Implementing and Experimenting with The Distributed Database System TileDB

Marcus Casey
Department of Computer Science and Engineering at University of Nevada, Reno

Abstract*—This paper details Marcus Casey's project in CS491 in working with TileDB. While not accomplishing necessarily the achieved goal in the outlined project, Marcus gained a large understanding of TileDB and distributed database systems. This project was incredibly difficult, with many challenges in implementation. TileDB and AWS nodes are almost harder to configure than actually utilizing the system. In this paper, introduction to TileDB and its high-level functioning are detailed, as well as challenges incurred with this project. The paper will also detail how the system is implemented and functions and the limitations of TileDB and the created implementation for this project.

I. INTRODUCTION

TileDB is a novel storage manager for multi-dimensional arrays that can be utilized in scientific applications. TileDB can be utilized for both sparse and dense arrays. Sparse arrays represent an array when most of the array elements are empty stored as zero or null, or dense where every element contains a value. TileDB's main purpose is to sort data into collections regarded as fragments. A fragment can be dense or sparse and is grouped contiguous array elements into data tiles of a certain capacity.

A data tile is a regular sized chunk in the index space. When organized, the fragments then make random writes into sequential writes. A random write can be explained as writing to an SSD or data source the random write is searching, writing, searching writing based on a preset pattern. This can be good for smaller data sets with variables in performance being 10-20% extra time, however when scaled this can get absurd fast.

A sequential write is simply writing the values in no particular filter, simply in the order in which the data is stored. TileDB also implements a novel read algorithm which paired with its transformation of random writes, creates a very performant read system. TileDB is also considered a parallel system through multi-threading and multi-processing which allows thread and process safety through lightweight locking.

Lightweight locking is locking that occurs but takes very little CPU bandwidth and memory allowing the system to remain stable in case of failures or stale jobs. The researchers and authors of TileDB claim that the system is more performant in both reads and writes compared to the industry standard HDF5 and SciDB.

1.1 Workflow Overview

TileDB contains a C, C++, Ruby, Python, and soon-to-be more language API's. Contained in each API is a subset of handy features, the novel dense and sparse on-disk representations but also including tools for compressions and parallelism. TileDB's core treats arrays of arbitrary dimensionality and schema's; so it can be implemented on virtually any data set system.
Fragments, as mentioned earlier are the ordered collections that may overlap in the index space of the array and represent batches of updates to each. Array.

Dense fragments are grouped into regular-sized-chunks in the index known as data tiles. A sparse fragment is an element represented by storing its indices in a specific, global order. Sparse elements are grouped into data tiles of fixed capacity which can help balance input/output when utilizing the array. Fragments are a unique approach in writing performance, since they are performed in batches, each batch which can contain many small, random writes, is written to a fragment sequentially.

Meaning sparse fragments can be used to speed up random writes even when implemented into a dense array. These batches in random requests sequentially write many at a time as sparse fragments. This allows string representations and other variable-length values to be exported efficiently.

Many systems already turn random writes into sequential appends, however implementing that logic into array management systems is incredibly difficult and not a simple task. This is due to the fact that many fragments may contain the same logical region of an array; with some fragments being sparse, which can make it hard for the read algorithm to find the most recent fragment and the update to each returned element. TileDB contains a read algorithm that achieves this and avoids unnecessary reads when a fragment is totally replaced by a new fragment.

As the number of fragments grow and read performance degrades, it implements a consolidation algorithm that will then merge many fragments into one. This consolidation is performed in the background while other concurrent reads and writes occur.
TileDB's arrays function with a dimension and attribute variable. Each dimension contains a domain and all domains occupy the logical space of the array. When the dimension domain values combined, coordinates are created and a uniquely identify an array element (cell). A cell can be null or contain an attribute. Attributes can be ints, floats, and chars or a fixed or variable sized vector such as a string.

In dense arrays and sparse arrays covered early, only int and float values may be stored respectively. This is due to the coordinates of non-empty cells being able to be stored as real numbers and can allow multi-dimensional spaces without discretization (turning these values into different types). The last covered topic in this paper but perhaps one of the most important, is global cell order. Global cell order represents the linear order in which multi-dimensional data is stored to disk or memory.

The manner in which the data is ordered can greatly impact performance, since it is affecting which cells are near each other in storage. The mapping of thee multiple dimensions to the linear order is global cell mapping. In TileDB, the cells that are accessed together should be co-located on disk and memory, allowing shorter disk seeks, page reads, and cache misses.

TileDB considers the best choice of global cell order to be organized by application specific characteristics. An example TileDB provides is if an application reads data a row at a time, it should be laid out in rows.

If a column layout was used it would result in a larger amount of page reads. TileDB offers numerous amounts of methods to configure Global cell order and if interested the readers should refer to TileDB's official documentation. TileDB is stored in a directory in the file system on the machine.

The array directory as detailed by TileDB contains one sub-directory per array fragment, plus a file storing the array schema in binary format. Every fragment directory contains one file per fixed-sized attribute, and two files per variable sized attribute. TileDB compresses each attribute tile separately and based on compression type specified in the array schema. This allows quick back-ups and returns on arrays.

II. IMPLEMENTATION

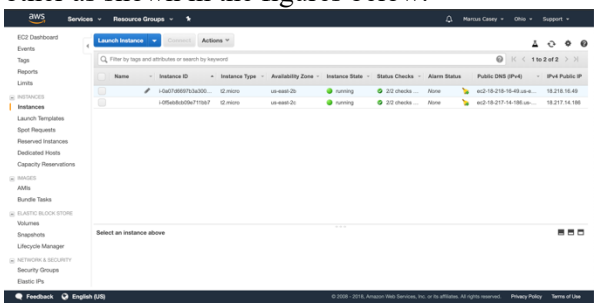
This research project was given five specific use cases in implementation with the overall use case of total size being 10X large than memory of the OS. The use cases were creation of five arrays:

- Array 1: 100% cells are non-empty values;
- Array 2: Only the first 10% of consecutive cells are non-empty values;
- Array 3: Only the first 50% of consecutive cells are non-empty values;
- Array 4: Only the last 10% of consecutive cells are non-empty values; and
- Array 5: 10% of random cells are non-empty values. (for simplicity, you can distribute the non-empty cells evenly within the array)

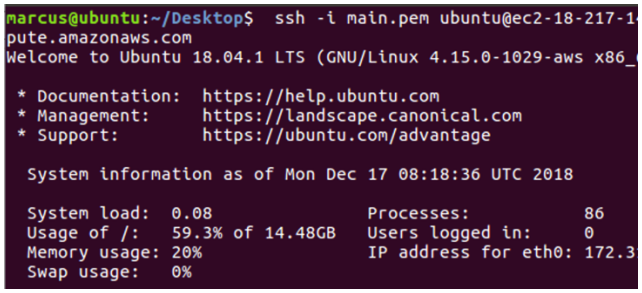
Each array was easily implemented based on the constraints on the array. At first all implementation was done on the author's home machine. However, these projects were to be achieved on a node-based system with both being able to communicate with each other via ssh.

To accomplish this implementation, the I created an auto-scaling group environment on the AWS management system with two fresh Ubuntu servers. Getting these servers configured and implemented was the bulk of the issue of the project.

As I had never used AWS, ssh, security keys, or basically any data distribution system the scale of TileDB, this was an enormous task. I struggled for a solid week on implementing my nodes. However, finally both servers were able to ssh which each other as shown in the figures below.

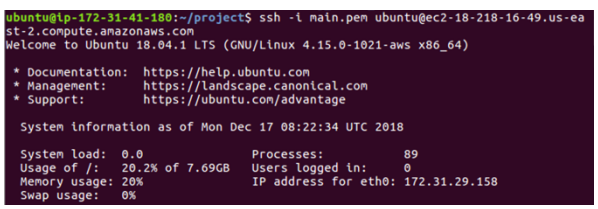


Working AWS instances



Connected to node 1

Now, once logged into the nodes I was able to implement TileDB. Each node can communicate with each other as seen in the figure below:



Connecting to node 2 from node 1

While not particularly impressive, achieving this was arguably way more effort than the actual effort of the research project. The systems encompassing AWS are not 'noob' friendly and are very difficult for first time users. However, as a student wanting to branch out more extensively into data and analytics, I knew this was a necessary evil I would have to grapple to understand as AWS is core to any data analytics firm or simple implementations.
When implementing TileDB I started with its C++ implementation. At first, the system worked and was able to compile. However, around December 13th, my C++ API implementation of TileDB simply stopped working.

No matter what I did I could not figure out what went wrong. I reset all of my environments and attempted to reinstall yet everything broke. Even on my home implementation C++ stopped working. I knew this had to be on my end configuration but realized I did not have the time to figure out exactly what was going on.

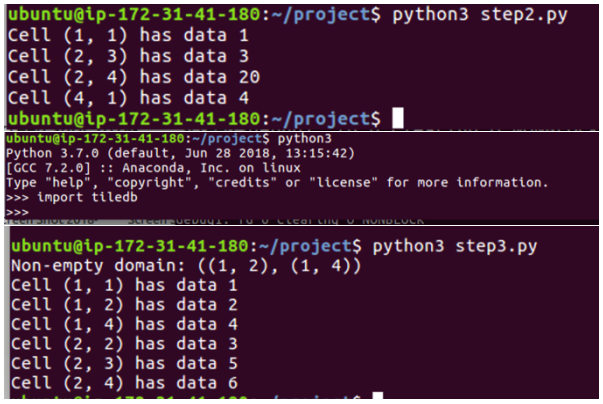
So, I made the decision to switch to Python which delayed my project and forced a week binge of re-implementing my work into Python. TileDB has a new Python API and framework being developed. I believe the Python API is what the creators will eventually settle with. This is due to the fact that having used both, I can say certainly the Python implementation of TileDB is simply superior in readability and execution.

The Python API is still under development, however had all the features required for this project. I simply installed TileDB and Python 3 on each instance and continued with my research.

Now, the actual implementations of the arrays went smoothly initially. Since the goal was simply % base and due to my lack of knowledge in the subject I simply created a 10-value dataset initially to test the theory of creating these arrays based on system memory. I have attached in my submission the base program I used to test these theories. Note that these are not in-depth, and I have other version that I am elaborating on to reach research conclusion.

The files included are the examples of sparse arrays

and creation of non-domain values as specified in the research project. As being unable to complete the specific requirements outlined in the project due to time-constraints I have included them, unfinished to specification.



Proof of TileDB implementation on AWS systems and execution of a sparse array with data and also a sparse array with empty data.

My code on this project can be found in the accompanying package submitted with this report. If for any reason someone is reading this and wants to view my code, they can email me at marcuscasey@unr.edu.

CONCLUSION

The conclusion of this report is that while TileDB is readable, simplistic in its nature, it can be incredibly difficult to implement. I believe that I struggled most with the lack of knowledge and know-how to get this project set up initially within the correct time frame. With more time I believe I could have achieved this project's goals. I do however believe this project to be a success still, for my personal knowledge and understanding of the TileDB system has expanded immensely, and not just of TileDB but data, AWS, Linux, nodes, and terminology. If I had not have had such issue with the C++ implementation, I believe I could have finished on time.

In going over the limitations of TileDB, the opportunities seem almost endless with an incredibly fluid API in multiple languages and an array-to-file system that seems more efficient and easier to set up than all competing systems.

ACKNOWLEDGEMENT

The author of this paper is grateful for the support that Dr. Zhao contributed to this project. With pointing towards AWS and helping me get the necessary vocabulary to even start working on this project, it would not have been possible without him.

As a special note, my company, Sierra Nevada Corporation's team has asked me to present TileDB to them, which a large effort of my knowledge in this report going towards this goal of understanding the basics, which also hindered the results of this project. While it's possible the actual array implementation was not perfect to specs, it may help me in my professional career and to that I am grateful to taking this class.

REFERENCES

[1] The TileDB Array Data Storage Manager