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.
***keywords:* Online Analytical / Transaction Processing (OLA/TP)**

I. 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:

a. It provides time-series indexes and versioned modification (travel over the different versions of the data.

b. 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.

c. It supports both batch and streaming data for speedy data processing and storage.

d. Supports dynamic schemas – changing schemas (set of columns) over time through its versioning.

e. 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.

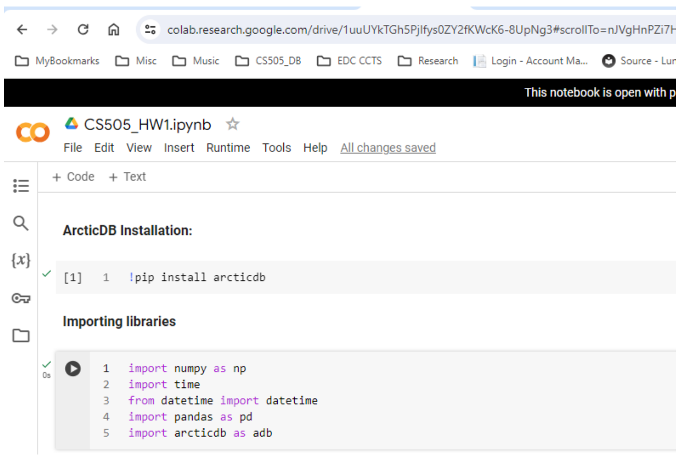
f. ArcticDB is fully fledged embedded analytical db system that it doesn't require a server to take advantage any of the core features.

g. 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.

II. ArcticDB Installation and Fundamentals

INSTALLATION AND IMPORTING LIBRARIES:

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 this database system:



ArcticDB Installation:

```
[1] !pip install arcticdb
```

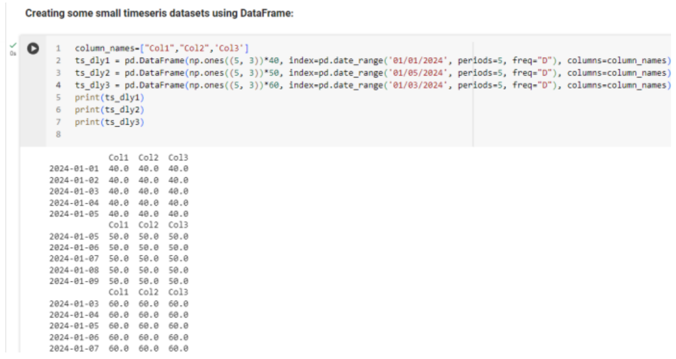
Importing libraries

```
import numpy as np
import time
from datetime import datetime
import pandas as pd
import arcticdb as adb
```

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.

<u>Operation1:</u> Creating DataFrame for some small sample dataset:

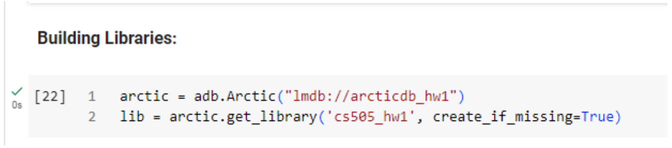


Creating some small timeseris datasets using DataFrame:

```
column_names=["Col1","Col2",'Col3']
ts_dly1 = pd.DataFrame(np.ones((5, 3))*40, index=pd.date_range('01/01/2024', periods=5, freq="D"), columns=column_names)
ts_dly2 = pd.DataFrame(np.ones((5, 3))*50, index=pd.date_range('01/05/2024', periods=5, freq="D"), columns=column_names)
ts_dly3 = pd.DataFrame(np.ones((5, 3))*60, index=pd.date_range('01/03/2024', periods=5, freq="D"), columns=column_names)
print(ts_dly1)
print(ts_dly2)
print(ts_dly3)
```

```
            Col1  Col2  Col3
2024-01-01  40.0  40.0  40.0
2024-01-02  40.0  40.0  40.0
2024-01-03  40.0  40.0  40.0
2024-01-04  40.0  40.0  40.0
2024-01-05  40.0  40.0  40.0
            Col1  Col2  Col3
2024-01-05  50.0  50.0  50.0
2024-01-06  50.0  50.0  50.0
2024-01-07  50.0  50.0  50.0
2024-01-08  50.0  50.0  50.0
2024-01-09  50.0  50.0  50.0
            Col1  Col2  Col3
2024-01-03  60.0  60.0  60.0
2024-01-04  60.0  60.0  60.0
2024-01-05  60.0  60.0  60.0
2024-01-06  60.0  60.0  60.0
2024-01-07  60.0  60.0  60.0
```

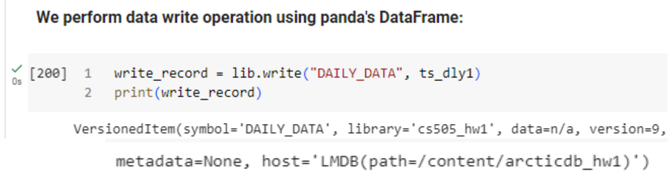
Operation2: Building Libraries with “ts_daily1” dataset we just created:



Building Libraries:

```
[22]  arctic = adb.Arctic("lmdb://arcticdb_hw1")
      lib = arctic.get_library('cs505_hw1', create_if_missing=True)
```

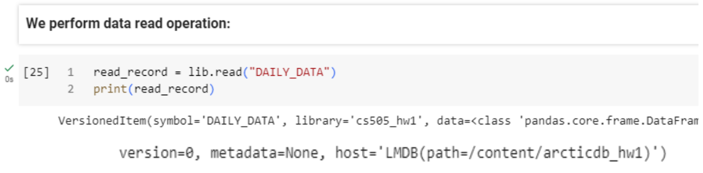
Operation3: Let’s write the data through the library (note the data version)



We perform data write operation using panda's DataFrame:

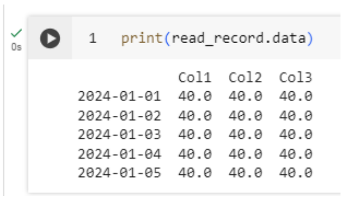
```
[200]  write_record = lib.write("DAILY_DATA", ts_dly1)
       print(write_record)

VersionedItem(symbol='DAILY_DATA', library='cs505_hw1', data=n/a, version=9,
      metadata=None, host='LMDB(path=/content/arcticdb_hw1)')
```

Operation4: Reading data:



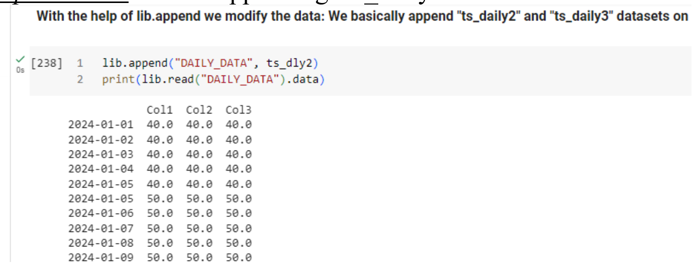
We perform data read operation:

```
[25]  read_record = lib.read("DAILY_DATA")
      print(read_record)

VersionedItem(symbol='DAILY_DATA', library='cs505_hw1', data=<class 'pandas.core.frame.DataFram
      version=0, metadata=None, host='LMDB(path=/content/arcticdb_hw1)')
```



```
print(read_record.data)

            Col1  Col2  Col3
2024-01-01  40.0  40.0  40.0
2024-01-02  40.0  40.0  40.0
2024-01-03  40.0  40.0  40.0
2024-01-04  40.0  40.0  40.0
2024-01-05  40.0  40.0  40.0
```

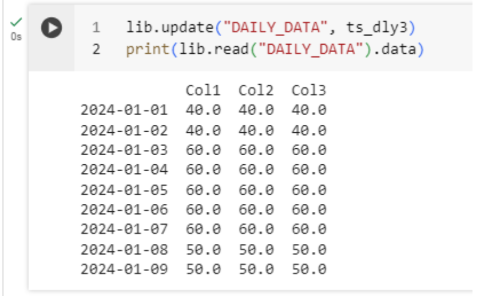
Operation5: We are appending “ts_daily2” data in it:



With the help of lib.append we modify the data: We basically append "ts_daily2" and "ts_daily3" datasets on

```
[238]  lib.append("DAILY_DATA", ts_dly2)
       print(lib.read("DAILY_DATA").data)

            Col1  Col2  Col3
2024-01-01  40.0  40.0  40.0
2024-01-02  40.0  40.0  40.0
2024-01-03  40.0  40.0  40.0
2024-01-04  40.0  40.0  40.0
2024-01-05  40.0  40.0  40.0
2024-01-05  50.0  50.0  50.0
2024-01-06  50.0  50.0  50.0
2024-01-07  50.0  50.0  50.0
2024-01-08  50.0  50.0  50.0
2024-01-09  50.0  50.0  50.0
```

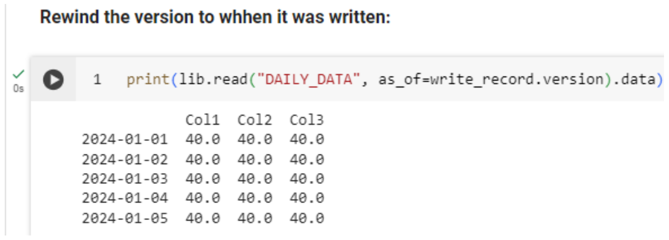
Operation6: Updating the data with “ts_dly3”:



```
lib.update("DAILY_DATA", ts_dly3)
print(lib.read("DAILY_DATA").data)

            Col1  Col2  Col3
2024-01-01  40.0  40.0  40.0
2024-01-02  40.0  40.0  40.0
2024-01-03  60.0  60.0  60.0
2024-01-04  60.0  60.0  60.0
2024-01-05  60.0  60.0  60.0
2024-01-06  60.0  60.0  60.0
2024-01-07  60.0  60.0  60.0
2024-01-08  50.0  50.0  50.0
2024-01-09  50.0  50.0  50.0
```

(**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 rather 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:



Rewind the version to whhen it was written:

```
print(lib.read("DAILY_DATA", as_of=write_record.version).data)

            Col1  Col2  Col3
2024-01-01  40.0  40.0  40.0
2024-01-02  40.0  40.0  40.0
2024-01-03  40.0  40.0  40.0
2024-01-04  40.0  40.0  40.0
2024-01-05  40.0  40.0  40.0
```

I put this exercise here: https://github.com/purupanta/Courses/blob/main/uky_cs505/hw1/cs505_hw1.ipynb

III. ArcticDB Storage Structure

ARCTICDB CONFIGURATION:

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&acces
s=ABCD&secret=DCBA')
```

In Memory configuration: More suitable when testing or not willing to create files in the disk.

```
ac = adb.Arctic('mem://')
```

ARCTICDB LAYERED ARCHITECTURE:

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 stores data in **columnar format, Decomposed Storage Model (DSM)** in the storage with key, value indexing in four different layers of the architecture:

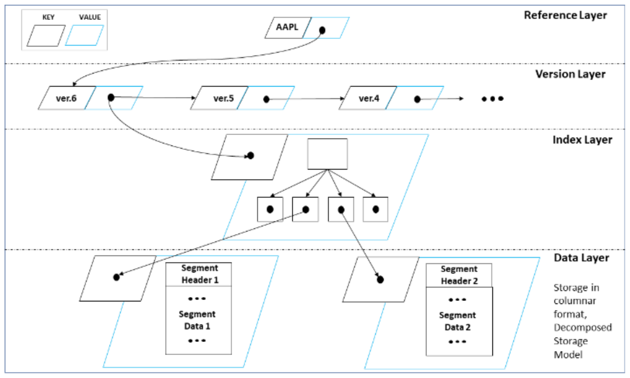


Fig: Layered architecture of ArcticDB

a. <u>Reference Layer:</u>

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.

<u>Example of Reference Key:</u>
```
prefix/vref/*sUt*symbol_01*
```

<u>Where:</u>
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

b. <u>Version Layer:</u>

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.

<u>Example of Atom Key:</u>
```
prefix/vref/*sTt*symbol_01*31*166511…*176662…*
```

<u>Where:</u>

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.

c. <u>Index Layer:</u>

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

d. <u>Data Layer:</u>

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).

IV. ARCTICDB INDEXING AND COMPRESSION TECHNIQUES

<u>INDEXING:</u>

Indexes are the powerful tools being used in the ArcticDB database to rapidly speed up 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.

An index gets constructed in ArcticDB for ordered 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 non-numeric in the dataset.

ArcticDB supports the following index types through pandas DataFrame:

pandas.Indes (int64 or float64)

RangeIndex (Some restrictions apply)

DatatimeIndex

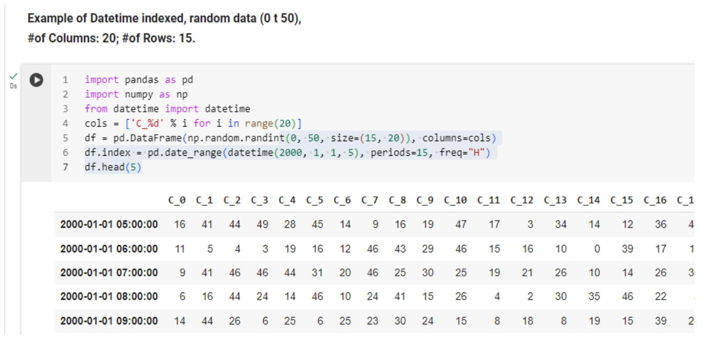
MultiIndex

OLAP: ArcticDB indexing may drastically reduce the amount of data (through row, column slicing and filtering) that needs to be read, processed, or sorted for ArcticDB engine. It also improves the search performance over reduced data returned through indexing. Overall, it helps to deliver the analytical output faster.

OLTP: ArcticDB indexing helps locating data quickly and helps to select out reduced-sized dataset and improves transaction performance.

For example, A datetime-indexed, random data (0 to 50) with (row, col) = (15, 20) can be created as:

```
import pandas as pd
import numpy as np
from datetime import datetime
cols = ['C_%d' % i for i in range(20)]
df = pd.DataFrame(np.random.randint(0, 50,
size=(15, 20)), columns=cols)
df.index = pd.date_range(datetime(2000, 1, 1,
5), periods=15, freq="H")
df.head(5)
```



Example of Datetime indexed, random data (0 t 50),
#of Columns: 20; #of Rows: 15.

```
import pandas as pd
import numpy as np
from datetime import datetime
cols = ['C_%d' % i for i in range(20)]
df = pd.DataFrame(np.random.randint(0, 50, size=(15, 20)), columns=cols)
df.index = pd.date_range(datetime(2000, 1, 1, 5), periods=15, freq="H")
df.head(5)
```

	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_10	C_11	C_12	C_13	C_14	C_15	C_16	C_1
2000-01-01 05:00:00	16	41	44	49	28	45	14	9	16	19	47	17	3	34	14	12	36	4
2000-01-01 06:00:00	11	5	4	3	19	16	12	46	43	29	46	15	16	10	0	39	17	1
2000-01-01 07:00:00	9	41	46	46	44	31	20	46	25	30	25	19	21	26	10	14	26	3
2000-01-01 08:00:00	6	16	44	24	14	46	10	24	41	15	26	4	2	30	35	46	22	
2000-01-01 09:00:00	14	44	26	6	25	6	25	23	30	24	15	8	18	8	19	15	39	2

COMPRESSION:

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. **The lossless compression, LZ4 enabled in ArcticDB reduces the performance of both OLAP and OLTP operations**.

There are compression settings available in the Arctic "**_config.py**" configuration file [7]:

1. ***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')
```

2. ***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'))

3. ***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)

4. ***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)

5. ***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

REFERENCES

[1] ArcticDB On-Disk Storage Format, URL: https://docs.arcticdb.io/4.2.1/technical/on_disk_storage/

[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 Souurce, URL: https://github.com/man-group/arctic/blob/master/arctic/_config.py