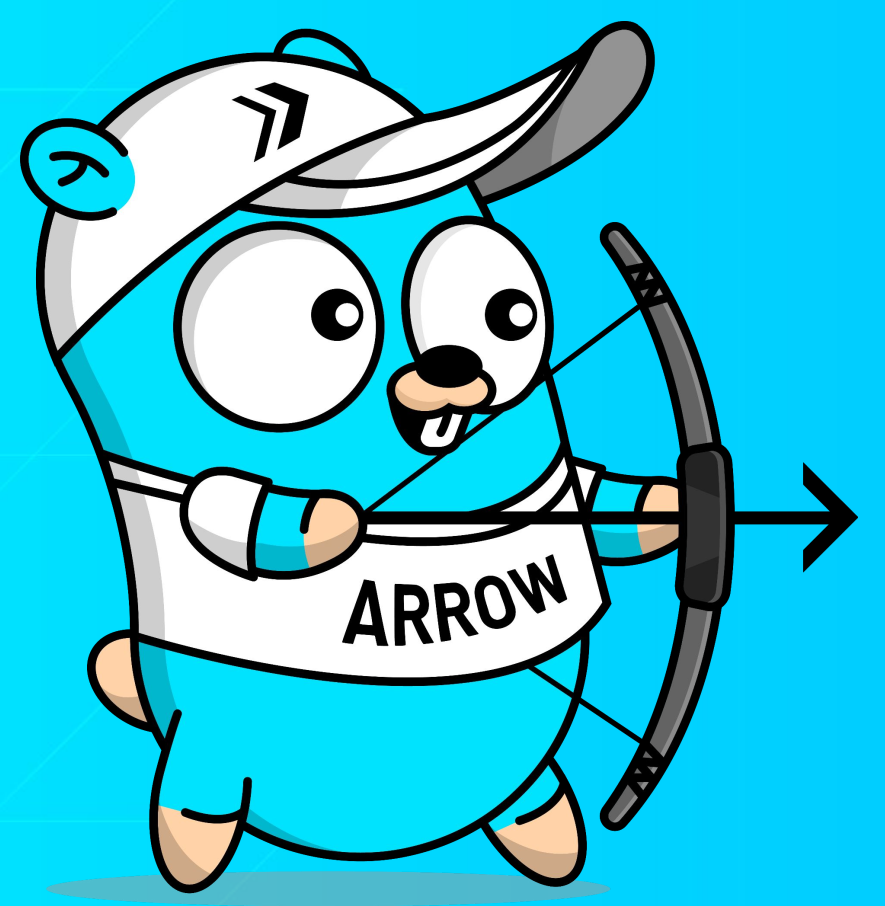
Practical Examples for Arrow Flight and other Data Science workflows

Amazon Link for the Book: buff.ly/3OcoxyB

"In-Memory Analytics with Apache Arrow"

Go Arrow/Parquet docs: https://pkg.go.dev/github.com/apache/arrow/go/v9





Q&A

Thanks Everyone!

The Go Gopher image is released under the Creative Commons Attribution 3.0 License, originally created by artist Renee French

XKCD Comics released under Creative Commons Attribution-NonCommercial 2.5 License, created by Randall Munroe https://xkcd.com