Multinode-TileDB

A Distributed Array Database

Qian Long, Evangelos Taratoris, Giuseppe Zingales

December 11, 2014

1 Introduction

Array Databases are designed for large scale computation and storage of data that fit the array model. Users of array databases might want to perform complicated analytics such as machine learning algorithms or other statistics. One particular interesting and not well studied aspect of array databases is how to optimize them for sparse and skewed data. Such data is likely to occur naturally in scientific settings like measurements from sensors. *TileDB*, a system we summarize in the next section, is a single node database designed for exactly this kind of data. Of course, in this day and age, the data that people collect most likely does not fit into one machine. To solve this problem, we look to distributed systems.

In this project, we designed and implemented a first iteration of a distributed array database based on *TileDB* called *Multinode-TileDB*. We designed and implemented a communication framework between nodes in the distributed cluster. We added a few single node queries to the current *TileDB* implementation. We parallelize them along with some of the queries already implemented in *TileDB*. Finally we ran experiments on a cluster of machines to analyze the performance of our system.

2 Background

The *TileDB* system is based on the C-store DBMS. Most DBMSs are write-optimized, which means that all the attributes of a record is stored contiguously on disk. Those systems tend to be row-oriented. However, if we need a system that is read-optimized then a column-oriented approach is more efficient. C-store is a column-oriented DBMS that was released on 2006, and on which idea *TileDB* was buit.

TileDB is an array data management system (ADMS). ADMS systems primarily deal with data that are more naturally represented by arrays rather than relational tables. Some examples are data from space exploration programs and oceanographic research. These are all examples where the data are associated with coordinates in a multi-dimensional space, and need to be filtered, sampled, aggregated, and analyzed.

TileDB is a very complex system with many parts and modules. We highlight its array data model and internal storage because those are the most relevant for

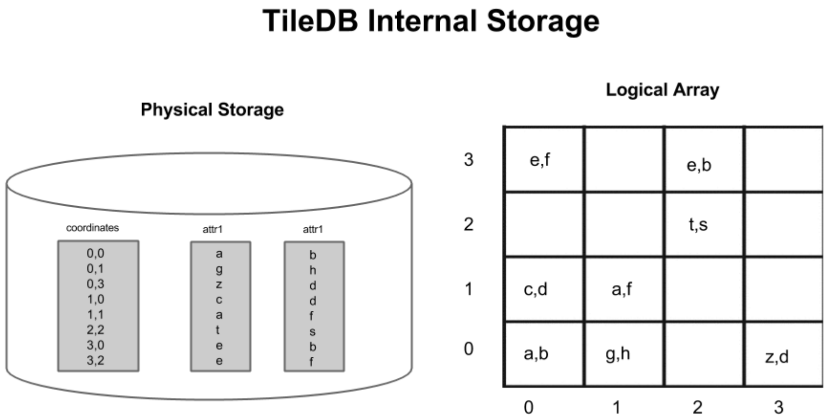

Figure 1: TileDB Internal Storage

understanding *Multinode-TileDB*. But TileDB also has other features including smart indices and other book keeping structures.

2.1 Array Data Model

An array consists of a set of dimensions (coordinates) and a set of attributes. Each dimension and each attribute may have an arbitrary data type. The current implementation of *TileDB* supports INT, FLOAT, DOUBLE, and INT64_T. A combination of coordinates and attributes defines a cell in the array. *TileDB* physically stores only non-empty cells. As with C-Store, *TileDB* vertically partitions the data on disk. It stores the coordinates in one file and stores each attribute in a separate file.

2.2 TileDB Internal Storage

TileDB vertically partitions the data on disk. It stores the coordinates in one file and stores each attribute in a separate file. This is similar to the strategy that C-Store uses to store data. Figure 1 illustrates how a logical array is transformed into the internal storage of *TileDB*. Typically the input data is a CSV file with the number of lines equal to the number of non-empty cells and the number of columns equal to the number of coordinate dimensions plus number of attributes.

3 Design

Multinode-TileDB is designed for analytic workloads meaning that it needs fast reads of the data but does not support single insertions. Following the design of *TileDB*, *Mulinode-TileDB* has a batch computation model, meaning that a new array is written to disk as a result of a query. We can explore an iterative or more pipelined model later.

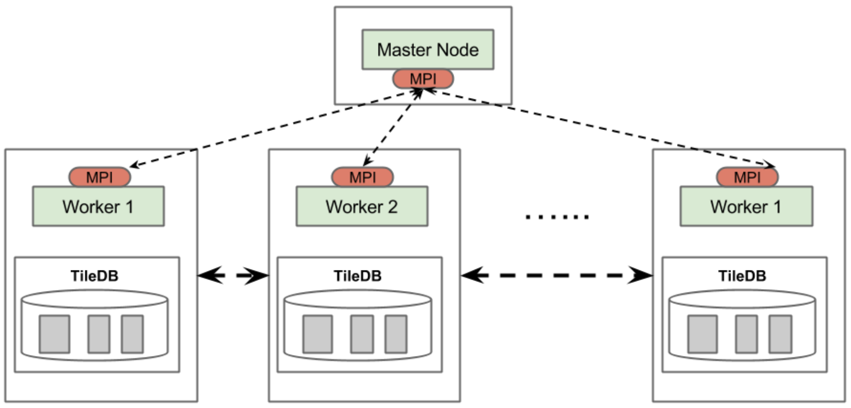

Figure 2: Distributed Architecture

3.1 Distributed Architecture

Multinode-TileDB has a share-nothing, centralized distributed architecture. There is one coordinator node and n worker nodes. As shown in Figure 2, each worker node runs a local deployment of *Multinode-TileDB*, which consists of *TileDB* bundled with our message passing framework implemented on top of MPI.

The coordinator listens for user requests and directs them to the workers. Every node is able to communicate with every other node but only the coordinator relays full results to the user. In this project, we focus on queries that do not require data shuffling.

3.2 Communication

Multinode-TileDB uses the MPI (Message Passing Interface) as its reliable communication layer. We designed a message passing framework on top of MPI for communication between nodes about database operations. We use the low level MPI feature of sending and receiving character buffers to build our message passing framework between nodes. We built serializers and deserialziers to transfrom high level DB operations into byte arrays that are sent across the network and interpreted by other receivers.

Internally, the communication between the coordinator and worker nodes is asynchronous. However to the user, experiences the queries as blocking. In fact, the user would not even be aware of the distributed architecture. For example, if the user requests an aggregate query on some array, s/he sends this request to the coordinator. The coordinator would send that filter message to all of the workers asynchronously so that the workers can do work in parallel. The coordinator blocks until all of the workers have replied, then the coordinator would send the final result to the user. To the user, the query is blocking, which is normal behavior in a single node database.

In this project, we only focus on the coordinator and workers part of the design because that's the most interesting and difficult. Adding the a user interface could easily be done by using our api. This also allows a Developer to embed *Multinode-TileDB* in a larger variety of locations.

4 Implementation

4.1 Queries

4.1.1 Single Node Methods

The current implementation of *TileDB* provides a complete storage manager that manages reading and writing to/from disk as well as indices and other bookkeeping structures. It also has a few queries, namely Load, export_to_CSV, and Subarray. We implemented two more basic single node queries to *TileDB*: Filter and Aggregate.

Filter As mentioned above, the internal data representation of *TileDB* is stored on-disk as a column store, with one file for the coordinates and one file for each of the attributes. A filter query takes in a predicate which consists of an operator, an operand, and the attribute to filter against and creates an output array with the matching cells. Since the data is stored by column, filtering can be implemented using minimal disk accesses. We simply have to scan the corresponding attribute file until we find a cell that matches the predicate while keeping track of the offset. When we have found a matching attribute, then we scan in the rest of the files (coordinate and the other attribute) to the same offset to retrieve the rest of the logical cell. As this process is running, we append the matching cell to the new output array.

Aggregate The algorithm for aggregate is very similar to that for filter. The aggregate query takes in an Aggregator type and an attribute to aggregate on. For aggregate, we scan the corresponding attribute file, apply the aggregator and output the result. This is even more I/O efficient than filter because we don't have to touch the other files, nor generate new tables.

Aggregate by itself is not too interesting without group by, which we unfortunately could not get to for this project. In the future, when we implement group by, we would be scanning the coordinate file along with the aggregate file to find the aggregate for each group, probably building a hash map along the way. If things do not fit into memory, we would have to spill over to disk.

4.1.2 Parallel Methods

We implemented parallelized versions of the the above methods for *Multinode-TileDB*.

Load In a parallel load, we assume that each of the worker nodes already has its partition of the overall global array as a csv file, and that these partitions are roughly equal. When a load operation comes in, the coordinator sends a load message to each worker node. This message contains two important arguments: The local path of the CSV file and the ArraySchema which describes how information is stored in the CSV file (e.g. number of dimensions, number of attributes, types, etc).

Each worker performs a local load on its data partition and sends back to the coordinator an ack once it is done or errors. As a brief summary, in a local load, the worker's *TileDB* takes in an input csv file, sorts it according to a desired sort order, then creates m + 1 binary files, for the m attributes and 1 more for the coordinates. This was already implemented in the original *TileDB*.

Filter In parallel filter, the coordinator sends each worker node a Filter Message, which contains the ArraySchema of the table to filter, the name where to store the newly filtered array, and the predicate on which it should filter. Each worker responds with an ack signaling completion or error.

Aggregate Parallel aggregate is slightly different from the load and filter methods above because the coordinator also needs to do some work to obtain the final result for the user.

As with the previous two methods, an Aggregate Message is sent to each worker, with an array name, the aggregator type, and the target attribute column. Each worker node computes a partial aggregate on its local data and send this information back to the coordinator. The coordinator then metabolizes the partial aggregates from each worker into one final result that is returned to the user.

Get The get method is a basic way to retrieve the entire global array from the system in a human readable format. In get, the coordinator sends a Get Message to each of the worker nodes, which simply consists of the arrayname to get.

After each worker node receives the Get Message, it executes export_to_CSV locally to create a CSV of the data. Then it streams the csv file back to the coordinator node. We also handle the case when the csv does not fit into memory because the worker streams chunks of CSV at a time back to the coordinator.

We decided to include this method instead of bundling it with methods such as filter because the user normally probably would not be interested in viewing gigabytes of data. Often times, the user may want to execute several queries on the database before seeing the final result, and thus may want a separate method to actually view the data.

4.2 Known Issues

CSVs Larger than Memory An issue that we discovered only well into the gathering data stage is that there was a bug in the underlying code, preventing *TileDB* from loading a dataset that was larger than memory. We suspect the error is by incorrect or inefficiently flushing to disc during the initial sorting phase of the load operation.

Subarray We initially planned on testing subarray but some issues occurred when we tested on larger datasets. We could not debug this in time, so we took it out of the experiments. This isn't particularly detrimental because filter serves a very similar (yet slightly less efficient) version of subarray.

4.3 Tools

Multinode-TileDB is implemented in C/C++. We added around 3000 lines of code on top of the original *TileDB*.

As mentioned above, we use MPI as our communication layer. MPI allows for reliable communication between nodes and is portable enough to run on multiple platforms. We use a variant of MPI called MPICH2 (v3.0.4) that is one of the more popular implementations of MPI.

We set up an MPI cluster on the csail OpenStack for deploying and testing our system.

5 Testing and Results

We tested three core methods in *Multinode-TileDB* and compared their scale up and speed up as the number of nodes increased and the overall array size increased. The methods we tested were: Load, Filter, Aggregate.

5.1 Test Setup

We setup our testing environment on an 9 node OpenStack Cluster. Each of the machines had 1GB of RAM, 16GB of HDD, and a single 2.4GHZ processor. The machines were configured such that one machine was always the coordinator and the rest of the machines were the workers. We varied the number of nodes in the cluster for our experiments.

We generated 3 distinct dataset for testing, totaling 500MB, 1GB, and 2GB, respectively. Each dataset was partitioned and distributed equally across all of the nodes involved for a particular test. Each (dataset, number of nodes) pair marks a distinct test where the total data (dataset) was partitioned across that particular number of nodes. We ran 3 trials for each test such that we could try to average out the noise.

Data Generation We generated all the test data randomly and simultaneously on all nodes while also guaranteeing that each node had the correct partition of the dataset for each test. We were able to achieve this by running a simple python script on all machines, seeding the PRNG to a pre-determined value, and using round robin distribution. This is equivalent to and much faster than generating the dataset on one machine and manually sending partitions over the network.

Furthermore, we also sped up data generation of the bigger datasets by copying the smaller datasets to form larger ones. We realize that this reduces variation in the data but it is still uniformly distributed, so it should not affect our results.

5.2 Results

5.3 Analysis

1. Load: The load function would load a file of size $\frac{DataSetSize}{NumNodes}$ MB from disk into memory. This data was with the 2 cordinates and 2 attributes varying in the set of [0, 1,000,000]

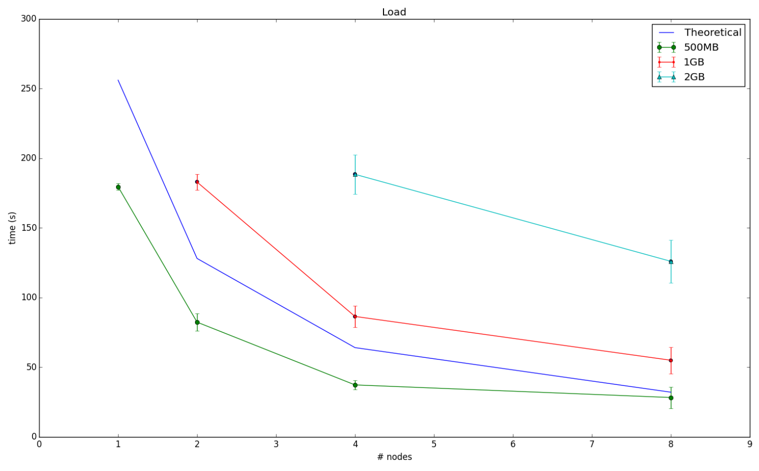

Figure 3: Load

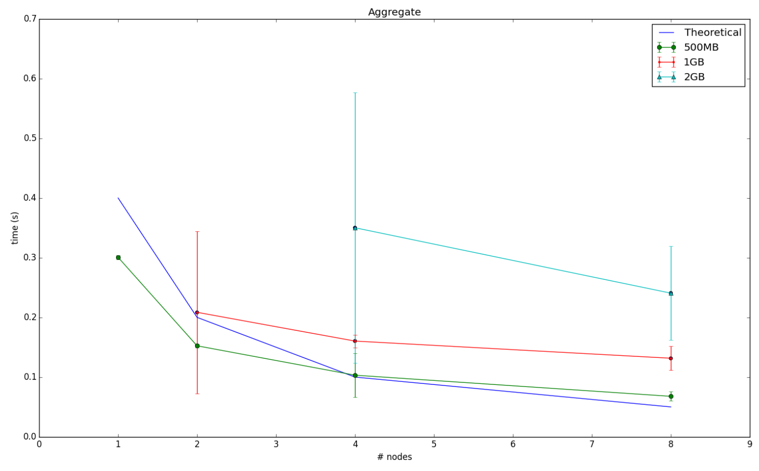

Figure 4: Aggregate

2. Filter: Was run with a predicate looking for values of attribute 1 that were greater than 500,000. Leading to a selectivity of roughly .5
3. Aggregate: Was run looking for the max over attribute 1.

5.3.1 Speed Up

Aggregate shows only a bit slower than linear speed-up this is often the case in functions that run very close to zero time. Two things make achieving linear speed-up very difficult. 1) That noise makes up a bigger percentage of the total run time, since noise tends to be very loosely related to data size. This can be seen by how the deviations get substantially larger compared to the total runtime. Secondly there is a minimal overhead associated with any operation, and especially so on one that runs over a network. This raises the minimal runtime dramatically.

Since the aggregate we ran was max over a single attribute, which does very little other than sequentially scanning the data, we can give a lower bound on how fast sequentially scanning through an attribute is, looks to be around 1GB per second, we weighed it more towards the larger datasets since they would be less affected by the constant time due to MPI.

Unfortunately because of the bug in the underlying TileDB code we were not able to test data that was larger than memory. Meaning we do not have a good estimate of how long it would take our system to sequentially scan over an attribute that was not already in memory.

Filter Filter looks like it performs quite well on average. However, there is a large variance in the data, particularly on the 1GB Dataset with 4 nodes. We designed the datasets to have uniform distribution so the work should be equal on each worker. Thus, the variance in the data is probably best explained by the variance in the machines that we are testing on.

Load Load shows less linear speed up then the previous two queries. This is probably due to the straggler effect, once again. Since the load operation requires significant computation and I/O (load produces several temporary data files on disk), the effects of any slow machine would be greatly exaggerated. Another interesting thing to note is that load has less variance than the previous two queries, probably also due to the complexity of the operation. Load is already very slow compared to the other two operations, so it is more resilient to noise in the system.

5.3.2 Scale Up

Looking at the cross section of the graphs we see that the our system scales up rather well. (ideal would be a perfect horizontal line). For load the time it took 1 node to processing 500MB $\approx$ 2 nodes processing 1GB $\approx$ 4 nodes processing 2GB. The scale up for filter and aggregate are a bit harder to determine, due the variance.

In figure 4, ignoring the 1GB dataset, if we look at (500MB, 2) and (2GB, 8) we see a 25% cost in scaling up. Our leading hypothesis is that to this cost is the straggler effect, that as the there are more and the operations gets faster, one node being slow dramatically reduces the total runtime, the huge variance in the of data point (1GB, 4) would support this hypothesis.

6 Conclusion

Our current project is only a first iteration of *Multinode-TileDB*. One of our main contributions is the communication infrastructure we have implemented on top of *TileDB*. This infrastructure can be used as building blocks for more complicated queries in the future. We have also demonstrated that the "very parallelizable" queries did indeed perform well as compared in terms of speed up and scale up.

6.1 Future Work

There is still much work left in designing and implementing more complicated queries like join and group by or even more ambitious operations like matrix multiple or k-nearest neighbor, namely queries that will require data shuffling among nodes. Also, there is more work to be done on the distributed side such as reliability and availability.

Complex Queries To implement more complex queries, we will probably want a globally sorted data with each worker holding a partition. The coordinator will then need to keep track of the array schemas of each worker and stats on their data. This way, the coordinator can make intelligent decisions of where to direct user requests because they might only be applicable to a few nodes. For example, if the array is globally sorted, then the subarray query (filtering on coordinates) would only have to go to the nodes that have overlapping data. We will also need to extend our current message passing framework to allow for data shuffling.

Availability A simple fix that would greatly improve performance is to have replication of worker nodes.

Since this DBMS is optimized for reads, we do not need to be as worried about data consistency among the replicas. Having replication for the worker nodes would greatly help against the straggler effect since each operation is only as fast as its slowest link. Redundancy would also help with non-data related errors and crashes. In the current implementation of *Multinode-TileDB*, workers are doing extra work when one of the siblings fail because it doesn't know that the operation has crashed.

A substantially more complicated idea for improving reliability is having a more decentralized architecture. Right now, the coordinator is a single point of failure and a bottleneck in terms of data processing. Instead of having one dedicated node for the coordinator, we can install a coordinator on each of the worker nodes. A user could send a request to any machine in the cluster and that machine would act as the coordinator node for that query.

Yet another idea is to have layered communication. Having all the worker nodes send its information to one coordinator node could easily overwhelm that node, especially if the operation was very network intensive(like get). In a layered approach, each node would send its information to a supervisor (another worker). The supervisor would collect data from multiple workers (depending on the branching factor) and then send this information to an even higher up supervisor, eventually the messages would reach the coordinator. One concrete example could be aggregate. Instead of having the coordinator receive partial aggregations from each worker, the supervisors would be the ones performing the partial aggregation, which reduces the final aggregating for the coordinator .While this increases the total amount of information that is send through the network it is more load balanced for everyone

References

[1] Stonebraker, M., Brown, P., Poliakov, A., Raman, S. The Architecture of SciDB. *In Proc of Scientific and Statistical Database Management - 23rd International Conference (SSDBM 2011)*, 2011.

[2] Stonebraker, M., Abadi, D., Batkin, A., Chen, Xuedong., etc, C-Store: A Column Oriented DBMS. *In Proc of VLDB - 23rd International Conference (VLDB 2005)*, 2005.

[3] Papadopoulos, Stavros., etc, Array Storage Management Revisited. *Work in Progress*