*System overview on ArcticDB*

Purushottam Panta, PhD student, Department of Computer Science, University of Kentucky

This technical document lays out the details of installing a high-performance, serverless database system, ArcticDB. We comprehend basic read and write operations, indexing, analytical processing and transaction processing on ArcticDB through this document. We also walked through the details of the storage structure architecture of ArcticDB database system.

# ArcticDB overview

ArcticDB is a high performance Data frame database written in C++, designed and built on the python data science ecosystem.

ArcticDB is a serverless database system that is designed primarily to integrate with the Python and Pandas development ecosystem. It doesn’t require any additional infrastructures than running python environment with object storage being accessible. ArcticDB supports streaming data ingestion and supports data with or without schema. This database system can process millions of on-disk data rows (tuples) in seconds with its incredibly fast performance making it outstanding for analytical (OLAP) workloads, rather than the transactional (OLTP) workloads..

There are some functionalities that ArcticDB supports more than any other columnar database systems, such as Apache Parquet, HDF:

1. It provides time-series indexes and versioned modification (travel over the different versions of the data.
2. Data storage in ArcticDB are not structured as in raw file-path like various other columnar database systems has; it rather structured into libraries and symbols.
3. It supports both batch and streaming data for speedy data processing and storage.
4. Supports dynamic schemas – changing schemas (set of columns) over time through its versioning.
5. ArcticDB is most suited for OLAP (rather than OLTP) database system that optimizes and speeds up large numerical dataset and queries that operates over bulk of rows at a time.
6. ArcticDB is fully fledged embedded analytical db system that it doesn’t require a server to take advantage any of the core features.
7. ArcticDB storage engine was written in C++, and designed to be compatible with modern cloud and on-premises object storage. Python API with pandas, Dataframes can be use to have data-interaction with ArcticDB, it doesn’t support query languages such as SQL.

# ArcticDB Installation and Fundamentals

## Installation:

Python version 3.6 to 3.11 supports ArcticDB. We can use any development environment that supports running python. It can be a google-colab or python notebook on anaconda or so. I have been using google-colab for further exercise of ArcticDB hereon. It can simply be installed with “install arcticdb” command in python running environment. In the following screen I have installed arcticdb and imported some useful libraries to build datasets and interact with arcticdb:

A screenshot of a computer

Description automatically generated

## Basic Operations:

In this section we are going to create a simple data frame to insert dataset into ArcticDB and perform some read-write operation on it.

*Operation1:* Creating DataFrame for some small sample dataset:

A screenshot of a computer

Description automatically generated

*Operation2:* Building Libraries with “ts\_daily1” dataset we just created:  
A computer code with text

Description automatically generated

*Operation3:* Let’s write the data through the library (note the data version)

A screenshot of a computer

Description automatically generated



*Operation4:* Reading data:

A screenshot of a computer

Description automatically generated



A screenshot of a computer

Description automatically generated

*Operation5:* We are appending “ts\_daily2” data in it:

A screenshot of a computer

Description automatically generated

*Operation6:* Updating the data with “ts\_dly3”:

A screenshot of a computer

Description automatically generated

(**Note**: when performing the update, newer data from “ts\_dly3” (dated from 2024-01-03, 2024-01-04, 2024-01-05, 2024-01-06, and 2024-01-07 have updated the existing columns in the database, they are actually versioned, and the newer version is shown up by default).

*Operation7:* Since all operations in ArcticDB are being tracked with version. Let’s rewind them:

A screenshot of a computer

Description automatically generated

# ArcticDB Storage Structure

## ArcticDB Layered Architecture

ArcticDB is a columnar optimized data storage that supports time-series data with versioning on them.  ArcticDB can be considered as a more customizable storage format, which is different from any other storage design having columnar formats [1][2]. Also, as a client-side library, it can be scaled out and support modern S3 (Amazon) storage. It currently supports the following two backends:

import arcticdb as adb

*A Lightning Memory-Mapped Database Manager (LMDB):* adb.Arctic('lmdb://path/to/desired/database')

*Any S3 API Compatible storage Simple Storage Service(S3) or Azure Blob Storage:*

adb.Arctic('s3 (or s3s for https)://ENDPOINT:BUCKET?region=my\_region&access=ABCD&secret=DCBA')

ArcticDB data structure is divided into key-value pairs. For instance, when backed by S3, the key is the path in that S3 storage bucket corresponding to the value in that location. ArcticDB data storage format consists of four layers:

1. Reference Layer:

In this layer, ArceticDB maintains a pointer actively pointing to the next layer (Version Layer) linked list, which helps fast retrieval of the latest version of the symbol. Also, note that ArceticDB’s storage format is **mutable** only in the Reference Layer - each reference-layer-key can be overwritten.

Where:

prefix: Key prefix

vref: Key Type

\*: Delimiter

s: Symbol data type

U: Type of index

t: Format type (Binary symbol name)

symbol\_01: Key Unique ID

1. Version Layer:

This layer basically contains a linked list of **immutable** atom keys (key-value pairs). Each linked list elements have two pointers in the data segment – one pointing to the next version entry in the link list and the other pointing to an index key which provides the rout to the next layer (Index Layer) in the architecture. So, traversing through the linked lists in the version layer allows us to travel backward through time and retrieve the previous versions.

Where:

prefix: Key Prefix

ver: Key Type ver for Version key

s: Symbol data type

T: Index type (Timestamp index)

t: Format type (Binary symbol name)

symbol\_01: Key Unique ID

\*: delimiter

31: Key Version Identifier (Atom keys have also version associated with each.)

16651…: Unix created Timestamp

176662…: Content hash

\*0\*0: Start / End timestamp unused for ver keys.

1. Index Layer:

Index Layer is also an **immutable** layer, which primary function is to provide the B-tree index over the next layer (Data Layer). Each pointer from this layer is basically a key containing a data segment residing in the Next (Data) Layer

1. Data Layer:

The Data Layer is also **immutable** layer containing compressed data segments. The data frames provided by user-agent are **sliced by both column and rows,** which facilitates speedy search over date-range (rows) and columns during read operations (Which are defined by row\_per\_segment and columns\_per\_segment library configuration options).

## Feature selection from the source data**TODO**

No funding or financial assistantship was in place for this research and literature.

Feature selection helps in building more efficient, accurate, and interpretable machine learning models by focusing on the most relevant information while discarding noise and irrelevant data. We basically identify prominent features from the dataset for further process and build the machine learning models on them.

# ARCTICDB Indexing and Compression techniques

Indexes are the powerful tools being used in the database in order to obtain rapid speed up in query response. Index is used to quickly look up the dataset like an index provided in the back of the book, so being able to perform queries much faster.

A full index gets constructed in ArcticDB for numerical and time series (such as DataTimeIndex) pandas indexes. It will help to optimize the slicing across the index entries. Data can still be stored impacting the performance of row-slicing (slowing down) when the index is not sorted or not numeric in the dataset.

Arctic is a base ground-up of now what we have ArcticDB with LZ4, a byte-oriented compression algorithm being used on it. LZ4 compression allows the original dataset to be perfectly reconstructed without loss from the compressed data, in contrast to the lossy compression techniques which permit approximate reconstruction as needed. There are compression settings available in the Arctic “**\_config.py**” configuration file[7]:

## **DISABLE\_PARALLEL:** The LZ4 compression in parallel (multi-threading) is enabled by default, hurting the performance.

export DISABLE\_PARALLEL=1 (or 0)

ENABLE\_PARALLEL = not os.environ.get('DISABLE\_PARALLEL')

## **LZ4\_HIGH\_COMPRESSION:** It has tradeoff between runtime speed and compression ratio.

export LZ4\_HIGH\_COMPRESSION = 1 (or 0)

LZ4\_HIGH\_COMPRESSION = bool(os.environ.get('LZ4\_HIGH\_COMPRESSION'))

## **LZ4\_WORKERS:** Being used to configure the compression thread pool size (default-size: 2 for non-high-compression, 8 for high-compression)

export LZ4\_WORKERS=##

LZ4\_WORKERS = os.environ.get('LZ4\_WORKERS', 2)

## **LZ\_N\_PARALLEL:** Min # of chunks required to use parallel compression (Default = 16)

export LZ4\_N\_PARALLEL=##

LZ4\_N\_PARALLEL = os.environ.get('LZ4\_N\_PARALLEL', 16)

## **LZ4\_MINSZ\_PARALLEL:** Min data size required to use parallel compression (Default = ~0.5MB)

export LZ4\_MINSZ\_PARALLEL=#####

LZ4\_MINSZ\_PARALLEL = os.environ.get('LZ4\_MINSZ\_PARALLEL', 0.5 \* 1024 \*\* 2) # 0.5 MB

# Conclusion

We have taken random forest and support vector machine and long short-term memory algorithms to classify the demented and nondemented subject in our dataset. We then will be evaluating the performance of each of them with AUC score and recall score.

## Random Forest

Random Forest is an ensemble learning method that constructs multiple decision trees and combines their predictions to make

## Support Vector Machine (SVM)

A Support Vector Machine (SVM) is a supervised machine l

## Long Short-Term Memory (LSTM):

Long Short-Term Memory (LSTM) is a type of recurrent neural

## Results and Future works

1. Performance of the classification
2. Three different classification models were tested and measured the performance in terms of AUC

##### References

1. ArcticDB On-Disk Storage Format, URL: [https://docs.arcticdb.io/4.2.1/technical/on\_disk\_storage/](https://docs.arcticdb.io/4.2.1/technical/on_disk_storage/l)
2. Storage Models and Data Layouts, URL: <https://15721.courses.cs.cmu.edu/spring2023/slides/03-storage.pdf>
3. ArcticDB Source Code, URL: <https://github.com/man-group/ArcticDB>
4. ArcticDB Technical Documentation, URL: <https://docs.arcticdb.io/>
5. ArcticDB product website, URL: <https://arcticdb.io/>
6. Loseless Compression, URL: <https://www.sciencedirect.com/topics/engineering/lossless-compression>
7. ArcticDB Configuration, URL: https://github.com/man-group/arctic/blob/master/arctic/\_config.py