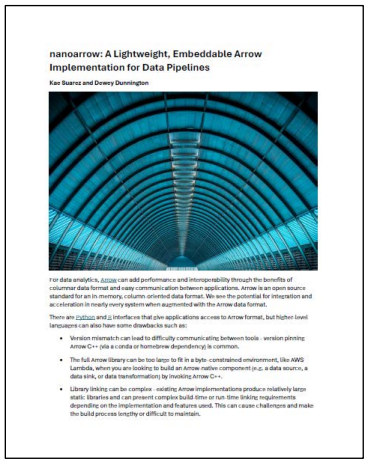
VOLTRON DATA

carahsoft.


nanoarrow: A Lightweight, Embeddable Arrow Implementation for Data Pipelines

Kae Suarez and Dewey Dunnington

yl·y2l·NJNJ2øY ! [ƛƎKűøŜƛƎKűΣ 9Yōŝ ŘŘl·ōfŜ !NJNJ2ø LYLJfŜYŜyűl·űƛ2yŦ2NJ 5l·űl· tƛLJŜfƛyŜă

Thank you for downloading this Voltron Data resource. Carahsoft is the distributor for Voltron Data's AI and ML solutions available via NASA SEWP V, ITES-SW2, NASPO ValuePoint, and many more contract vehicles.

To learn how to take the next step toward acquiring Voltron Data's solutions, please check out the following resources and information:

For Voltron Data overview:
carah.io/Voltron-Data

For upcoming events:
carah.io/Voltron-Events

For additional resources:
carah.io/Voltron-Resources

For additional Artificial Intelligence:
carah.io/ai-solutions

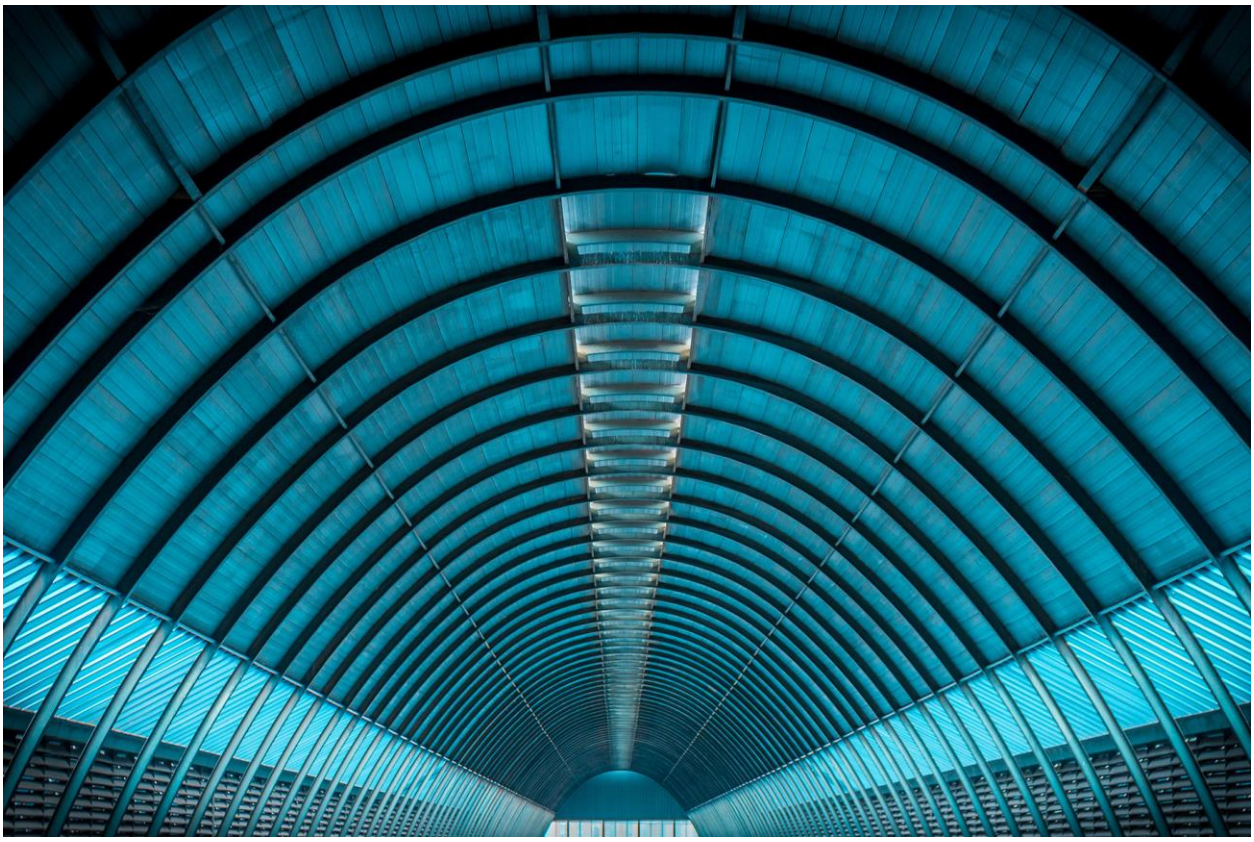
To set up a meeting:
VoltronData@carahsoft.com

To purchase, check out the contract vehicles available for procurement:
carah.io/procurement

For more information, contact Carahsoft or our reseller partners:
VoltronData@carahsoft.com | 888-606-2770

nanoarrow: A Lightweight, Embeddable Arrow Implementation for Data Pipelines

Kae Suarez and Dewey Dunnington



For data analytics, Arrow can add performance and interoperability through the benefits of columnar data format and easy communication between applications. Arrow is an open source standard for an in-memory, column-oriented data format. We see the potential for integration and acceleration in nearly every system when augmented with the Arrow data format.

There are Python and R interfaces that give applications access to Arrow format, but higher-level languages can also have some drawbacks such as:

- Version mismatch can lead to difficulty communicating between tools - version pinning Arrow C++ (via a conda or homebrew dependency) is common.
- The full Arrow library can be too large to fit in a byte-constrained environment, like AWS Lambda, when you are looking to build an Arrow-native component (e.g. a data source, a data sink, or data transformation) by invoking Arrow C++.
- Library linking can be complex - existing Arrow implementations produce relatively large static libraries and can present complex build-time or run-time linking requirements depending on the implementation and features used. This can cause challenges and make the build process lengthy or difficult to maintain.

So with that in mind, we want to supply an answer to the question: what if we want to build an optimized component in an Arrow-based pipeline, without any of this overhead? Luckily, there are also implementations in C/C++ where applications can benefit from performance and benefits that come with low-level languages. The one we will highlight here, **nanoarrow**, is a lightweight library that can be used to quickly leverage the power of the Arrow format with none of the overhead. It provides basic data structures, memory management, and stream management — no more, no less.

nanoarrow Use Case

Let's say you're building a lightweight application to gather logs from multiple sources for consumption. The system that receives the logs is small, maybe even an embedded system, and you want to just get a stream of Arrow data out of it that can be grabbed with Arrow Database Connectivity (ADBC). In order to ***be*** lightweight, you can leverage nanoarrow.

The end user does not have any knowledge of the underlying implementation — in fact, this is a strength of the ADBC driver manager. For a Python user, accessing the stream as an ADBC connection could look like:

PYTHON

```python
import adbc_driver_manager
import pyarrow as pa

db = adbc_driver_manager.AdbcDatabase(
    driver="build/libadbc_simple_csv_driver.dylib",
    entrypoint="SimpleCsvDriverInit"
)

conn = adbc_driver_manager.AdbcConnection(db)
stmt = adbc_driver_manager.AdbcStatement(conn)
stmt.set_sql_query("test.csv")
array_stream, rows_affected = stmt.execute_query()
reader = pa.RecordBatchReader._import_from_c(array_stream.address)
reader.read_all()
```

And for an R user:

R

```r
library(adbcdrivermanager)

simple_csv_drv <- adbc_driver(
  "build/libadbc_simple_csv_driver.dylib",
  "SimpleCsvDriverInit"
)

adbc_database_init(simple_csv_drv) |>
  read_adbc("test.csv") |>
  as.data.frame()
```

Of course, this is within one system for testing — but that ADBC connection could point anywhere.

As for the underlying driver, the basic premise is to build a standard CSV reader using a file stream and use nanoarrow to direct the data into Arrow arrays with their accompanying schema. From there, nanoarrow can also be used to put those arrays into a stream, which can ultimately be accessed from the end application via ADBC.

We exclude the code for the driver here for brevity, but if you want to see how this can look, refer to the repository.

The code would be similar for tasks such as passing data up for visualization or machine learning, or structuring data into columnar form for more efficient analytics. No matter what, ***you*** know what your system needs and nanoarrow can get you exactly that, without having to worry about the latest linking conflicts.

So, how does this all work under the hood? What makes nanoarrow so lightweight, and how does ADBC magic obscure this all from end users?

Arrow Database Connectivity (ADBC)

ADBC is a database connector, much like JDBC and ODBC. It implements database connections with drivers, much like JDBC and ODBC, but is made for columnar environments — enabling the high-performance gains from the Arrow columnar format (learn why Snowflake implemented ADBC in this post). It comes with a driver manager to allow you to load drivers on the fly, which we leveraged above. Because the manager can delegate all the functions to the driver properly, users only have to load the drivers and never have to think about what's under the hood.

Today, though, we'll peer past that to explore how nanoarrow fueled our use case.

Arrow C Interfaces

nanoarrow works by wrapping the Arrow C interfaces, adding functionality to powerful fundamental tools.

The Arrow C data and Arrow C streaming interfaces are minimalist implementations of Arrow schema, arrays, and streams, enabling the use of Arrow anywhere C is available. This solution aims to:

- Expose the ABI-stable Arrow C Data and Arrow C Stream interfaces
- Make it easy for third-party projects to implement support for Arrow (including partial support where sufficient)
- Allow zero-copy sharing of Arrow data between independent runtimes and components running in the same process
- Match the Arrow array concepts closely to avoid the development of yet another marshaling layer
- Avoid the need for one-to-one adaptation layers such as the limited JPype-based bridge between Java and Python

- Enable integration without an explicit dependency (either at compile-time or runtime) on the Arrow software project

These can be implemented by just copying them into your code base. However, using the structures the Arrow C interfaces implement takes effort. This is *because* the interfaces are minimal, implementing only data structures without any helpers like constructors. However, since these are just C structures, helpers would not be incompatible, and users would likely implement helpers on their own.

nanoarrow was built to make deploying the Arrow C interface in an application easy.

nanoarrow

nanoarrow enhances the Arrow C experience by implementing the helper functions for you while maintaining a small size and all the other advantages of the Arrow C interfaces.

With nanoarrow in hand, using these lightweight, highly-portable interfaces is easy, enabling use cases such as:

- Building Arrow-native components for byte-constrained environments.
- Moving and parsing Arrow data in a way that is resilient to changes in packaged versions of Arrow.
- Interacting with data feeds like Arrow Database Connectivity (ADBC) cheaply.

To understand nanoarrow, it's most useful to define what it does — and does not do. nanoarrow provides functionality to provide:

- Types: nanoarrow implements read and write support for Arrow's rich set of built-in types and community-contributed extension types. nanoarrow has the tools you need to communicate an output type, allow a user to communicate an input type, or provide an implementation of your own extension type.
- Memory allocation and management: nanoarrow implements a growable buffer and Array builders to help you build columnar data from row-based storage. If your data is already columnar (e.g., std::vector<>), nanoarrow's buffer abstraction can provide a zero-copy wrapper around your data exported as an Arrow array.
- Stream management: The ArrowArrayStream (i.e., a stream of record batches) is the most widely adopted structure in the Arrow C interface. nanoarrow provides helpers to export a compliant ArrowArrayStream representing a streaming source or fully in-memory structure like a PyArrow Table.
- Native C++ compatibility through a nanoarrow.hpp : nanoarrow is written in C to maximize portability and minimize compile time; however, it provides C++-style memory management and error propagation helpers and integrates well into an existing C++ code base.
- Distribution: nanoarrow is distributed as just two files: nanoarrow.c and nanoarrow.h. These files are designed to be copy/pasted into an existing code base, making it possible to

embed nanoarrow in a standalone Python package, a standalone R package, or an independent C/C++ library. nanoarrow's contribution is typically a few hundred KB, so you can embed it in one or many components in your pipeline.

For the user, this leaves computation, moving data from files into streams from the interface, and so on. If you are using or considering using a native Arrow implementation like Arrow C++ to implement an optimized data producer, data consumer, and/or compute function, nanoarrow may be right for you.

But if you have a highly custom workflow that does not need the built-in utilities, or only need the data format or stream interface, then nanoarrow is an optimal choice for you.

Conclusion

The largest benefit of Arrow for any application is the Arrow columnar format, both for performance and interoperability with the wide variety of tools that support Arrow. nanoarrow gives you that interoperability and performance, easy setup for streaming data, and utilities to get you started. Minimize the complexity of your stack with a library that can be imported with just a copy + paste, and get the computational benefits and interoperability of Arrow. It's that easy.

Voltron Data designs and builds composable data systems using standards within the Arrow ecosystem like Arrow Database Connectivity and nanoarrow. Learn more about the standards we use to augment data systems and unlock interoperability or visit our Product page to learn how we can support your organization.