Parallel Load and Query Processing in a Distributed Array Database

by

Qian Long

B.S., Massachusetts Institute of Technology (2014)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

© Massachusetts Institute of Technology 2015. All rights reserved.

Author .................................................................

Department of Electrical Engineering and Computer Science

May 22, 2015

Certified by ............................................................

Samuel R. Madden

Professor

Thesis Supervisor

Accepted by ............................................................

Albert R. Meyer

Chairman, Masters of Engineering Thesis Committee
Parallel Load and Query Processing in a Distributed Array Database

by

Qian Long

Submitted to the Department of Electrical Engineering and Computer Science on May 22, 2015, in partial fulfillment of the requirements for the degree of Masters of Engineering in Computer Science and Engineering

Abstract

Scientists across many research domains collect large amounts of multi-dimensional data in their day to day work. They require high performance, scalable systems to manage and process their data. Oftentimes, the underlying distribution of these types of data is skewed and sparse, rather than dense and uniform. As input data sizes continue to grow at a rapid rate, main memory and storage capacity become bottlenecks on single machines. Thus, we look to distributed array databases as a long term solution for managing and querying this type of data.

This thesis presents Multinode-TileDB, a distributed framework that extends TileDB, a new array database management system designed, from the ground up, to handle skewed and sparse arrays. We design the overall distributed architecture and propose and implement parallel algorithms for load, join, subarray, and filter while focusing on load balance and performance. Our experiments show speedup gains as cluster size increases and how different data partitioning schemes benefit the different parallel queries.

Thesis Supervisor: Samuel R. Madden
Title: Professor
Acknowledgments

I am lucky enough to have met so many wonderful people during my past five years at MIT who have influenced me and helped me get to where I am today. Some of them deserve special mention here.

First of all, I would like to thank Prof. Mike Stonebraker and Prof. Sam Madden for giving me the opportunity to work on this fascinating project. Thank you both for your support and guidance.

I would like to thank my mentor Stavros Papadopoulos for all of his advice, guidance, and encouragement every step of the way. Your ideas made this thesis possible. I could not have done this without your help.

I would like to thank Evangelos Taratoris and Giuseppe Zingales for their technical contributions to the early stages of this project. I would also like to thank Nikos Armenatzoglou for his help in gathering and processing data for the experiments. I would also like to thank Dorothy Curtis for helping me with setting up a testing environment on the machines.

I would like to thank Carrie Liang for always being there for me. Thank you for your irreplaceable friendship. This paragraph is for you.

I would like to thank Deborah and Sarah for encouraging me to join one of the best classes at MIT, which has kept me energized and focused these past few weeks. I hope that we will run together again someday. Also, thanks, Sarah, for making me aware of important deadlines.

I would like to thank Bryan, Catherine, Emily, Kelly, Michelle, Pratiksha, Rui, Sharon, Stephen, and Wenting for your friendship and support throughout my time here. I will always cherish our wonderful memories together.

Lastly, I would like to thank my parents for their unconditional love and support throughout my life. Thank you for all of the sacrifices you have made in giving me the best possible future.

And finally, to the reader, keep going!
Contents

1 Introduction 13
   1.1 Motivation ................................................. 13
   1.2 Thesis Contribution and Organization ...................... 14

2 Background 15
   2.1 Array Data Model ............................................ 15
   2.2 TileDB Overview ............................................. 15
      2.2.1 Internal Storage Engine ............................... 16
      2.2.2 Bookkeeping Structures ............................... 18
   2.3 Related Work ................................................. 19
      2.3.1 Irregular vs Regular Tiling on Disk ................. 19
      2.3.2 Partitioning Data Across Nodes .................... 21

3 System Design 23
   3.1 Distributed Architecture .................................. 23
   3.2 Communication Module ..................................... 24
      3.2.1 MPI .................................................. 25
      3.2.2 Implementation Details ............................... 26
   3.3 Computation Model .......................................... 26
   3.4 Data Partitioning Schemes ................................ 27
      3.4.1 Hash Partitioning ..................................... 27
      3.4.2 Ordered Partitioning ................................. 27
List of Figures

2-1 TileDB Internal Storage on Disk ........................................ 16
2-2 Global sort order examples ............................................. 17
2-3 Regular and Irregular Tiling on Uniform Array ...................... 19
2-4 Regular and Irregular Tiling on Skewed Array ........................ 20
3-1 Distributed Architecture .................................................. 24
4-1 Data Shuffling for \textit{Joins} on Ordered Partition .................... 38
5-1 AIS Dataset Sample from Jan 2011 ................................. 42
5-2 Load Execution Breakdown (18GB dataset) ........................….. 44
5-3 Load Speedup and Scaleup Performance ............................. 47
5-4 Join Execution Breakdown on Worker Nodes (50GB x 53GB) ...... 48
5-5 Join Performance ......................................................... 50
5-6 Subarray Execution Breakdown on Worker Nodes (50GB) ........ 52
5-7 Subarray Performance ..................................................... 53
List of Tables

5.1 Default Experiment Parameters .................................. 42
5.2 Coordinate and Attribute types for AIS Dataset .............. 43
5.3 Load breakdown details for each type .......................... 45

B.1 Number of Samples for Confidence $1 - \delta = .99$ and Error Margin $\epsilon = .02$ 61
Chapter 1

Introduction

Array database management systems are complex software applications designed to manage multi-dimensional data. Two important things that array databases provide are storage management and primitive operators for typical computations on array data. Specific operators include traditional database queries as well as order statistics and machine learning algorithms.

The explosive growth of data, particularly in scientific applications, enabled by growing accuracy and lowering cost of measurement devices, requires array database systems that can efficiently manage, process, and perform queries on large amounts of data.

1.1 Motivation

As mentioned above, scientists across many research disciplines typically use multi-dimensional arrays to represent their data. Examples such as geospatial data, temporal data, oceanographic data, telematics data, and genomics data are naturally modeled as arrays [9]. Oftentimes, the underlying distribution of these types of data are not uniform, but rather, skewed and sparse. Intuitively, in geographic contexts, there tends to be data concentrated in more populous regions. Furthermore, as the size of input data continues to grow at a rapid rate, main memory and storage capacity become bottlenecks on single machines. Thus, we look to distributed databases
as a long term solution to this problem.

Current commercial array database systems were not originally designed to handle data containing skew and sparsity. They were initially designed assuming dense uniform arrays and have only recently began considering data skew [9].

In contrast to existing systems, TileDB [3] (summarized in Section 2), is a new array database specifically designed, from the ground up, to optimize performance on sparse and skewed arrays.

In this thesis, we present Multinode-TileDB, a distributed framework extending TileDB to multiple nodes, with a focus on load balance and performance of database queries on sparse and skewed arrays.

1.2 Thesis Contribution and Organization

This thesis contributes a design and implementation of Multinode-TileDB, a distributed array database built on TileDB [3]. We present a communication module and parallel algorithms and implementations of several traditional database queries.

Specifically, we present two schemes for partitioning data across nodes: hash partitioning and ordered partitioning (Section 3.4). We propose and implement algorithms for parallel queries on these two partitioning schemes, including parallel load, join, subarray and filter. Our design differs from previous distributed array databases in that we partition data across nodes at a finer granularity to maximize for load balance among nodes. Furthermore, our parallel algorithms directly build off of design choices from TileDB for handling sparse and skewed arrays, especially with regards to the storage engine. Finally, we provide an experimental analysis of Multinode-TileDB’s performance.

The rest of the thesis is organized as follows. Chapter 2 discusses the background and related work. Chapter 3 discusses the overall system design. Chapter 4 describes the algorithms and implementations of parallel queries. Chapter 5 presents experiments and discusses results. Chapter 6 concludes and discusses future work.
Chapter 2

Background

This chapter presents background on the array data model, details about the current single node version of TileDB, and previous literature pertaining to array databases.

2.1 Array Data Model

The data model that we work with is called the array data model. An array consists of a set of coordinates (also referred to as dimensions) and a set of attributes. Each coordinate and each attribute may have an arbitrary data type. Each coordinate also has a corresponding domain, which marks array boundaries. A combination of coordinates with its corresponding attributes defines a logical cell in the array. A logical cell in an array database is comparable to a row in a relational database.

2.2 TileDB Overview

As mentioned previously, TileDB [3] is an array database designed for sparse and skewed arrays, currently implemented for a single node. At a high level, TileDB optimizes for sparse and skewed arrays by organizing cells into fixed physical size tiles (same number of bytes). Each tile, thus, covers variable and possibly overlapping extents in the logical space. This is in contrast to fixed logical size tiles, which cover fixed size logical extents of the array but maybe vary in physical size (see Section
2.3.1) A huge benefit of fixed physical size *tiles* is that they allow for lightweight bookkeeping structures, as *tiles* are simply offsets within the data files. The minimal overhead is one of the reasons that *TileDB* is *skew agnostic* in terms of performance.

Next, we highlight the internal storage engine and bookkeeping structures as they are most relevant for the design of *Multinode-TileDB*.

### 2.2.1 Internal Storage Engine

*TileDB* physically stores only non-empty cells on disk. It vertically partitions the data on disk, meaning that the *coordinates* are stored in one file and each *attribute* is stored in a separate file, as shown in Figure 2-1. Thus, an array with *d coordinates* and *m attributes* will result in *m + 1* data files on disk.

For example, Figure 2-1 shows a logical array on the left. The example array has 2 *dimensions* and 2 *attributes*. The domains of both dimensions are [0,4]. The empty cells are blank. The right side of the figure shows the corresponding physical storage on disk. The non-empty *coordinates* are stored in one file and their corresponding *attributes* are each stored in a separate file. There is a one-to-one correspondence between the offsets in the *coordinates* files and each of the *attribute* files. In other
words, the combination of coordinates and attributes with the same offsets from their respective files constitute a logical cell.

The idea of vertically partitioning an array is similar to C-Store’s idea of vertically partitioning a relational database table [10], where each column of a relational table is stored in a separate file on disk. In general, column oriented storage (as opposed to row oriented storage) is more efficient for analytics on a subset of attributes because it avoids reading irrelevant data from disk.

We mentioned in the overview that TileDB organizes cells into fixed physical size tiles. Specifically, we can view tiles as contiguous fixed size data regions starting from some offset from the beginning of either the coordinates or attribute data files. We use the terms coordinate tiles and attribute tiles to distinguish the two.

Furthermore, TileDB also imposes a global ordering, also known as a sort order, on the array, which is provided by the user upon loading (Section 4.1). Cells are packed into fixed physical sized tiles based on the global ordering. This effectively forms a clustered index that can greatly improve query performance. Currently supported sort orders include row major, column major, and hilbert order. The sort order of an array determines how cell_ids are computed. Cell_ids are used in loading an array (Section 4.1) and used to establish the precedence relationship between two cells. Cell_ids are not explicitly stored after loading since they are easily computed from closed formulas.

Figure 2-2 illustrates the three mentioned sort orders. The arrow in each array
begin on the cell with the smallest cell_id, based on that particular sort order, and points towards increasing cell_ids.

2.2.2 Bookkeeping Structures

In addition to the actual data files, TileDB also stores the following bookkeeping structures on disk to facilitate query processing. The main ones include:

offsets.bkp

This bookkeeping structure stores tile locations as byte offsets into the coordinates data file, mentioned in the above section. These tile offsets allow us to easily seek to actual contents in a data file when reading the data of a tile. TileDB also reads several tiles from disk directly following a requested tile in order to amortize seek time.

mbrs.bkp

This bookkeeping structure stores the minimum bounding rectangle (MBR) of each tile. MBRs constitute the tightest hyper-rectangle in the logical space containing all of the cells in a tile. This file is used to facilitate subarrays (Section 4.2).

For row major and column major sort orders, these minimum bounding rectangles are skinny rectangles in 2-dimensional space. For hilbert order, these are more square shaped. We can see this from Figure 2-2, where the arrows trace out the order in which cells are packed into tiles. For simplicity and clarity, if we assume the figure contains all non-empty cells with a tile size of 4 cells, the tiles formed from row major order are shaped as 4 x 1 rectangles. The ones formed from column major order are shaped as 1 x 4 rectangles, and the tiles formed from hilbert order are shaped as 2 x 2 squares.

bounding_coordinates.bkp

This bookkeeping structure stores the first and last coordinates of each tile,
which are also referred to as *bounding coordinates*. This is used to facilitate *joins* (Section 4.3).

### 2.3 Related Work

Array databases have been well studied in recent years. In this section, we look at some different techniques in storage management and data partitioning from past literature.

#### 2.3.1 Irregular vs Regular Tiling on Disk

As mentioned in Section 2.2.1, *TileDB* organizes cells into fixed physical size *tiles*. This is also known as *irregular tiling* in the literature. Another technique, on the opposite end of the spectrum, is organizing cells into fixed logical extents, also known as *regular tiling* in the literature [9]. *SciDB* is an existing array database management system that uses *regular tiles* in its storage engine [9].

For a uniform array, *irregular tiling* and *regular tiling* result in *tiles* that are perfectly partitioned in both physical and logical space, as illustrated in Figure 2-3. In this figure, we use *hilbert* order for the *irregular tiling* and we use 4 by 4 logical extents for *regular tiling*. *Tiles* resulting from both methods have the same content,
Figure 2-4: Regular and Irregular Tiling on Skewed Array

as illustrated.

However, for a skewed array, irregular tiles cover a variable logical space while regular tiles cover a variable physical size, as illustrated in Figure 2-4. This figure shows regular tiling also using 4 by 4 logical extents and irregular tiling using row major order. We see that the regular tiles all have a different number of cells and the irregular tiles do not all cover the same size logical extents, which are bold-ed in the array on the left.

Performance under regular tiling can be quite sensitive to skewed arrays because it is hard to fine tune for a good logical extent size. Specifically, large extents correspond to a coarser tile granularity, which results in a small number of very large tiles. Queries that only need to retrieve a small part of a tile will incur wasted I/Os when reading unnecessary cells from disk. Conversely, small tile extents correspond to a finer granularity that require more bookkeeping overhead.

ArrayStore studies a multi-layer tiling approach, first organizing cells under one tiling strategy and then grouping those tiles together under a second tiling [8]. They conclude that using regular tiling for both layers led to the best performance. However, the need to fine tune for optimal tile extents for first tier regular tiles still remains.

TileDB addresses these issues with by using a one layer irregular tiling approach (Section 2.2.1) which is easy to tune and agnostic to skew.
2.3.2 Partitioning Data Across Nodes

In addition to tiling techniques in local storage, we are even more concerned with how to partition data across nodes in Multinode-TileDB’s distributed environment.

Both SciDB and ArrayStore’s multi-node implementation use a shared-nothing architecture. That is, each node in the cluster has its own local storage that is not shared with any other node. SciDB partitions data across nodes on a tile granularity, meaning that they first divide the array into tile and then map each tile to a node, with tiles being some variation of the multi-layered tiling strategy discussed above [9, 8, 4]. ArrayStore supports varies mappings from chunk to node, including round robin, random, range, and block cyclic [8].

SciDB and ArrayStore also have a notion of overlapping tiles where each node contains duplicate data from neighboring nodes to minimize data communication during queries [8, 9, 4]. This is beneficial for queries that require data from nearby nodes to be co-located on the same node (i.e., join). However, the overlapping size is yet another parameter that must be tuned carefully for good performance. If the overlap is too small, data has to be communicated anyways. If the overlap is too big, extra space is taken up. Furthermore, keeping track over overlapping tiles also adds additional management overhead to the entire system. We currently do not support overlapping tiles across nodes but may consider it in the future if data communication becomes a bottleneck.

In Multinode-TileDB, one of the things we focus on is load balance. Rather than assigning data to nodes at a tile level granularity, we assign data to nodes at a cell level granularity. We apply two classic partitioning schemes (also used by SciDB and ArrayStore) in Multinode-TileDB: hash and ordered partitioning (described in Section 3.4). Also, because we do not have duplicated tile overlaps between nodes, we do communicate data when processing queries that are not embarrassingly parallel. Though we try to minimize this communication overhead by using a cost model to decide how to transfer data when necessary (see Section 4.3 for join).
Chapter 3

System Design

This chapter presents the high level system design of Multinode-TileDB. We discuss the distributed architecture, the communication module, the computation model, and distributed data partitioning schemes.

3.1 Distributed Architecture

Multinode-TileDB has a shared-nothing, centralized architecture, consisting of one coordinator node and n worker nodes. Each node has its own local storage that is isolated from other nodes, and nodes communicate with each other via message passing. As shown in Figure 3-1, each worker node runs a local instance of Multinode-TileDB, which consists of TileDB’s current single node storage engine bundled with our communication module. Array data are stored on the worker nodes; the coordinator node does not contain any data. Instead, the coordinator is responsible for communicating with the client (not shown in the Figure 3-1).

In a typical workflow, a client issues a query to the coordinator. The coordinator may do some initial processing depending on the query, and then issue the message to all the workers to continue with the query. Workers may communicate data with each other (if necessary) and execute local operations to complete the query. Once the workers finish, they send acknowledgements back to the coordinator, who then may do additional post processing work. The final response is then relayed back to
Every node is able to communicate with every other node in the cluster but only the coordinator communicates with the client.

In this thesis, we only focus on the backend aspect of this system, the interface with a client is an orthogonal issue.

### 3.2 Communication Module

The communication module of Multinode-TileDB includes a message passing framework built on top of a reliable communication layer. We present an API that abstracts away low level communication details, so that nodes are only aware of sending pre-defined messages to each other. The module currently features a set of `send` and `receive` functions for communication.

The module also includes a framework for defining high level messages to be communicated among the nodes. Such messages include directives that the coordinator sends to the workers to perform requested queries as well as data messages that nodes use to communicate array data and metadata with each other.

Internally, the communication between the coordinator and worker nodes is synchronous. Upon receiving user requested queries, the coordinator dispatches a query
message to the workers and then waits for all the workers (i.e., the slowest worker) to complete the operation before moving on to the next user request. There are also synchronization points within each parallel query implementation. Details of the parallel queries are discussed in Chapter 4.

Furthermore, the communication module also provides interfaces for two main types of communication: point-to-point, and all-to-all. Multinode-TileDB uses point-to-point communication for the coordinator to send directives to the workers and for the workers to send acknowledgements back to the coordinator. Multinode-TileDB uses all-to-all communication for shuffling array data among the worker nodes.

3.2.1 MPI

We use an existing implementation of the standard Message Passing Interface (MPI) as our reliable communication layer, which internally uses SSH and TCP for communication. We only use a few MPI primitives to build the communication module [11]. They are

- **MPI_Send**
  This method sends data to the desired receiver. The receiver must be calling MPI_Recv in order to receive the data. This is a synchronous method that will block until data is received.

- **MPI_Recv**
  This method receives data from the specified sender. The sender must be calling MPI_Send at the same time. This is a synchronous method that will block until data is sent.

- **MPI_Alltoall**
  This method performs an all-to-all data communication in which every node sends and receives the same fixed, amount of data from every other node. This method is also synchronous that blocks until every node calls it.
• MPI_Alltoallv

This method is the exact same as the above method except that it allows for sending and receiving variable length data between nodes.

The communication module also provides a framework for serializing and deserializing high level messages into byte arrays to be sent across the network via MPI.

3.2.2 Implementation Details

One of the limitations in the above MPI primitives is that senders must specifically tell receivers how much data (or up to certain amount) is coming. We cannot expect that user input datasets and our internal data will fit in main memory. To get around this, we add special messages into the communication module specifically for notifying receivers whether or not more data is coming. We build abstractions around this so that we only have to call one or two methods from the module for sending and receiving large amounts of data. For example, when sending an entire file in load (see Section 4.1.1), internally, the communication module basically sends one chunk of the file and then sends a message to the receiver indicating that more is to come, then sends the next file chunk and the next message, until all of the file is sent.

3.3 Computation Model

Multinode-TileDB has a batch computation model directly extending from TileDB. In particular, this means that a new array is always written back to disk as a result of a query and we never overwrite existing arrays. Complex queries involving multiple primitive operators would be done in a pipelined fashion in which subsequent operators act on the completed results of the previous operator. Optimizations for complex queries is an area for future work.
3.4 Data Partitioning Schemes

In this distributed system, a main issue to address is how an array should be partitioned across the nodes such that both load balance and performance is achieved. We study two main types of partitioning: hash partitioning and ordered partitioning. Both have their advantages and disadvantages, and ultimately, different queries will favor different partitioning schemes. We discuss how the different queries are implemented on both partitions in Chapter 4 and their performance results in Chapter 5.

3.4.1 Hash Partitioning

In hash partitioning, we apply a hash function to map each logical cell to a particular node. Assuming a hash function with uniform random properties, this approach will guarantee perfect load balancing on the nodes. However, this also destroys spatial locality because each node will essentially contain a random subset of the array. Note that each worker still maintains an order on its partition of the array. The random partitioning, actually does not greatly affect the database queries that we study later. But it may affect machine learning algorithms (i.e., nearest neighbor and clustering) and linear algebra algorithms (i.e., matrix multiplication) because they often need to act on contiguous blocks of the array. Implementing these more sophisticated queries is another area for future work.

3.4.2 Ordered Partitioning

In ordered partitioning, we impose a total global order (Section 2.2.1) on cells across all the nodes (i.e., worker 1 has the cells with the smallest cell_ids in the total order and worker n has the largest cell_ids in the total order). To be precise, ordered partition guarantees the following property.

Let there be n workers. Let ni be the number of cells in worker i. Let sj^i_j be the cell_id of the jth cell of worker i. Ordered partitioning guarantees that sj^i_j = sj^i_{j-1} and s_k^i ≤ s_j^i for all 1 ≤ k ≤ ni and 1 ≤ j ≤ ni and 1 ≤ l < i ≤ n.
A naive solution is to sort the entire array according to the *global order* in a centralized location and distribute the sorted cells evenly to all the nodes. This solution does not suffice because centralized sorting of an entire array is prohibitively expensive. We present a more sophisticated algorithm in the next chapter that trades off some load balancing for performance.
Chapter 4

Database Queries

This chapter presents details about the parallel database queries *Multinode-TileDB* currently supports. They include *load*, *subarray*, *filter*, and *join*, and are implemented on both types of data partitioning (*hash* and *ordered*). A large portion of this chapter is devoted to *load* and *join* since they require data communication among nodes. The other queries, *subarray* and *filter*, are *embarrassingly parallel* (meaning no need for data communication) and essentially come for free in *Multinode-TileDB* in terms of implementation.

4.1 Load

*Load* is the most fundamental operator in a database. A *load* operator takes in user input data, normally in the form of CSV files, and transforms the data into an internal storage format (outlined in Section 2.2.1). Specifically, each line of the CSV file corresponds to a logical cell of the array, which is formatted as follows

\[ c_1, \ldots, c_d, a_1, \ldots, a_m \]

where \( c_1, \ldots, c_d \) correspond to the \( d \) *coordinates* and \( a_1, \ldots, a_m \) correspond to the \( m \) *attributes* of the array. Future queries on a loaded array act on the data created by our internal storage engine.
In Multinode-TileDB, we support two types of load, depending on how the user provides input data. One type of load is a centralized load where the input data is supplied as one giant CSV file to the coordinator node. The other type is a distributed load, where the input data is already pre-partitioned across the worker nodes from some previous process. We make no assumptions about how they are pre-partitioned so we treat the initial inputs as a randomly scattered subsets of the global input. One example of how this type of distributed input can arise is from Hadoop, where the output of a MapReduce job is often composed of multiple files scattered throughout a distributed file system [1].

In all of the loading schemes described below, the user first provides an array schema to Multinode-TileDB that informs our system of the structure of the forthcoming array. This is also known as defining the array. The array schema contains meta information about the array, such as the number of dimensions, the type and domain of each coordinate, the total number and type of each attributes, and the sort order (described in Section 2.2.1). After the array is successfully defined, it is ready to be loaded.

4.1.1 Centralized Load

In centralized load, the user supplies one large CSV file to Multinode-TileDB. The coordinator node receives the input CSV, determines where each line should go (depending on whether hash or ordered partition) and sends it to the corresponding worker node. The functions used by the coordinator for mapping cells to nodes are described below. After the worker nodes receive their corresponding partitions of the initial input, they simultaneously invoke TileDB's local load (described in 4.1.3).

Hash Partition Centralized Load

In hash partitioning for centralized load, the coordinator applies the following function to each line of the CSV to determine its placement. Let \( H \) denote a uniform hash function. Let \( c_i \) denote the \( i \)th dimension of the coordinates. Let \( n \) denote the number
of worker nodes in the cluster. The assigned worker \( w \), where \( 1 \leq w \leq n \), of a particular cell is

\[
w = H(c_1, \ldots, c_d) \mod n \tag{4.1}
\]

This type of partitioning naturally load balances across worker nodes provided that hash function \( H \) produces random, uniform values.

**Ordered Partition Centralized Load**

In ordered partitioning, the goal is to divide the dataset into \( n \) partitions of roughly equal physical size, one for each of the \( n \) workers, while respecting the total global order of the cell ids (Section 3.4.2). In other words, we want to find \( n - 1 \) boundary points that evenly divide the sorted input dataset. Once we have determined the boundary points, we use the following equation to assign cells to workers.

Let \( b_i \) represent the \( i^{th} \) boundary point where \( 1 \leq i \leq n - 1 \). Let \( s_j \) denote the cell_id of the \( j^{th} \) element of the original input dataset of size \( N \), where \( 0 \leq j < N \). A cell is assigned to worker \( w \), where \( 1 \leq w \leq n \), using the following equation.

\[
w = \begin{cases} 
1 & \text{if } s_j \leq b_1 \\
 i & \text{if } b_{i-1} < s_j \leq b_i \text{ for } 2 \leq i \leq n - 1 \\
n & \text{if } s_j > b_{n-1}
\end{cases} \tag{4.2}
\]

The baseline approach, as mentioned in Section 3.4.2, is to have the coordinator sort the entire input CSV and take every \( \frac{N}{n} \)th element as a boundary point. However, the cost of sorting the entire dataset on a single machine is prohibitively expensive, thus rendering this approach infeasible.

Instead, we present an alternative approach to find boundary points via sampling. The idea is to estimate boundary points that roughly equally partition the dataset by obtaining a sample of the dataset, sorting the samples, and evenly dividing the sorted samples. In other words, we assign the \( n \)-quantiles of a sorted sample of a dataset as the \( n - 1 \) boundary points, where quantiles are points that mark the boundary
between consecutive partitions. This idea has been used before in the context of general parallel sorting [5], but we are applying it specifically to our parallel load algorithm.

Furthermore, we use reservoir sampling [12] as our sampling algorithm because it has the nice property of not needing to know the input data size in order to generate uniform random sample points. This algorithm also avoids random disk seeks.

By contrast, a naive sampling technique is to pick $K$ random positions in the input dataset of size $N$ and use the cells at those positions as our sample pool. However, this technique requires $K$ random disk seeks to read the cells, which we want to avoid. Instead we want to sample as we sequentially read the file. In a straightforward simple random sampling technique, we pick each point with probability $\frac{K}{N}$ where $K$ is the sample size and $N$ is the input dataset size. In a CSV file, $N$ is determined by the number of new line characters in the file, which can be computed only by scanning the entire file. Thus, using reservoir sampling saves us a full scan of the input dataset.

Putting together the pieces, we have the following simple algorithm for estimating load balanced boundary points. We first scan the entire input dataset and collect $K$ samples using reservoir sampling. The sampled points are cell_ids of each logical cell, which we compute on-the-fly as we scan. Then, we sort the $K$ samples and select the $n$-quantiles (every $\frac{K}{n}$th element) of the sorted list. These $n$-quantiles constitute the $n - 1$ boundary points. Finally we scan the input dataset again and assign cells to workers using Equation 4.2.

Now, the last question is determining how many samples we need such that the boundary points approximately load balances all $n$ partitions. To find the $n - 1$ boundary points with error margin $\epsilon$ and confidence $1 - \delta$, where $0 \leq \epsilon \leq 1$ and $0 \leq \delta \leq 1$, we need to sample at least $\frac{1}{2\epsilon^2} \cdot log\left(\frac{2(n-1)}{\delta}\right)$ points [7, 5]. The formal problem and proof are shown in Appendix A.

4.1.2 Distributed Load

As mentioned earlier, for a distributed load, we assume that each worker node initially starts with a random subset of the entire global array, which we refer to as a CSV
split. This type of load is very similar to the centralized versions. However, instead of a one-to-all data transfer from the coordinator to the $n$ workers, this type of load uses an all-to-all data shuffle, in which every worker simultaneously sends and receives data from every other worker. This is because every worker will initially start out with data that belonging to another node. Thus, at a high level, distributed load happens in two broad steps: 1) all-to-all data communication and 2) local load. We will describe each in more detail below for both hash and ordered partitioning.

**Hash Partition Distributed Load**

In distributed load for hash partitioning, every worker scans their input CSV split in parallel, applies the Equation 4.1 for assigning cells to workers, and sends data to the appropriate node. Once data shuffling finishes, we perform a local load.

**Ordered Partition Distributed Load**

This load is the distributed version of centralized load with ordered partitioning. This also has more overhead than its hash partition counterpart due to calculating partition boundary points in a centralized location.

We will present the algorithm and then analyze its correctness. The steps for this operation are as follows.

1. Every worker computes the size of their input split $N_i$ and sends it to the coordinator.

2. The coordinate receives size $N_i$ from each worker $i$. The coordinator computes $N = N_1 + \ldots + N_n$, the total size of the entire global array. The coordinator sends sample size $k_i = \frac{K}{N} \cdot N_i$ to each worker $i$ where $K$ is the total sample size of the global input. $k_i$ indicates the number of samples that worker $i$ should draw from its input split.

3. Each worker $i$, in parallel, scans their input CSV split, computes cell_ids, and samples $k_i$ points. Every worker sends their $k_i$ samples to the coordinator node.
4. The coordinator now has $K = k_1 + ... + k_n$ samples of the entire global array.
   It sorts those samples and computes the $n-1$ boundary points that divides the samples into equal quantiles. The coordinator sends the $n-1$ boundary points to all workers.

5. Every worker node, in parallel, scans their dataset again and determines which node each cell goes to based on the $n-1$ boundary points using Equation 4.2. The all-to-all data communication also happens during the scan, in a lock step manner. This means that after every worker node has read a fixed amount of their data into a buffer in memory, they will perform an all-to-all data shuffle to exchange data with other workers. Cell_ids are also sent along with the cell so that they do not need to be recomputed on the receiving node for local load.

6. Finally, each worker performs a local load on their, now, correct data.

We set the total sample size $K$ to $\frac{1}{2x^2} \cdot \log(\frac{2(n-1)}{\delta})$, the same bounds as the centralized version in Section 4.1.1. The difference between this algorithm and the centralized version is that the samples now come from multiple source splits of variable size. For correctness, the samples chosen in this way must be equivalent to the centralized version. That is, any cell in any of the $n$ splits must have probability $\frac{K}{N}$ of being sampled. The proof is outlined below.

Proof. Let $n$ be the number of workers. Let $N_i$ be the size of worker $i$’s initial split. Let $N$ be the total size of the input, so $N_1 + N_2 + ... + N_n = N$. Let $k_i$ be the number of samples taken from worker $i$. Let $K$ be the total number of samples taken from all $n$ workers, so $k_1 + ... + k_n = K$. We want to pick $k_i$ for each worker $i$ such that we ensure each cell in the global array (union of all splits) has probability $\frac{K}{N}$ of being chosen as part of the $K$ samples. In our algorithm described above, we pick $k_i = \frac{K}{N} \cdot N_i$. Thus, the probability of a cell chosen from worker $i$’s split is
\[ P(\text{choosing a cell from worker } i) = \frac{k_i}{N_i} = \frac{K}{N} \cdot N_i \cdot \frac{1}{N_i} = \frac{K}{N} \]

Our current implementation of this algorithm assumes initial splits of equal sizes, so \( N_i = \frac{N}{n} \). Thus, we are currently omitting the first 2 steps outlined above and setting every \( k_i \) directly to \( \frac{1}{n \cdot 2^x} \cdot \log(\frac{2(n-1)}{\delta}) \). This comes from \( k_i = \frac{K}{N} \cdot N_i = \frac{K}{N} \cdot \frac{N}{n} = \frac{K}{n} = \frac{1}{n \cdot 2^x} \cdot \log(\frac{2(n-1)}{\delta}) \)

### 4.1.3 Local Load

We briefly summarize the local load inside TileDB for context. This is the final step that occurs in all of the loading techniques described above, after data shuffling is completed. At this point, each worker has a roughly equal-sized subset of CSV lines out of the global input CSV. Local load occurs in three broad steps.

1. Compute and inject cell_ids (if necessary). This is only needed in hash partitioning scheme because cell_ids were already computed prior to data shuffling for ordered partitioning.

2. Sort on cell_ids.

3. Make tiles from the the sorted CSV. This step creates the data files and bookkeeping structures on disk described in Section 2.2.1.
4.2 Filter and Subarray

Filter and subarray are embarrassingly parallel on both hash and ordered partition meaning that they require no data shuffle among nodes. We briefly summarize the algorithm for how they are processed locally in TileDB.

Filter  The filter query takes in as input an expression string and produces a new output array containing cells that satisfy the expression. TileDB takes advantage of its column oriented storage (Section 2.2.1) to minimize disk accesses when processing filter. It processes the filter query by first scanning only the attribute files related to the expression. The other data files are only scanned when the expression is satisfied. Qualifying cells are appended to a new output array.

Subarray  The subarray query takes in as input a range of coordinates and produces a new output array with cells in the range. This query is essentially a filter on the coordinates. TileDB optimizes this query by using the MBR (minimum bounding rectangles) index described in Section 2.2.2 to identify tiles completely within the range the tiles partially overlapping with the desired range. Tiles completely within the subarray range are directly copied into the output array. Then only the partially overlapping tiles are scanned and processed to find exact cells contained in the subarray range.

In Multinode-TileDB, once the coordinator receives a request for either query, we simply dispatch the request to the worker nodes for executing the query locally.

One thing to note is that for subarray on an ordered partition, the work performed by each worker may be skewed depending on where the subarray range is in the logical space. We observe this effect experimentally in Section 5.5.
4.3 Join

We are mainly concerned with joins on coordinates in Multinode-TileDB because we expect that to be most common case, though other joins may be considered for future work. Join on coordinates is also referred to as a dimension-to-dimension join. For convenience, we will refer to dimension-to-dimension joins simply as joins. The join query takes as input two arrays that are compatible, meaning they have the same number of coordinate dimensions with the same domains and also have the same sort order. The output array contains cells with matching coordinates from the two arrays and includes the attributes from both joined cells. Thus, a join of an array with $a$ attributes with another array of $b$ attributes produces an output array with $a + b$ attributes.

4.3.1 Join on Hash Partition

Similar to filter and subarray, join on hash partitioned data requires no data shuffle. Because we hash on coordinates, two cells with the same coordinates from two different arrays would be assigned to the same node. Thus, all of the cells that could possible join between two arrays are already co-located within the same worker.

4.3.2 Join on Ordered Partition

Join on ordered partition requires a data shuffle step to co-locate logical cells with the same coordinates from the two arrays on the same worker. We utilize the bounding_coordinates.bkp index (Section 2.2.1) on each worker node to determine inter-node, cross-array, tile overlaps between the two arrays. In other words, we want to determine the tiles from one array on one node that overlaps with the tiles from the other array on another node. After determining the existence of inter-node, cross-array, tile overlaps, we must then determine which node should send their overlapping tiles to the other such that overall communication costs are minimized.

As an example, Figure 4-1 shows arrays $A$ and $B$ partitioned across three nodes. Both are ordered partitioned such that smaller cell_ids reside on worker 1 and increase
Figure 4-1: Data Shuffling for Joins on Ordered Partition

toward worker 3. Computing cell_ids on the cell coordinates effectively linearizes the array. For simplicity, each node contains four tiles of fixed physical size (marked by dotted lines). Array $A$ is somewhat uniform and array $B$ is more sparse and skewed. Note that the figure shows the arrays in the logical space, which means that tiles with a smaller area represents a more dense region while tiles with a larger area represents a more sparse region. Node boundaries on the array are marked by bold-ed, solid lines. Only the relevant tiles involved in data shuffle are labeled. The highlighted tiles overlap in the logical space of the two arrays and must be co-located in the same node for local joins. To minimize data transfer in the network, we want to transfer the smaller of the two overlaps. In the example, the optimal situation is to transfer tile $t_{b5}$ from node 2 to node 1 and tile $t_{b8}$ from node 2 to node 3, as indicated by the arrows in the figure.

We outline the basic parallel join algorithm used by Multinode-TileDB in the steps below.

1. After receiving the join message from the coordinator node, each worker node reads its bounding_coordinates.bkp into memory. Then the entire node cluster performs an all-to-all data shuffle of all the bounding coordinates. Every worker node will end up with the bounding coordinates of every other node. In practice, this file is quite small so this step hardly take up bandwidth.
2. Every worker node computes its overlapping tiles with every other worker.

3. Every worker computes a cost matrix to determine which worker should send the data in each overlapping pair. The cost of a transfer is calculated as $\text{num\_overlapping\_tiles} \cdot \text{tile\_capacity}$, where $\text{tile\_capacity}$ is the number of cells in a tile. Recall that these tiles are of fixed physical size (Section 2.2.1) so cost directly correlates with amount of data to be transferred. In every overlapping node pair, the node with the lower cost will transfer its overlapping tiles to the other node. One other thing to note is that since every worker already has the bounding coordinates of every other node, it can compute the entire cost matrix of every potential overlapping pair of nodes. Thus, no extra communication is needed to determine who should send the overlapping data.

4. Upon receiving the overlapping tile (if any) from nearby nodes, each worker can now execute a local join. Note that we can still directly use a merge join for this step (see Section 4.3.3). From the point of view of a worker node, the extra tiles received from each overlapping neighbor must either fall completely before or completely after its local partition of the array, which maintains the sorted order. This is a direct result from an ordered partitioned load.

4.3.3 Local Join

Local join is the last step of join on ordered partition and the only step in join on hash partition. TileDB’s local join uses a merge join algorithm on the two joining arrays because they are already in the same sort order due to the join compatibility requirements. After verifying that two input arrays are join compatible, we scan the coordinate tiles of both arrays to look for matches. Once a match is found, the attributes from both cells are concatenated together and appended to the output. TileDB further optimizes the join by first determining if two potential tiles overlap in their bounding ranges. Coordinate tiles are only retrieved from disk and scanned if the two tiles actually overlap and possess the possibility of having cells that can join. TileDB uses the bounding_coordinates.bkp index (Section 2.2.1) to check for
overlapping tiles.
Chapter 5

Experiments

In this chapter, we present Multinode-TileDB’s performance on real world skewed and sparse data. We describe the experiment setup, the dataset, and compare the performance of the various load, join, and subarray queries described in Chapter 4. We omit filter because it does not provide additional insights apart from the previous queries.

5.1 Experiment Setup

Multinode-TileDB is implemented in C++, compiled with gcc 4.6.4 on machines running Ubuntu 12.04. We use MPICH2 [11] as the MPI implementation in our communication module (Section 3.2).

Experiments are conducted on a cluster of 13 machines. Each have 2.13 GHz, 256 GB RAM and either 32 or 64 processors and attached SATA hard disk with terabytes of storage. We use one processor per machine for our experiments. We run all the specified queries on varying input data sizes and varying cluster sizes. The data sizes are 4GB, 18GB, and 50GB and the cluster sizes are 2, 4, 6, 8, 10, 12 (excluding the coordinator node). We drop memory caches before each query execution.

Table 5.1 shows our default parameter settings.

41
Table 5.1: Default Experiment Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling Confidence $1 - \delta$ (Ordered Partition Loads)</td>
<td>.99</td>
</tr>
<tr>
<td>Error margin $\epsilon$ (Ordered Partition Loads)</td>
<td>.02</td>
</tr>
<tr>
<td>Tile Capacity</td>
<td>10000 cells</td>
</tr>
<tr>
<td>Max MPI Buffer Size</td>
<td>0.1MB</td>
</tr>
<tr>
<td>Cell Order</td>
<td>hilbert</td>
</tr>
</tbody>
</table>

Figure 5-1: AIS Dataset Sample from Jan 2011

5.2 Dataset

We use the AIS ship datasets for our experiments [2], which contain vessel tracking data collected by the United States Coast Guard. We use this real world dataset because the data naturally exhibits skew and sparsity, as ships tend to cluster around major ports. Figure 5-1 visualizes a sample of the dataset from January 2011. The figure clearly outlines the coastal regions of the United States and we can see dense regions near populous ports and sparse regions farther out to sea.

We use data collected from 2009 to 2011, constructing one dataset file for each month of each year. We combine different datasets together to create larger datasets
Table 5.2: Coordinate and Attribute types for AIS Dataset

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>coordX</td>
<td>int64</td>
</tr>
<tr>
<td>coordY</td>
<td>int64</td>
</tr>
<tr>
<td>SOG</td>
<td>float</td>
</tr>
<tr>
<td>COG</td>
<td>float</td>
</tr>
<tr>
<td>Heading</td>
<td>float</td>
</tr>
<tr>
<td>ROT</td>
<td>float</td>
</tr>
<tr>
<td>Status</td>
<td>int</td>
</tr>
<tr>
<td>VoyageID</td>
<td>int64</td>
</tr>
<tr>
<td>MMSI</td>
<td>int64</td>
</tr>
</tbody>
</table>

of varying sizes for the experiments.

Every data point contains the following items: coordX (longitude), coordY (latitude), SOG (knots), COG (degrees), Heading (degrees), ROT (degrees/min), Status, VoyageID, and MMSI. We model the data as a 2 dimensional array with coordX, in domain [0, 360000000], and coordY, in domain [0, 180000000], as the coordinates and the other 7 elements as the attributes. Table 5.2 shows the data types for each coordinate and attribute.

5.3 Load Performance

We load datasets of size 4GB, 18GB, and 50GB on clusters 2, 4, 6, 8, 10, and 12 machines for each dataset. For each dataset/cluster combination, we load the dataset 4 times, one for each of the initial data distributions (centralized or distributed) and for each of the data partitioning schemes (hash or ordered). For the centralized loads, the input dataset resides on the coordinator node. For the distributed loads, we pre-partition the input dataset across the worker nodes prior to the experiment such that each worker starts out with an equally sized, random partition of the global input.

In the following sections, we analyze the cost breakdown of each load type and their speedup and scaleup performance.
Figure 5-2: Load Execution Breakdown (18GB dataset)
5.3.1 Breakdown Costs

All the parallel loads described in Section 4.1 can be broken into 4 broad steps: 1) compute cell_ids and sample, 2) determine boundary points, 3) data shuffle, 4) local load. For hash partitioning, the first two steps are not needed. Table 5.3 briefly summarizes on where each step occurs for each load and in the case of data shuffle, whether all-to-all or one-to-all communication is used.

Figure 5-2 compares the query performances across multiple cluster sizes for loading a 18GB dataset and shows the cost breakdowns. The stacked bar graphs display average execution times of all workers involved in the query for each broad step of the load with minimum and maximum execution times shown as error bars.

From Figure 5-2, we see that the query times are dominated by computing cell_ids and sampling, data shuffle, and local load. The time for determining partition boundary points in the ordered partition cases is quite small. Recall that this step involves communicating samples to the coordinator node, sorting the samples, and finding the quantiles of the sorted samples. Table B.1 in the appendix shows the number of samples taken per node for both centralized and distributed cases for our experiments.

We also see that the execution times of all the steps generally decrease as cluster size increases in the distributed loads, which is desired behavior. There is much less speedup in the centralized loads. We also see that the distributed hash partition load performs the best. This makes sense because there is extra overhead in the distributed ordered partition load for sampling and determining boundary points. The reason that the data shuffle step in distributed ordered partition load is larger than in distributed
hash partition load is because the former sends more data overall. Specifically, in the distributed ordered partition load, we are sending the cell_ids computed in the sampling step so that they do not need to be recomputed during local load.

The local load step in the hash partitioned loads are longer than the ordered partitioned loads because it includes computing cell_ids. In the ordered partitioned loads, this was done at the beginning.

One last thing to note is that for all cluster sizes, there is small variance in query time among the workers. For hash partitioning, this was expected due to the nature of the algorithm. For ordered partitioning, this means that our formula for determining sample size is empirically sound and that our chosen confidence level and error margin are good enough for load balancing.

5.3.2 Speedup and Scaleup

Figure 5-3 shows the speed up and scale up performance of the loads across increasing cluster sizes and increasing dataset sizes. From the graph, we see that there is speed up for all 3 dataset sizes as the number of workers increases for both the distributed loads. However, there is little speed up for the centralized loads, as shown by their mostly flat lines. This is expected because the entire array is first processed at one location (coordinator node) and distributed to each worker. The initial processing step is most expensive for centralized ordered load which involves computing and writing out cell_ids.

For distributed loads, we also observe that the gap between distributed hash partition load and distributed ordered partition load increases as dataset size increases. As mentioned above, this is due to an increase in total data that must be sent during data shuffle from the computed cell_ids. Similarly, the cell_ids are also sent in the centralized ordered load, which farther exacerbates its parallel performance.
Figure 5-3: Load Speedup and Scaleup Performance
Figure 5-4: Join Execution Breakdown on Worker Nodes (50GB x 53GB)
5.4 Join Performance

In the join experiments, arrays of similar sizes are first loaded and then joined. Similar to the load experiments, we join datasets of 3 size categories: 4GB x 4GB, 18GB x 16GB, and 50GB x 53GB. The two 4GB datasets are each from a different month in 2009. The 18GB and 16GB datasets are from the first 6 months and last 6 months of 2009, respectively. The 50GB and 53GB datasets are from 2010 and 2011, respectively.

5.4.1 Breakdown Costs

Recall that there are 2 broad steps in parallel join: 1) data shuffle and 2) local join (the former is not needed for hash partitioned data). Figure 5-4 shows the performance breakdowns of join on both ordered and hash partitioned data of a 50GB array and a 53GB array for increasing cluster sizes. Once again, average worker execution times are shown in the stacked bars and minimum and maximum worker execution times are shown as error bars. We see that the data shuffle step in the ordered partition join varies for each cluster size but does not follow a clear pattern. This is because data placement is quite unpredictable for ordered partition particularly for skewed data so the amount of data that needs to be shuffled is unpredictable as well. Though we see that it does not take up more that 20% of the total cost for our datasets.

Another thing to note between the two joins is the noticeable difference in the range of time it takes the workers to execute a local join. We see that, in general, the workers on the hash partitioned data have a smaller range of query execution time than on ordered partitioned data. This means that the hash partitioned data leads to better load balancing on joins, which was expected.

5.4.2 Speedup and Scaleup

Figure 5-5 shows the speedup and scaleup performance of join for the 3 dataset size categories. The graph shows that joins on both partitions achieve near linear speedup,
Figure 5-5: Join Performance
with join on hash partition outperforming join on ordered partition since the former does not require data shuffle. However, we also see that, for each dataset size category, the performance gap widens between the two as the cluster size increases. This is because increasing the number of nodes increases the probability of having inter-node, cross-array, tile overlaps, and thus requiring more data communication.

5.5 Subarray Performance

We execute the same subarray query on datasets of size 4GB, 18GB, and 50GB for both ordered and hash partitioning. The query itself bounds a geographic region on a small area near New York City. The resulting sizes are 0, \sim400K, and \sim1M cells for the respective dataset sizes.

5.5.1 Breakdown Costs

As mentioned multiple times before, parallel subarray is embarrassingly parallel, so the only step on each worker node is executing the query locally. Figure 5-6 shows the average execution time of local subarray on both ordered and hash partitioned data, with the minimum and maximum times shown as error bars. It is interesting to observe that, in contrast to load (Figure 5-2) and join (Figure 5-4), this particular subarray performs better, on average on ordered partitioned data than on hash partitioned data. This can be explained by the lack of spatial locality in hash partitioned data. Specifically, due to the uniform randomness of hash functions, each worker will essentially contain a random subset of the array. Thus, the cells that actually fall in the subarray range will be scattered on multiple workers, which leads to each worker doing roughly equal amounts of work.

In contrast, on ordered partitioned data, the cells contained in the subarray are likely to only fall on a few nodes. Those nodes will end up doing more work than the nodes that do not have overlaps with the requested region. Recall that TileDB can use the MBR bookkeeping structure to prune away non-overlapping tiles without reading them from disk (Section 2.2.1).
Figure 5-6: Subarray Execution Breakdown on Worker Nodes (50GB)
Thus, as Figure 5-6 shows, there is a large range in execution time for the workers on an *ordered* partitioned array versus a *hash* partitioned array.

### 5.5.2 Speedup and Scaleup

Figure 5-7 shows the speedup and scaleup performance of *subarray* on both partition types for all three dataset sizes. We see that *subarray* does not scale as well as *load* and *join*. The optimizations employed by *TileDB*, especially with the *mbr.bkp* bookkeeping structure, makes the performance hard to predict, particularly with...
regards to which nodes cells end up on. And, of course, performance will vary widely depending on the requested subarray range, which will need future experimenting. In our experiment, the requested subarray range is very small.

5.6 Results and Analysis

The preliminary experiments presented above give a good insight to the performance of parallel load, join, and subarray. As a summary, both distributed load and join perform better on hash partitioned arrays but that is not the case for subarray. Furthermore, there is fairly good speedup and scaleup for the distributed loads and joins on both partition types. There is less speedup for subarray and very little speedup for centralized load.
Chapter 6

Conclusion and Future Work

This thesis presented a design and implementation of Multinode-TileDB, a distributed infrastructure for TileDB, which includes a communication module and several algorithms for parallel queries. Our design focuses on load balancing and performance of parallel query processing on skewed and sparse arrays. We study two type of data partitioning (hash and ordered) among nodes in a cluster, and implement parallel database queries for both types, including parallel load, join, subarray, and filter. We analyzed the performance of the queries by experimenting on real world datasets. Our results show that hash partitioned load is faster than ordered partitioned load. They also show that a hash partitioned array is definitely more beneficial for join but necessarily for subarray.

6.1 Future Work

Multinode-TileDB is only an initial study into the complexities of building a distributed database for sparse and skewed arrays. Our preliminary experimental results give insights as to how to optimize the current system. In addition, there are also many directions for related future work. They include:

Parallel Linear Algebra and Machine Learning Operations The rapid increase of data requires distributed support for machine learning algorithms. A nat-
ural extension to this thesis is to design and implement machine learning algorithms using this infrastructure, targeted at skewed arrays.

**Fault Tolerance** All distributed systems require fault tolerance for practical use. Fault tolerance in general is a fairly well studied topic but introducing fault tolerance properties into *Multinode-TileDB* while maintaining performance will be quite a challenge.

**Elasticity** Related to fault tolerance, which involves nodes recovering from failure, elasticity refers to nodes intentionally coming and leaving the network. Challenges include how to maintain load balance and performance in the presence of nodes joining and leaving and how to repartition data as the database grows and shrinks.
Appendix A

Sampling Size Proof

Recall that we made the following claim in Section 4.1 when discussing load.

**Claim 1.** *In order to find \( p \) partition boundaries with error margin \( \epsilon > 0 \) and confidence \( 1 - \delta \), where \( 0 \leq \delta \leq 1 \), we must sample at least \( \frac{1}{2\epsilon^2} \cdot \log\left(\frac{2N}{\delta}\right) \) points from the input dataset.*

We will formally define the problem and then outline the proof. This is an adaption of a proof for a more complicated algorithm by Manku et al [7].

Let the \( \phi \)'th quantile on a dataset of size \( N \), where \( 0 \leq \phi \leq 1 \), be defined as the element in position \( \lceil \phi N \rceil \) of the sorted dataset. Let error margin \( \epsilon \), where \( 0 \leq \epsilon \leq \phi \) be defined such that an element within \( \epsilon \) approximation of quantile \( \phi \) is between positions \( \lceil (\phi - \epsilon)N \rceil \) and \( \lceil (\phi + \epsilon)N \rceil \) of the sorted dataset.

We will need two lemmas. The first is Hoeffding’s inequality [7, 6], stated below.

**Lemma 1** (Hoeffding’s Inequality). *Let \( X_1, X_2, \ldots, X_n \) be independent random variables such that \( 0 \leq X_i \leq 1 \) for each \( i = 1, 2, \ldots, n \). Let \( X = \sum_{i=1}^{n} X_i \) and let \( E[X] \) be the expected value of \( X \). Then for any \( \lambda > 0 \), \( P(X - E[X] \geq \lambda) \leq \exp\left(\frac{-2\lambda^2}{n}\right) \).

**Lemma 2.** *A sample size of \( S \geq \frac{1}{2\epsilon^2} \cdot \log\left(\frac{2N}{\delta}\right) \) drawn from a set of \( N \) elements guarantees, with probability \( 1 - \delta \), that the \( \phi \)'th quantile in the sorted sequence of \( S \) samples is contained in the subset of elements between the positions \( \lceil (\phi - \epsilon)N \rceil \) and \( \lceil (\phi + \epsilon)N \rceil \) of the original sorted dataset with size \( N \).*
Proof. Let \( N_{\phi-\epsilon} \) and \( N_{\phi+\epsilon} \) denote the set of elements preceding the \((\phi - \epsilon)\)'th and succeeding the \((\phi + \epsilon)\)'th quantiles of \( N \) elements, respectively.

A sample is bad if it does not follow the properties of the previous lemma, otherwise, it is good. Thus, a sample is bad if and only if it contains more than \( \lceil \phi S \rceil \) elements from \( N_{\phi-\epsilon} \) or more than \( S - \lceil \phi S \rceil \) elements from \( N_{\phi+\epsilon} \).

We will first consider the \( N_{\phi-\epsilon} \) case. We can model the process of drawing \( S \) samples as \( S \) independent coin flips, with a \( \phi - \epsilon \) chance of "success" (the sample falls below position \( \lceil (\phi - \epsilon)N \rceil \) of the sorted input). The expected number of "successful" coin flips is \((\phi - \epsilon)S\). If the sample has more than \( \phi S \) successful trials, then it is considered bad. Each draw of the sample is an independent random variable, \( X_i \in [0,1] \) where 0 denotes failure and 1 denotes success. Let \( X = \sum_{i=1}^{n} X_i \) indicate the number of samples falling in the bad range. The probability of drawing a sample containing more than \( \lceil \phi S \rceil \) elements from \( N_{\phi-\epsilon} \) is stated as follows.

\[
P(X \geq \phi S) = P(X - E[X] \geq \phi S - (\phi - \epsilon)S) = P(X - E[X] \geq \epsilon S)
\]

Applying Hoeffding’s Inequality, we observe that this probability is bounded by \( \exp(-\frac{2(\epsilon S)^2}{S}) = \exp(-2\epsilon^2 S) \).

The \( N_{\phi+\epsilon} \) case is symmetric. In this case, the coin flip is "successful" if the position of the drawn sample falls above the position \( \lceil (\phi - \epsilon)N \rceil \) position of the sorted input. The expected number of successful coin flips is then, \((1 - (\phi + \epsilon))S\). A sample containing more than \( S - \lceil \phi S \rceil \) "successful" trials is considered bad. Thus, the probability of drawing a sample containing more than \( S - \lceil \phi S \rceil \) elements from \( N_{\phi+\epsilon} \) is

\[
P(X \geq S - \lceil \phi S \rceil) = P(X - E[X] \geq S - \phi S - (1 - (\phi + \epsilon))S) = P(X - E[X] \geq \epsilon S)
\]
The probability of drawing a sample containing more than \( S - \lfloor \phi S \rfloor \) elements from \( N_{\phi+\epsilon} \) is also bounded by Hoeffding’s Inequality as \( \exp(-2\epsilon^2 S) \).

Thus, the entire probability of drawing a bad sample pool, \( \delta \), is bounded by 
\[
2 \cdot \exp(-2(\epsilon^2 S)).
\]
Solving for \( S \), we obtain 
\[
S = \frac{1}{2\epsilon^2} \cdot \log\left(\frac{2}{\delta}\right).
\]

Lemma 2 gives us the sample size while considering only one quantile. In Claim 1, we are interested in \( p \) quantiles. This is easily argued from the lemma (also adapted from [7]). Now, suppose we are selecting \( p \) quantiles. Let \( \delta' \) be \( \frac{\delta}{p} \). By Lemma 2, the probability that one quantile is not \( \epsilon \) approximate is bounded by \( \delta' \). The probability that any one of the \( p \) quantiles is not \( \epsilon \) approximate is, thus, \( \delta \). Thus, the final sample size is 
\[
\frac{1}{2\epsilon^2} \cdot \log\left(\frac{2p}{\delta}\right).
\]
Appendix B

Sample Sizes

Table B.1 shows the minimum sample sizes needed in our ordered partition load algorithms for a confidence of 0.99 and an error margin of 0.02 for each different cluster size used in our experiments. In the centralized load case, only the coordinator draws samples from the input (indicated by the total samples column in the table). In the distributed load case, each worker draws the indicated number of samples from their respective input CSV splits corresponding to a particular cluster size.

Table B.1: Number of Samples for Confidence $1 - \delta = .99$ and Error Margin $\epsilon = .02$

<table>
<thead>
<tr>
<th>Num Workers</th>
<th>Total Samples</th>
<th>Samples Per Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6623</td>
<td>3311</td>
</tr>
<tr>
<td>4</td>
<td>7997</td>
<td>2000</td>
</tr>
<tr>
<td>6</td>
<td>8635</td>
<td>1440</td>
</tr>
<tr>
<td>8</td>
<td>9056</td>
<td>1131</td>
</tr>
<tr>
<td>10</td>
<td>9367</td>
<td>937</td>
</tr>
<tr>
<td>12</td>
<td>9621</td>
<td>802</td>
</tr>
</tbody>
</table>
Bibliography


