# FOUNDATIONAL ALGORITHMS UNDERLYING HORIZONTAL PROCESSING OF VERTICALLY STRUCTURED BIG DATA USING PTREES

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Mohammad Kabir Hossain

In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

Major Department:
Computer Science

March 2016

Fargo, North Dakota

**NORTH DAKOTA STATE UNIVERSITY**

Graduate School

**Title**

# FOUNDATIONAL ALGORITHMS UNDERLYING HORIZONTAL PROCESSING OF VERTICALLY STRUCTURED BIG DATA USING PTREES

**By**

Mohammad Kabir Hossain

# ABSTRACT

For Big Data, the time taken to process a data mining algorithm is a critical issue. Many reliable algorithms are unusable in the big data environment due to the fact that the processing takes an unacceptable amount of time. Therefore, increasing the speed of processing is very important. To address the speed issue we use horizontal processing of vertically structured data rather than the ubiquitous vertical (scan) processing of horizontal (record) data. pTree technology represents and processes data differently from the traditional horizontal data technologies. In pTree technology, the data is structured column-wise (into bit slices) and the columns are processed horizontally (typically across a few to a few hundred bit level columns), while in horizontal technologies, data is structured row-wise and those rows are processed vertically. pTrees are lossless, compressed and data-mining ready data structures. pTrees are lossless because the vertical bit-wise partitioning that is used in the pTree technology guarantees that all information is retained completely. There is no loss of information in converting horizontal data to this vertical format. pTrees are data-mining ready because the fast, horizontal data mining processes involved can be done without the need to reconstruct the original form of data. This technique has been exploited in various domains and data mining algorithms, ranging from classification, clustering, association rule mining, as well as other data mining algorithms. In this research work, we evaluate and compare the speeds of various foundational algorithms required for using this pTree technology in many data mining tasks.

## ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. William Perrizo. His guidance was invaluable in helping me finish this research work. His enthusiasm in and profound knowledge of the discipline of computer science are a source of inspiration to me.

I am very grateful to my committee member Dr. Saeed Salem for his unfailing support and advising enabling me to successfully overcome the challenges throughout my graduate studies. I would like to express my gratitude to my other committee members, Dr. Gregory Wettstein and Dr. Mukhlesur Rahman, for taking the time to help me to finish this work.

I would also like to thank my colleagues in the DataSURG group Arjun Roy, Arijit Chatterjee, Damian Lamp and Maninder Sing for all the help they gave to me during my research. Their assistance was crucial for my completion of this work.

Last but not the least, I would like to thank all my family members here in US and in my home country Bangladesh. Without their support it would have been impossible for me to finish. Specially I would like to acknowledge the support of my wife Fatema Shireen who managed everything all by herself while I was busy in my lab. Thanks goes to my 7th grader son Nabeel who time to time monitored my progress (well, within his capacity). Also thanks to my 2 years and 9 months old daughter Nyla and my 8 months old son Najeef who inspired me to finish this work.

## DEDICATION

To the memory of my father Md. Ruhul Amin, who sent me to achieve this success but is only seeing this from heaven, and to my mother Jahanara Begum, who is continuously praying for me to be successful in every aspect of my life.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF SYMBOLS

| | |
|---|---|
| $pTree$ | Predicate tree |
| $LSP$ | Least significant pTree |
| $MSP$ | Most significant pTree |
| $L1D$ | $L_1$ Distance |
| $SED$ | Squared Euclidean Distance |
| $DP$ | Dot product |

# CHAPTER 1. INTRODUCTION AND RELATED WORK

As a result of the advancement of digital technology enormous amount of data are collected from different sources like satellites, customer checkout from super stores, stock exchange, social media, images from video surveillance camera etc. The volume of data often becomes very large ranging from hundreds of gigabyte to several terabytes[22, 21]. This type of large and complex data is known as "Big data". In [16] big data is defined as "Big data is data that exceeds the processing capacity of conventional database systems. The data is too big, moves too fast, or does not fit the strictures of your database architectures. To gain value from this data, you must choose an alternative way to process it."

Big data is characterized by 3-V properties [17]. These are volume, variety and velocity. Volume refers to the fact that the data set is enormous in size, measured in gigabytes (GB), terabytes (TB), petabytes (PB), etc. Variety means the types of data. In addition, difference sources will produce big data such as sensors, devices, social networks, the web, mobile phones, etc. For example, data could be web logs, RFID sensor readings, unstructured social networking data, streamed video and audio. Velocity means how frequently data is generated and processed. Data may be generated in every millisecond, second, minute, hour, day etc. Based on the application the data may be processed real-time or when needed.

For mining big data efficiently for the purpose of decision making or decision support, there is a number of challenges related to it. Due to its characteristics some of the inherent challenges in big data are capture, storage, search, processing and visualization [44]. Among them the biggest challenge is fast processing of data with an acceptable degree of accuracy. There are many algorithms that can process data accurately but when the volume of the data grows, they fail to process the data fast enough for getting the processing result in a reasonable amount of time. Thus the algorithms lack to be scalable in big data environment.

In our research we attempt to solve this problem using a data mining ready vertically structured data representation known as predicate tree or pTree [20].

Predicate trees are constructed by vertically slicing the attributes of a data set and then further slicing the attributes into their bits after the values are converted into binary. Each bit slice is called raw pTree or level-0 pTree. On top of a level-0 pTree other level pTrees can be built resulting multi-level pTrees. The basic operations performed on a pTree are AND, OR, NOT operations which are essentially the bit-wise operations. Using these bit wise operations other mathematical and relational operations are implemented which are necessary for implementing different data mining algorithms[10, 11].

## 1.1. Data mining

In general data mining refers to discovering of useful hidden information within a (often large) data set which are not readily available. We need to take help of some techniques to find those information. In [43] data mining is defined as “Data mining is the process of discovering insightful, interesting, and novel patterns, as well as descriptive, understandable, and predictive models from large-scale data”. Among the techniques used for data mining association rule mining, classification and clustering are most widely used.

### 1.1.1. Association rule mining

Association rule mining (ARM) is a method that finds the interesting relations between variables in a large data set [32]. Each rule has two parts antecedent and consequence where a set of variables in the antecedent implies the another set of variables in the consequence with a confidence level showing how strong the rule is. In [6] customers’ buying patterns of different product in a super store based on strong rules are shown. Interstingness of a rule is measured by two parameters namely support and confidence. Support of a rule indicates the proportion in the database which contains the variables in the antecedent part and confidence of a rule is the proportion that contains the variables in the antecedent part which also contains the variables in the consequence part. Usually we look after high support and high confidence rules as they are considered as strong rule but in some application such as outlier detection we also look for low support but high confidence rule [30].

### 1.1.2. Classification

Classification predicts the class of an unclassified data sample [23]. Classification is also known as supervised learning because of the fact that there is training data set with known class label available to predict the class label of data sample of unknown class label. Based on the use of this training label there may be two types of classifier [41]. One of them is known as model based classifier which builds a classification model using the training data set. It then uses this model to classify unknown samples. Other type of classifier is known as lazy classifier which does not build any model ahead of time rather it takes into account the whole training set to classify an unknown sample.

### 1.1.3. Clustering

Clustering is a technique to group similar types of data together into clusters [24]. Clustering is also known as unsupervised learning because there is no class label associated with the data samples. The process of clustering tries to increase the similarity between the data points within a cluster and decrease the similarity between the clusters [40].

## 1.2. Technologies related to big data

There are many very useful and fundamental technologies that are closely related to and widely used in big data [13]. In this section we are discussing some of them such as cloud computing, Internet of Things (IoT), Business Intelligence (BI) and Hadoop.

### 1.2.1. Cloud computing

Big data and cloud computing are closely related to each other. Computation intensive operations of cloud computing is done on big data and big data stresses the storage of a cloud system. Big data takes the advantage of huge computing and storage resources of cloud computing which are managed under concentrated management. Thus it provides big data applications necessary computing capacity. The development of cloud computing provides solutions for the storage and processing of big data. On the other hand, the emergence of big data also accelerates the development of cloud computing [18].

### 1.2.2. Internet of Things(IoT)

In IoT a huge amount of networking sensors are embedded into various equipments and machines [27]. These sensors that are deployed in real world collect data of different kinds like environmental data, geographical data, astronomical data, and logistic data thus generating bid data. This big data has different characteristics compared with general big data because of the different types of data collected. The most classical characteristics include heterogeneity, variety, unstructured feature, noise, and high redundancy. Successful implementation of IoT relies on effective integration of big data and cloud computing. The widespread deployment of IoT will also bring many cities into the big data era.

### 1.2.3. Business intelligence (BI)

Business intelligence is a collection of decision support technologies designed to report, analyze, and present data [12]. These technologies are often used to read data that have been previously stored in a data warehouse or data mart. They enable knowledge workers such as executives, managers, and analysts to make better and faster decisions.

### 1.2.4. Hadoop

Hadoop is an open source software framework for processing huge data sets on a distributed system [15]. Its development was inspired by Googles MapReduce and Google File System. It was originally developed at Yahoo! and is now managed as a project of the Apache Software Foundation.

## 1.3. Vertical Data structure

In data mining algorithms it is very often assumed that the data being structured as a relational table in a database or a data cube in a data warehouse [19] where data are stored horizontally meaning row by row. This kind of horizontally structured records and scan-based data processing is known to be ineffective for mining big data as horizontal methods of data mining do not scale with a very large dataset [37].

Over the years, there has been a slow but increasing focus on the vertical database (also known as column-oriented database) management systems (DBMSs). Vertical DBMSs are different from the traditional ones in the way data is stored and accessed. This type of databases store the data column-wise. This allows vertical DBMSs to have extensive usage in various data warehouse applications because of their better performance in terms of read I/O in comparison to the conventional DBMSs. This is primarily due to the fact that vertical DBMSs only retrieve the columns defined in the query rather than the entire row as in case of traditional DBMSs [37].Consider a student table containing attributes name, age, gender and grade. In a traditional database, the student table is stored row-wise, one student record followed by another. In a vertical database, the table is vertically sliced, i.e., each attribute is stored in a separate individual file. In this section, we discuss few major open-source and column-oriented Database Management Systems, namely, C-Store, Infobright, InfiniDB, MonetDB and pTree.

### 1.3.1. C-Store

C-Store was a collaborative research project at MIT which has now been developed and commercially called as Vertica. It brought lot of novel and interesting features to the column-oriented architecture such as efficient packing of objects, use of overlapping projections to store the data, query optimizer and executor based on columns, etc. [39].

### 1.3.2. Infobright

Infobright is a database and warehouse system available in commerical as well as free community edition. The key idea that sets this system aside from other vertical systems is the use of Knowledge Grid - a small metadata layer in place of the regular indexes. Knowledge Grid consisting of Knowledge Nodes are much smaller in size in comparison to the regular indexes and thus allow much faster as well as inmemory processing. The main functionality of Knowledge Nodes is to describe chunks of compressed data also called the Data Packs [38].

### 1.3.3. InfiniDB

InfiniDB is an efficient, multi-threaded analytic database system based on column-oriented storage architecture. Similar to Infobright, InfiniDB is available in two versions - Community Edition available free under GPL Licence and the commerical Enterprise Edition (scaled up version). InfiniDB is equipped with a comprehensive list of features [1]. InfiniDB uses an automated mix of vertical and horizontal partitioning. While the vertical partitioning allows faster processing by bringing only selected columns in the memory, a logical horizontal partitioning helps in reducing overall I/Os in horizontal and vertical direction.

### 1.3.4. MonetDB

MonetDB is one of the most mature columnoriented database system. Developed by Centrum Wiskunde and Informatica, Netherlands, MonetDB has purely been a research project since its inception in 1995 [7]. Over the years, it has developed signifi- cantly in different aspects of database management system and currently hosts a family of products such as XQuery [9, 7], MonetDB-GIS [7, 8], etc.

### 1.3.5. pTree

pTrees are lossless because the vertical bit-wise partitioning that is used in the pTree technology guarantees that all information is retained completely. There is no loss of information in converting horizontal data to this vertical format. pTree vertical data structures have been exploited in various domains and data mining algorithms, ranging from classification [4, 5, 25], clustering [31, 42], association rule mining [33, 34], to outlier analysis [35] as well as other data mining algorithms. pTree technology is patented in the United States by NDSU. Treeminer Inc.[2] has licensed the pTree patents while Dr. William Perrizos DataSURG group is further developing the technology, including better algorithms for pTree processing and processing on multi-core CPUs, GP-GPUs and FPGAs.

## 1.4. Problem description

The main challenge of this research work is to perform different mathematical operations over vertically structured large data set using pTrees in order to overcome challenges of Big Data processing. We know one of the biggest issue of Big Data processing is that it is often impossible to finish the processing job in reasonable amount of time due to its size. Many efficient data mining algorithms fail to scale due to the size of the data set.

Our focus in this research to provide a solution to execute basic mathematical operations in a way that the size of the data does not influence the performance of the algorithms designed for these operations. In order to achieve this, all the algorithms we designed will perform various logical operations across the pTrees which are used to represent the data set. The algorithms must not loop through the bits of the pTrees.

Over the years many different types of operations are implemented using pTrees. In [14] different types of aggregate functions are implemented using the basic operations. They include Sum, Average, Max, Min, Median/Rank, Top-k etc. In [31], the summation of square distances are calculated using pTrees. Although these work have significant contribution to the pTree based vertical data mining techniques, some of them are not true vertical processing as they need to scan the data point horizontally one-by-one and take the decision regarding that data point.

In this research work focused on development and implementation of some foundational operations that will be performed on vertically structured data sets which are represented in pTrees. We developed the following mathematical operations:

- Addition: We developed two addition algorithms. The first algorithm adds two attributes of a data set represented in pTrees while the other algorithm adds a constant value with an attribute of a data set.

- Subtraction: Like addition we developed two subtraction algorithms. One algorithm subtract an attribute from another and other subtract a constant value from an attribute.

- Multiplication: Multiplication algorithm has two variation as well. One of them multiply two attributes and the other multiply an attribute by a constant value.

We then use these algorithms to efficiently calculate three important measurements used in different data mining techniques. They are:

- $L_1$ Distance

- Squared Euclidean Distance

- Dot Product

We also developed an algorithm to compare two attributes of a data set to show if the values of one attribute is greater than or equeal to or less than the corresponding value of another attribute.

We then compared the efficiency of the algorithms with the horizontal processing of the same operation.

## 1.5. Organization of this dissertation

In chapter 2 we discuss the Boolean algebra and how it is used to implement different mathematical operations. We also define 2's complement and describe its use in representing negative integers. In chapter 3 we review the pTree technology in detail. We discuss some basic definitions of pTree technology, show the construction of pTree and pTree set, explain the advantages of pTree and discuss the steps to follow for a pTree processing. In chapter 4 we discuss the existing pTree based algorithms. Then we show our newly developed algorithms to perform different mathematical operations. Next in chapter 5 we discussed experimental design to design experiments to prove the better performance of our algorithms. Later in this chapter we analyzed the results of our experiments. Then we finished this dissertation by our concluding remarks and future research direction in chapter 6.

# CHAPTER 2. BOOLEAN ALGEBRA AND ITS USE

Boolean algebra was introduced by George Bool in 1854 for systematic treatment of logic [28]. It is defined on a set of two elements $B = \{0, 1\}$, with rules of two binary operators $\boldsymbol{AND}$ and $\boldsymbol{OR}$ and a unary operator $\boldsymbol{NOT}$ as shown in the following truth table. The $\boldsymbol{AND}$ operator produces 1 if and only if both the operands are 1, otherwise the result is 0. On the other hand the $\boldsymbol{OR}$ operator produces 0 if and only if both of its operands are 0, otherwise the result is 1. The $\boldsymbol{NOT}$ operation just flips the value from 0 to 1 or from 1 to 0.

| X | Y | X ***AND*** Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(a)

| X | Y | X ***OR*** Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

(b)

| X | ***NOT*** (X) |
|---|---|
| 0 | 1 |
| 1 | 0 |

(c)

**Fig. 1.** Truth Table for - (a) *AND*, (b) *OR* and (c) *NOT* operations.

Operations showed in figure 1 are the basic Boolean operations but we can derive some other complex operations using these operations. Of them the one we are most interested is the $\boldsymbol{XOR}$ (Exclusive OR) operation which is defined as:

$$X\,\boldsymbol{XOR}\,Y = X\,\boldsymbol{AND}\;\boldsymbol{NOT}\,(Y)\,\boldsymbol{OR}\;\boldsymbol{NOT}\,(X)\,\boldsymbol{AND}\,Y$$

What it is basically doing is when both the operands are same it produces 0, otherwise produces 1. Figure 2 shows the behavior of $\boldsymbol{XOR}$ operation:

Another complex operation is $\boldsymbol{XNOR}$ operation which is defined as:

$$X\,\boldsymbol{XNOR}\,Y = X\,\boldsymbol{AND}\,Y\,\boldsymbol{OR}\;\boldsymbol{NOT}\,(X)\,\boldsymbol{AND}\;\boldsymbol{NOT}\,(Y)$$

| X | Y | X ***XOR*** Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Fig. 2.** Truth table of *XOR* operation

This operation, also known as equivalence operation, produces 1 when both the operands are same (both are either 0 or 1) and produces 0 when both of its operands are different. So this operation is used to compare the equality of two bits. Figure 3 shows the behavior of ***XNOR*** operation:

| X | Y | X ***XNOR*** Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Fig. 3.** Truth table of *XNOR* operation

### 2.1. 2's Complement and its use

In binary number system complement is used to compute the negative equivalent of an integer. There are two types of complement in binary number system. They are two's complement and 1's complement. Here we are defining them.

**Definition 1** (2's Complement)**.** *Assume a binary number $N$ of $n$ bits. 2's complement of $N$ is defined as:* $2^n - N$

**Definition 2** (1's Complement)**.** *Assume a binary number $N$ of $n$ bits. 1's complement of $N$ is defined as:* $2^n - N - 1$

**Lemma 1.** *For any binary number of N,*

*2's complement of N = 1's complement of N + 1*

***Proof:*** *Assume* $N$ *has* $n$ *bits. So according to the definition 1, 2's complement of* $N = 2^n - N$ $= 2^n - N - 1 + 1$ = *1's complement of* $N + 1$ *[using definition 2]*

**Lemma 2.** *Complement of the complement of a number gives the original number*

***Proof:*** *Assume a binary number* $N$ *of* $n$ *bits. Assume its 2's complement of* $N$ *is* $M$ *which will also be a n-bit number. So according to definition 1,* $M = 2^n - N$*. Now, 2's complement of* $M = 2^n - M = 2^n - (2^n - N) = N$*, hence proved.*

## 2.2. Subtraction Using 2's complement

Assume two $n$-bit binary numbers $M$ and $N$. When we add 2's complement of $N$ with $M$, mathematically we get the result of $M - N$ in the following way:

$M$ + 2's complement of $N$

$= M + 2^n - N$ [using definition 1]

$= 2^n + (M - N)$

Thus ($M$ + 2's complement of $N$) gives us $(M - N)$ plus $2^n$ which is the $(n+1)$th bit also known as the carry bit of this addition operation. So if $M \geq N$, ($M$ + 2's complement of $N$) gives the $M - N$ after discarding the carry. If $M < N$, ($M$ + 2's complement of $N$) = $2^n - (N - M)$ = 2's complement of $(N - M)$. That is, ($M$ + 2's complement of $N$) has no carry in this case. It also indicates that the result of $(M - N)$ is negative and is represented in the 2's complement form. So taking 2's complement of ($M$ + 2's complement of $N$) will give the absolute value of the $(M - N)$ using lemma 2.

## 2.3. Calculation of summation of two integers

The algorithm presented in this paper utilizes the procedures of adding two numbers represented in binary bits starting from adding two single bit numbers then expanding the process to add two arbitrary n-bit numbers. When adding two single bit numbers (assume a and b) the possible result we shall get is a two bit number where the least significant bit is the sum (s) and most significant bit is the carry (c).

| a | b | c | s |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

**Fig. 4.** Truth table of adding two bits

The following bit wise operations compute the value of s and c from a and b

$$s = a\,\boldsymbol{XOR}\,b$$

$$c = a\,\boldsymbol{AND}\,b$$

Now assume we have two $n$-bit numbers $A$ and $B$ which are represented in binary as follows:

$$A = a_{n-1}a_{n-2}\dots a_1a_0$$

$$B = b_{n-1}b_{n-2}\dots b_1b_0$$

To add two number of n-bit each, the algorithm starts from least significant bit (which is bit 0) and proceed to left to the more significant bits and ends at the most significant bit (which is bit n-1). At any step i of these steps the algorithm adds three bits $a_i$ (the ith bit of $A$), $b_i$ (the $i^{th}$ bit of $B$) and $c_i$ (the carry produced as a result of similar operation in step (i-1) except for i=0 where c=0). Following operations are executed in step i:

Assume when adding $a_i$ and $b_i$ we get sum $s_1$ and carry $c_1$. So

$$s_1 = a_i \boldsymbol{XOR}\, b_i$$

$$c_1 = a_i \boldsymbol{AND}\, b_i$$

Now we need to add $s_1$ with $c$ and we will get final sum $s_i$ and we will get another carry $c_2$ as follows:

$$s_i = s_1 \boldsymbol{XOR}\, c_i$$

$$c_2 = s_1 \boldsymbol{AND}\, c_i$$

The final carry will be the $\boldsymbol{OR}$ between $c_1$ and $c_2$. So our final equations to get sum and carry will be:

$$s_i = a_i \boldsymbol{XOR}\, b_i \boldsymbol{XOR}\, c_i$$

$$c_{i+1} = (a_i \boldsymbol{AND}\, b_i) \boldsymbol{OR}\, (a_i \boldsymbol{XOR}\, b_i) \boldsymbol{AND}\, c_i$$

## 2.4. Subtraction of integers

As shown in section 2.3 we showed how we can add two integers. We can extend this idea to calculate subtraction of two integers directly. Similar to the figure 4 we can construct truth table to subtract two bits. Figure 5 shows the truth table to subtract two bits where bit $b$ is subtracted from $a$ to produce sum $s$ and borrow $r$.

| a | b | r | s |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 |

**Fig. 5.** Truth table for subtracting two bits

The following operations compute the value of s and r from a and b

$$s = a\,\boldsymbol{XOR}\,b$$

$$r = \boldsymbol{NOT}\,(a)\,\boldsymbol{AND}\,b$$

Finally to subtract an integer $B$ from $A$ to get $S$ we need to follow these equations:

$$s_i = a_i\,\boldsymbol{XOR}\,b_i\,\boldsymbol{XOR}\,r_i$$

$$r_{i+1} = (\boldsymbol{NOT}\,(a_i)\,\boldsymbol{AND}\,b_i)\,\boldsymbol{OR}\,(\boldsymbol{NOT}\,(a_i)\,\boldsymbol{XOR}\,b_i)\,\boldsymbol{AND}\,r_i$$

The sign of the result depends on the bit $S_n$. If $S_n$ is 0 the result is positive and $S_n$ is 1 then the result is negative. In that case the result is in 2's complement form and we would need to take complement again to get the absolute value of the result.

## 2.5. Multiplication of two integers

When we multiply two bits $a$ and $b$ the result $p$ is 1 when both bits are 1. If any of the bits is 0 or both of them are 0 the result is 0. That is exactly same as $\boldsymbol{AND}$ operation between $a$ and $b$. Notice that there is no carry in multiplying two bits.

$$p = a\,\boldsymbol{AND}\,b$$

In order to multiply an integer $A$ by $B$ first we multiply each bit $a_i$ of $A$ $\forall i$ by the LSB of $B$ i.e. $b_0$. Then we store them in $S$ as:

$$s_i = a_i\,\boldsymbol{AND}\,b_0 \forall i$$

Then we multiply each bit $a_i$ of $A$ $\forall i$ by the next bit of $B$ i.e. $b_1$ and add them with $S$ after shifting 1 bit position. And this way continue to the MSB of $B$. Then we get the final result of $A * B$ in $S$.

## 2.6. Comparing two integers

If we compare two bits $a$ and $b$ there are three possible results namely $a = b$ or $a > b$ or $a < b$. Figure 6 shows the truth table of these results which are denoted by $EQ$, $GT$ and $LT$ respectively. Following equations generates these results.

| a | b | EQ | GT | LT |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |

**Fig. 6.** Truth table for comparing two bits

$$EQ = a\boldsymbol{XNOR}\,b$$

$$GT = a\boldsymbol{AND}\,\boldsymbol{NOT}\,(b)$$

$$LT = \boldsymbol{NOT}\,(a)\boldsymbol{AND}\,b$$

Assume two $n$-bit numbers $A$ and $B$ which are represented in binary as follows:

$$A = a_{n-1}a_{n-2}\dots a_1a_0$$

$$B = b_{n-1}b_{n-2}\dots b_1b_0$$

Here $A$ will be equal to $B$ if the following condition is satisfied:

$a_{n-1} = b_{n-1}\ and\ a_{n-2} = b_{n-2}\ and \cdots and\ a_1 = b_1\ and\ a_0 = b_0$

$A$ will be greater than $B$ if the following conditions are met:

$a_{n-1} > b_{n-1}$

$Or, a_{n-1} = b_{n-1} \ and \ a_{n-2} > b_{n-2}$

$\vdots$

$Or, a_{n-1} = b_{n-1} \ and \ a_{n-2} = b_{n-2} \ and \cdots and \ a_1 = b_1 \ and \ a_0 > b_0$

And for $A$ to be less than $B$ the conditions are:

$a_{n-1} < b_{n-1}$

$Or, a_{n-1} = b_{n-1} \ and \ a_{n-2} < b_{n-2}$

$\vdots$

$Or, a_{n-1} = b_{n-1} \ and \ a_{n-2} = b_{n-2} \ and \cdots and \ a_1 = b_1 \ and \ a_0 < b_0$

Alternatively if we already compute $EQ$ and $GT$ then $LT$ can be computed as

$$LT = \boldsymbol{NOT}\,(GT\,\boldsymbol{OR}\,EQ)$$

We can use the equation of $EQ, GT$ and $LT$ to implement these conditions.

# CHAPTER 3. REVIEW OF PTREE TECHNOLOGY

The predicate tree or pTree is a vertical data technology which records the truth value of a given predicate on a given data set. For each value of the data set it stores 1 if the given predicate is true and stores 0 if the predicate is false. Thus with these 1's and 0's we get a bit vector which forms the leaf of the tree. Then we group the bits of the leaf with a fixed number of bits. Then we apply some predicate to every group and represent them by 1 or 0 (based on the truth value of the predicate) as the parent node of that group. Then we group the bits of all the bits of the parent nodes same way to form upper level parent nodes. This process continues until we form a single bit root node. Then we examine the tree to see if all the bits in a group is all 1's (called pure-1 node) or all 0's (called pure-0 node). In that case we remove all its child nodes. Consider the example in figure 7 (a) where temperature of a city is recorded for 8 days and a predicate $P$ chosen as "the weather is freezing" (i.e. below 32). So for first four data value $P$ is true, for next one it is false and so on. Thus we get a bit vector "1110100" which becomes the leaf of the pTree as shown in figure 7 (b). Figure 7 (b) shows the grouping of the leaf nodes with two bits and generation of the parent nodes. Figure 7 (c) shows the final pTree.

## 3.1. Important definitions

**Definition 3** (Level-0 pTree)**.** *In the process of creating a pTree, the bit slice that is created by representing the truth value of a predicate by 1 or 0 is known as* ***level-0 pTree****. Number of bits (0 or 1) in a level-0 pTree is known as the* ***length*** *of pTree.*

A level-0 pTree is sometime mentioned by only pTree. The root of pTree in figure 7 (b) is an example of a level-0 pTree.

| Temp | $P$ |
|---|---|
| 25 | 1 |
| 20 | 1 |
| 15 | 1 |
| 28 | 1 |
| 35 | 0 |
| 30 | 1 |
| 37 | 0 |
| 40 | 0 |

(a)

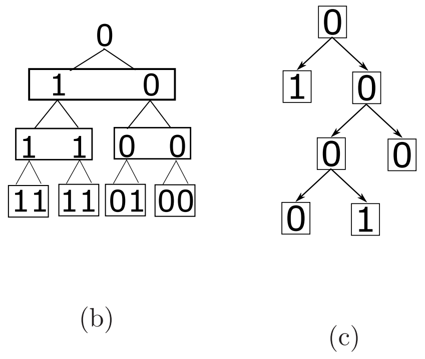

**Fig. 7.** Constructing pTree - (a) Data set (b) Grouping of bits (c) Final pTree $P$.

**Definition 4** (Muli-Level pTree). *When the bit slice of a level-0 pTree is grouped together with a fixed number of bits and the branches of the tree is formed then this pTree is called Multi-Level pTree. The fixed number of bits is called the stride of the pTree.*

Multi-level pTree can be used to compress a large data size. But much of the time compressing into a tree is unnecessary because the uncompressed bit arrays can be processed very quickly.

**Definition 5** (pTree set). *A* ***pTree set*** *$P$ of size $N$ is a collection of $N$ pTrees where each pTree in the collection is accessed by $P[i]$ where $0 \leq i \leq (N-1)$. Number of pTrees in a pTree set is known as the* ***size*** *of the pTree set.*

In figure 8 (b) $P_1, P_2$ and $P_3$ are example three of pTree sets. Size of each pTree set is 3.

## 3.2. pTree representation of data set

The task of generating pTree typically starts by converting a relational table of horizontal records to a set of vertical bit vectors by decomposing each attribute in the table into separate bit slices. If an attribute has numeric value we convert the data into binary then we consider the predicates to be "$2^0$ position is 1", "$2^1$ position is 1", "$2^2$ position is 1" and so on. Then each bit position of the binary value generates one bit slice. But if an attribute has categorical value then first we need to create a bitmap for the attribute and then generate bit vectors for each category. Such vertical partitioning guarantees that the information is not lost.

Consider a data set $\mathcal{D}$ with $n$ attributes as $\mathcal{D} = (A_1, A_2, \ldots, A_n)$. In order to represent it by pTree we will require a set of $n$ pTree sets as $\{P_1, P_2, \ldots, P_n\}$ such that attribute $A_i$ will be represented by pTree set $P_i$. Suppose each value of $A_i$ is prepresented by an $N$-bit binary number $a_{i,N-1}, a_{i,N-2} \ldots, a_{i,j}, \ldots, a_{i,0}$. Then pTree $P_i[j]$ will represent the bit slice of $a_{i,j}$. $P_i[0]$ pTree which represent the Least significant bit slice of $A_i$ is called the lest significant pTree (**LSP** in short) of pTree set $P_i$. Similarly $P_i[N-1]$ is known as the most significant pTree or **MSP**. Figure 8 shows the representation of data set into pTree. In this figure we see that the dataset had three attributes. So we need three pTree sets to represent them. Looking at the value of the attribute we see that we need 3-bit numbers to convert them to binary.

| $A_1$ | $A_2$ | $A_3$ |
|---|---|---|
| 5 | 3 | 4 |
| 3 | 1 | 6 |
| 2 | 5 | 2 |
| 6 | 7 | 0 |
| 7 | 4 | 5 |
| 4 | 5 | 3 |
| 6 | 2 | 6 |
| 4 | 1 | 2 |

(a)

| $P_1$ | | | $P_2$ | | | $P_3$ | | |
|---|---|---|---|---|---|---|---|---|
| $P_1[2]$ | $P_1[1]$ | $P_1[0]$ | $P_2[2]$ | $P_2[1]$ | $P_2[0]$ | $P_3[2]$ | $P_3[1]$ | $P_3[0]$ |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

(b)

**Fig. 8.** pTree representation - (a) Data set of 3 attributes (b) Corresponding pTree sets.

## 3.3. Operations on pTrees

Any Boolean operation discussed in section 2 can be applied on pTrees. Suppose $p$ and $q$ are two level-0 pTrees of length $L$. Assume a binary operator $\mathbf{OP_b}$ that we apply on these two pTrees. Then $p$ $\mathbf{OP_b}$ $q$ is calculated as $p[i]$ $\mathbf{OP_b}$ $q[i]$ $\forall i | i \ni (0 : L - 1)$ where $p[i]$ and $q[i]$ are the $i^{th}$ bit of $p$ and $q$ pTrees. Similarly if $\mathbf{OP_u}$ is a unary operator then $\mathbf{OP_u(p)}$ is calculated as $\mathbf{OP_u}(p[i])$.

So we can have the following binary operations on pTrees:

- AND
- OR
- XOR
- XNOR

And the following unary operation:

- NOT

Suppose $p = 10110110$ and $q = 11010010$ then the figure 9 shows the results of different pTree operations.

| $p$ | $q$ | $p\,\boldsymbol{AND}\,q$ | $p\,\boldsymbol{OR}\,q$ | $p\,\boldsymbol{XOR}\,q$ | $p\,\boldsymbol{XNOR}\,q$ | $\boldsymbol{NOT}\,(p)$ | $\boldsymbol{NOT}\,(q)$ |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

**Fig. 9.** Truth table showing different pTree operations.

Another important operation on pTree is **1Count** operation which returns the number of 1 in a 0-level pTree. So for the above example:

$$1Count(p) = 5$$

## 3.4. Advantage of pTree

As we mentioned before pTree is a data mining ready data structure which represent the data in a loss less manner. In addition to that when data is loaded into the memory it takes less or equal amount of space. When implementing different algorithm using pTree it requires less time. In this section we will show how pTree can be used to get space saving and speed gain.

### 3.4.1. Space advantage of pTree

Assume a dataset $S$ consisting of $N$ rows and $n$ columns containing value of $m$ bits. So if we convert the dataset into pTrees we will get $mn$ pTrees where the length of each tree will be $N$ bits.

In traditional approach, let the size of the dataset be $S_{trad}$. The size of each value will be $= \lceil \frac{m}{8} \rceil$ bytes and size of each column $= \lceil \frac{m}{8} \rceil N$ bytes which gives us the size of the whole dataset, $S_{trad} = \left\lceil \frac{m}{8} \right\rceil nN$ bytes.

In pTree approach, let the size of the size of the dataset be $S_{pTree}$. Size of each pTree will be $\left\lceil \frac{N}{8} \right\rceil$ bytes. So the size of the whole dataset, $S_{pTree} = mn \left\lceil \frac{N}{8} \right\rceil$ bytes.

In a large dataset where $N >> 8$, we can assume $N = 8 \times L$. We get $S_{trad} = 8 \left\lceil \frac{m}{8} \right\rceil nL$ bytes and $S_{pTree} = mnL$ bytes.

Now if $1 < m < 8$ then $\left\lceil \frac{m}{8} \right\rceil = 1$. Therefore, we get $S_{pTree} = \frac{m}{8} S_{trad}$. However when $m = 8$ we get $S_{pTree} = S_{trad}$

Now if $9 < m < 16$ then $\left\lceil \frac{m}{8} \right\rceil = 2$ and we get $S_{pTree} = \frac{m}{16} S_{trad}$. And when $m = 16$ we get $S_{pTree} = S_{trad}$.

So we conclude that $S_{pTree} < S_{trad}$ and $S_{pTree} = S_{trad}$ if $m$ is a multiple of 8.

### 3.4.2. Speed advantage of pTree

Assume that a logical operation (AND, OR, NOT, XOR) between two machine words (of size $W$) of memory takes $T_{log}$ units of time. When we do such a logical operations on two pTrees of length $N$ we actually do it on $L$ pairs of machine words where $L = \left\lceil \frac{N}{W} \right\rceil$. So the logical operation on two pTrees takes $LT_{log}$ unit of times. Assume an arithmetic operation (addition, subtraction, multiplication etc.) between two bytes of memory takes $T_{arith}$ units of time. Suppose we will do such an arithmetic operation on two columns of our previously described data set, $S$. For simplicity assume each value in the dataset takes 1 byte of memory. So each column has $N$ bytes memory and the arithmetic operation will take $T_{trad} = NT_{arith}$ units of time. Suppose to get the same arithmetic operation using pTree we need to perform $g$ number of logical operation on different pTrees. So the process will take $T_{pTree} = gLT_{log}$ unit of time. For simplicity consider that $N$ is multiple of W, so we can write

$$\frac{T_{pTree}}{T_{trad}} = \frac{g}{W} \frac{T_{log}}{T_{arith}}$$

Lets call the factor $\frac{T_{log}}{T_{arith}} = \alpha$. In old processors bit-wise logical operation would run faster than arithmetic operation like addition, multiplication, etc. resulting the factor $\alpha$ less than 1. However in modern day processors the arithmetic operations are optimized in such a level that they run as fast as bit-wise operations resulting the factor $\alpha$ close to 1 [3]. Again $W$ represent the number of bits in a machine word which the computers can handle at a time which is expected to be a large number comparing with $g$. As a result as long as $g < W$ the $T_{pTree}$ will be less than $T_{trad}$ giving the speed gain of pTree based algorithm processing over traditional processing of the same algorithm.

## 3.5. Steps for pTree processing

To obtain a pTree based solution to a data mining problem we might need to go through the following steps.

1. First thing we need to do is to convert the data set into pTree sets. This is a one time process and we then use the pTree sets again and again. If the data set is in traditional form i.e. in horizontal form, with one single scan of the data set we can convert the entire data set into pTree sets. Sometimes it may be possible to get the data directly in pTree form when it is being captured by hardware devices if we have the required hardware support. For example when taking image data using camera different color bands might be converted into pTrees directly from the device. It is also recommended that we calculate all the one count of the pTrees and store them along with the pTrees. Also it is proved to be beneficial if we calculate the complement of each pTree and store them so that we would use them when we would need them.

2. Next step would be to implement a pTree version of the desired algorithm. In next chapter we will discuss many basic algorithms that we can use to convert any algorithm into its pTree version

3. At the end of the step 2, we will get our data mining task producing some results. The result might be in pTree or in a pTree set. In that case we might need to translate the result into a more presentable form.

# CHAPTER 4. PTREE BASED ALGORITHMS

As we mentioned in section 3.5 we need to implement pTree version of a data mining algorithm in order to perform our desired data mining task, we need the basic algorithms designed to perform using pTrees. In this section we will discuss various types of algorithms that are already available and that we have developed in this research work. All these algorithms will use the operations mentioned in section 3.3.

## 4.1. Existing Algorithms

Over the years many basic algorithms have been developed using pTree such as those found in [14]. The idea behind pTree processing is that it will always do operation on entire pTrees and will never loop within a pTree. Here we describe some of them very briefly.

- Sum/Mean: Using this algorithm we can find the summation of an attribute of a data set represented by a pTree set. Then we can find the mean value of that attribute.
- Square Sum/Variance: We can also find the summation of the squared value of the attribute and then we can calculate the variance.
- Median/Rank-k: We can also find the Rank-k value of an attribute and then find the median.
- Max/Min: We find the maximum value and minimum value of an attribute.

## 4.2. 2's complement

From section 2.1 we know that 2's complement is used to represent the negative equivalent of an integer. Also *lemma* 1 shows that we can calculate 2's complement of a binary number by adding 1 with 1's complement. We can derive an easy way to calculate 1's complement of a binary number from definition 1. We know $2^n - 1$ is represented by $n$

1's. Then if we subtract any binary number $N$ from it, the result is just the flip of the bits. That is, the 1's become 0's and 0's become 1's as 1-1=0 and 1-0=1. And according to the definition it the 1's complement of $N$. The algorithm in figure 10 will exploit this idea to convert an integer into its 2's complement.

```
Algorithm 1: Compute 2's complement of a pTree set
Input:
pTree Set $S$ of $N$ pTrees
Output:
pTree Set $S$ in its 2's complement
Variables:
pTree $c, t$
integer $i$

ComppTreeSet($S$):
1.    $c \leftarrow 1$
2.    foreach $i$ in $(0 : N-1)$ do
3.      $t \leftarrow \textbf{NOT}\,(s[i])$
4.      $s[i] \leftarrow t\,\textbf{XOR}\,c$
5.      $c \leftarrow t\,\textbf{AND}\,c$
6.    endfor
```

**Fig. 10.** Algorithm to compute 2's complement of a pTree set

In line 2 we run a loop to start from the LSP to MSP of pTree set $S$. Each time the loop is run line 3 makes the 1's complement of $s[i]$. Then line 4 and 5 add $c$ with $s[i]$. As $c$ is initialized to 1, line 4 and 5 is basically adding 1 with the 1's complement of $s[i]$ and thus getting the 2's complement of $S$.

## 4.3. Absolute value

When we subtract an integer from another the result can be positive or negative depending on the value of the integers. We discussed in section 2.4 the negative results are shown in 2's complement form. In order to get the value of a negative result we need to get the absolute value of that negative result. This is done by using the *lemma* 2 discussed in section 2.1. Algorithm in figure 11 first examine the right most pTree of the input pTree set which is considered as sign pTree. When the value of that pTree is 0 the number represented in row of pTree set is positive and otherwise negative. When it is a positive value nothing is done as there is no difference in terms of value between the absolute value and the number itself. However when it is 1 it indicates that the number is negative and then number is complemented to get the absolute value.

```
Algorithm 2: Compute the absolute value of a pTree set
Input:
pTree Set $A$ of $N+1$ pTrees
Output:
pTree Set $S$ of $N$ pTrees
Variables:
pTree $c, t$
integer $i$

AbspTreeSet($A, S$):
1.    $c \leftarrow a[N]$
2.    foreach $i$ in $(0 : N-1)$ do
3.       $t \leftarrow a[i]\, \boldsymbol{XOR}\, a[N]$
4.       $s[i] \leftarrow t\, \boldsymbol{XOR}\, c$
5.       $c \leftarrow t\, \boldsymbol{AND}\, c$
6.    endfor
```

**Fig. 11.** Algorithm to compute absolute value of a pTree set

In this algorithm we uses a special property of $\boldsymbol{XOR}$ operation. If we observe the behavior of $\boldsymbol{XOR}$ operation shown in figure 2, we see when the operand $Y$ is 0 the value of $X\, \boldsymbol{XOR}\, Y$ is $X$ but when $Y$ is 1 the value of $X\, \boldsymbol{XOR}\, Y$ is $\boldsymbol{NOT}$ (X). Now in line 1 we assign $a[N]$ to $c$ which can be either 0 or 1. In line 3 we are doing $\boldsymbol{XOR}$ between $a[i]$ and $a[N]$. So if $a[N]$ is 1 we get the 1's complement of $a[i]$. If $a[N]$ is 0 then there is no change in $a[i]$. Then line 4 and 5 is adding $c$ with $a[i]$. So basically based on signed pTree we are getting 1'complement of $A$ and adding 1 with it to get the 2's complement of $A$ in $S$ or we are not changing $A$ and adding 0 and thus $S$ is simply equal to $A$. This is how the algorithm is calculating the absolute value of $A$.

## 4.4. Addition

In addition we add two operands and store the result in a third variable. We have two possibilities here, we may add two pTree sets and store the result in another pTree set or we may add a constant value with a pTree set and store the result in another pTree set. In both cases we will exploit the idea discussed in section 2.3. Figure 12 shows the addition of two pTree sets while 13 shows addition of a constant with a pTree set.

In both algorithms we consider unequal pTree set size or bit width by a condition like $N > M$. The summation is stored in the pTree set $S$ which has $N + 1$ pTrees. Line 2 demonstrate the looping through LSP to MSP of pTree set $A$ which is the largest pTree set in size. While adding two pTree sets we deals with two pTree sets to calculate sum and carry (line 4 and 5) up to $M^{th}$ pTree. For the rest of the pTrees we add only the carry $c$ (in line 7 and 8). Finally the carry $c$ is assigned to $S[N]$ in line 11. To add a constant with a pTree set we go through the similar steps except that we do not have a second pTree set, rather we have a constant value consists of 1 and 0. So we check if the constant bit is 1 or 0 and accordingly we adjust the calculation of sum and carry.

**Algorithm 3:** Computing the addition of two pTree sets

```
Input:
pTree Set $A$ of $N$ pTrees
pTree Set $B$ of $M$ pTrees
    where $N > M$
Output:
pTree Set $S$ of $N{+}1$ pTrees
Variables:
pTree $c$
integer $i$

AddpTreeSet($A, B, S$):
1.     $c \leftarrow 0$
2.     foreach $i$ in $(0 : N - 1)$ do
3.        if $i > M$
4.           $S[i] \leftarrow A[i]\,\textbf{XOR}\,B[i]\,\textbf{XOR}\,c$
5.           $c \leftarrow (A[i]\,\textbf{AND}\,B[i])\,\textbf{OR}\,(A[i]\,\textbf{XOR}\,B[i])\,\textbf{AND}\,c$
6.        else
7.           $S[i] \leftarrow A[i]\,\textbf{XOR}\,c$
8.           $c \leftarrow A[i]\,\textbf{AND}\,c$
9.        endif
10.    endfor
11.    $S[N] \leftarrow c$
```

**Fig. 12.** Algorithm to add two pTree sets

```
Algorithm 4: Computing the addition of a pTree set with a constant
Input:
pTree Set $A$ of $N$ pTrees
integer $V$ of $M$ bits
    where $N > M$
Output:
pTree Set $S$ of $N+1$ pTrees
Variables:
pTree $c$
integer $i$

AddpTreeSet($A, V, S$):
1.    $c \leftarrow 0$
2.    foreach $i$ in $(0 : N - 1)$ do
3.      if $i > M$ And $V[i] = 1$
4.        $S[i] \leftarrow \textbf{\textit{NOT}}\,(A[i])\,\textbf{\textit{XOR}}\,c$
5.        $c \leftarrow A[i]\,\textbf{\textit{OR}}\,c$
6.      else
7.        $S[i] \leftarrow A[i]\,\textbf{\textit{XOR}}\,c$
8.        $c \leftarrow A[i]\,\textbf{\textit{AND}}\,c$
9.      endif
10.   endfor
11.   $S[N] \leftarrow c$
```

**Fig. 13.** Algorithm to add a constant with a pTree set

## 4.5. Subtraction

Subtraction is very much similar to addition with few differences in the equation used. Section 2.4 discussed the equations that are used in the algorithms in figure 14 and 15. Figure 14 shows the subtraction of one pTree set from another pTree set while figure 15 shows the subtraction of a constant from a pTree set.

Both the versions of subtraction work as their similar addition algorithms. In both algorithms we consider unequal pTree set size or bit width by a condition like $N > M$. The subtraction result is stored in the pTree set $S$ which has $N+1$ pTrees. The result will be a signed integer in 2's complement form. That is the $S[N]$ pTree is a sign pTree (similar to a sign bit in a signed integer). A 0 value in this pTree indicates the row in that $S$ is a positive integer and a 1 indicate the same as a negative integer.

**Algorithm 5:** Subtraction of a pTree set from another pTree set

```
Input:
pTree Set $A$ of $N$ pTrees
pTree Set $B$ of $M$ pTrees, where $N > M$
Output:
pTree Set $S$ of $N+1$ pTrees
Variables:
pTree $c$
integer $i$

SubpTreeSet($A, B, S$):
1.   $c \leftarrow 0$
2.   foreach $i$ in $(0 : N-1)$ do
3.     if $i > M$
4.       $S[i] \leftarrow \textbf{\textit{NOT}}\,(A[i])\,\textbf{\textit{XOR}}\,B[i]\,\textbf{\textit{XOR}}\,c$
5.       $c \leftarrow (\textbf{\textit{NOT}}\,(A[i])\,\textbf{\textit{AND}}\,B[i])\,\textbf{\textit{OR}}\,(\textbf{\textit{NOT}}\,(A[i])\,\textbf{\textit{XOR}}\,B[i])\,\textbf{\textit{AND}}\,c$
6.     else
7.       $S[i] \leftarrow \textbf{\textit{NOT}}\,(A[i])\,\textbf{\textit{XOR}}\,c$
8.       $c \leftarrow \textbf{\textit{NOT}}\,(A[i])\,\textbf{\textit{AND}}\,c$
9.     endif
10.  endfor
11.  $S[N] \leftarrow c$
```

**Fig. 14.** Algorithm to subtract a pTree set from another pTree set

```
Algorithm 6: Subtraction of a constant from a pTree set
Input:
pTree Set $A$ of $N$ pTrees
integer $V$ of $M$ bits, where $N > M$
Output:
pTree Set $S$ of $N$+1 pTrees
Variables:
pTree $c$
integer $i$

SubpTreeSet($A, V, S$):
1.    $c \leftarrow 0$
2.    foreach $i$ in $(0 : N-1)$ do
3.      if $i > M$ And $V[i] = 1$
4.        $S[i] \leftarrow A[i]\,\boldsymbol{XOR}\,c$
5.        $c \leftarrow \boldsymbol{NOT}\,(A[i])\,\boldsymbol{OR}\,c$
6.      else
7.        $S[i] \leftarrow \boldsymbol{NOT}\,(A[i])\,\boldsymbol{XOR}\,c$
8.        $c \leftarrow \boldsymbol{NOT}\,(A[i])\,\boldsymbol{AND}\,c$
9.      endif
10.   endfor
11.   $S[N] \leftarrow c$
```

**Fig. 15.** Algorithm to subtract a constant from a pTree set

## 4.6. Multiplication

Like addition and subtraction we can multiply a pTree set by another pTree set or by a constant. The algorithms use the procedure discussed in section 2.5. Figure 16 shows the algorithm to multiply a pTree set another pTree set and figure 17 shows the algorithm to multiply a pTree set by a constant.

In the first multiplication algorithm (figure 16) we multiply pTree set $A$ of size $N$ by pTree set $B$ of size $M$ and the result is stored in pTree set $S$ of size $M + N$. In line 1 the algorithm loops through all the pTrees of $B$ from LSP to MSP. In each iteration it loops

```
Algorithm 7: Computing the multiplication of two pTree sets
Input:
pTree Set $A$ of $N$ pTrees
pTree Set $B$ of $M$ pTrees
Output:
pTree Set $S$ of $M$+$N$ pTrees
Variables:
pTree $c, p, q$
integer $i, j$

MultiplypTreeSet($A, B, S$):
1.    foreach $j$ in $(0 : M-1)$ do
2.      $c \leftarrow 0$
3.      foreach $i$ in $(j : N+j-1)$ do
4.        $p \leftarrow A[i-j]\, \boldsymbol{AND}\, B[j]$
5.        $q \leftarrow S[i]$
6.        $S[i] \leftarrow p\, \boldsymbol{XOR}\, q\, \boldsymbol{XOR}\, c$
7.        $c \leftarrow (p\, \boldsymbol{AND}\, q)\, \boldsymbol{OR}\, (p\, \boldsymbol{XOR}\, q)\, \boldsymbol{AND}\, c$
8.      endfor
9.      $S[N+j] \leftarrow c$
10.   endfor
```

**Fig. 16.** Algorithm to multiply two pTree sets

through the pTrees of $A$ (as in line 3), multiply the pTrees of $A$ by the a single pTree of $B$ namely $B[j]$ (line 4). Then add this product with $S$ (line 5, 6 and 7). In next iteration the algorithm multiply each pTree of $A$ by the next pTree of $B$. Then it adds this product with $S$ but shifting one place to the left. The shifting is done by the assignment of variable $i$ in loop in line 3.

In the second algorithm (figure 17) it uses the same technique as the first algorithm with one exception that now it is multiplying a pTree set $A$ by a value $V$ where bits of $V$ might be 1 or 0. So the algorithm checks if $V$ is 1 or 0 in line 3. If it is 0 then nothing is done (only shifts the $S$) as $A[i] \times 0 = 0$. When it is 1 then the algorithm needs to add only

**Algorithm 8:** Computing the multiplication of a pTree set by a constant

**Input:**
pTree Set $A$ of $N$ pTrees
integer $V$ of $M$ bits
**Output:**
pTree Set $S$ of $M$+$N$ pTrees
**Variables:**
pTree $c, q$
integer $i, j$

```
MultiplypTreeSet($A, V, S$):
1.   foreach $j$ in $(0 : M - 1)$ do
2.     $c \leftarrow 0$
3.     if $V[j] = 1$
4.       foreach $i$ in $(j : N + j - 1)$ do
5.         $q \leftarrow S[i]$
6.         $S[i] \leftarrow A[i - j]\ \boldsymbol{XOR}\ q\ \boldsymbol{XOR}\ c$
7.         $c \leftarrow (A[i - j]\ \boldsymbol{AND}\ q)\ \boldsymbol{OR}\ (A[i - j]\ \boldsymbol{XOR}\ q)\ \boldsymbol{AND}\ c$
8.       endfor
9.     endif
10.    $S[N + j] \leftarrow c$
11.  endfor
```

**Fig. 17.** Algorithm to multiply a pTree set by a constant

pTree of $A$ as $A[i] \times 1 = A[i]$. So line 5, 6 and 7 are modified accordingly.

## 4.7. Comparison

Section 2.6 shows the basic equations of comparing two integers. Using these equations we can compare two pTree sets as shown in the algorithm in firgure 18.

This algorithm compare two pTree sets $A$ and $B$ each of them has $N$ pTrees. It begins with initializing three pTrees $gt$, $eq$ and $lt$. Here $gt$ is used to show which values in $A$ is greater than $B$. Similarly $eq$ to show equality and $lt$ for less than comparison. In line 4 and 5 it uses the equations of section 2.6. In line 5 the algorithm uses ***XNOR*** operation to find where $A$ and $B$ are equal. It is due to the fact that when two operands are equal then ***XNOR*** gives us 1 and otherwise 0 as shown in the truth table in figure 3. Finally line 7 calculate the value of $lt$ from $gt$ and $eq$.

```
Algorithm 9: Compare two pTree sets
Input:
pTree Set $A$ of $N$ pTrees
pTree Set $B$ of $N$ pTrees
Output:
pTree $eq,gt,lt$
Variables:
integer $i$

ComparepTreeSet($A, B, eq, gt, lt$):
1.    $gt \leftarrow 0$
2.    $eq \leftarrow 1$
3.    foreach $i$ in $(N-1:0)$ do
4.        $gt \leftarrow gt\,\boldsymbol{OR}\,a[i]\,\boldsymbol{AND}\,\boldsymbol{NOT}\,(b[i])\,\boldsymbol{AND}\,eq$
5.        $eq \leftarrow (a[i]\,\boldsymbol{XNOR}\,b[i])\,\boldsymbol{AND}\,eq$
6.    endfor
7.    $lt \leftarrow \boldsymbol{NOT}\,(gt\,\boldsymbol{OR}\,eq)$
```

**Fig. 18.** Algorithm to compare two pTree sets

## 4.8. $L1$ Distance

Assume two data point $X$ and $Y$ in $n$ dimensional space where $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_n)$. $L_1$ distance between two data point is defined as:

$$d_{L_1} = \sum_{i=1}^{n} |x_i - y_i| \tag{4.1}$$

Suppose a data set $\mathcal{S}$ has n attributes. An attribute $A_i$ is represented by a pTree set $P_i$ containing $N$ pTrees. That is $\mathcal{S} = \{P_1, P_2, \ldots, P_n\}$. Assume we need to find the $L1$ distance of all the points of the data set from a fixed point $C = (c_1, c_2, \ldots, c_n)$. Suppose the distance will be calculated in pTree set $D$. This will be done by the following steps for all the values of $i$ from 1 to $n$.

**Step 1:** Subtract $c_i$ from pTree set $P_i$ using the algorithm in figure 15. Assume the resultant pTree set is $R_i$.

**Step 2:** Get the absolute value pTree of $R_i$ using the algorithm in figure 11. Assume the absolute values are stored in pTree set $S_i$

**Step 3:** Add $S_i$ with pTree set $D$ using the algorithm in figure 12

Algorithm in figure 19 shows these steps.

```
Algorithm 10: Calculate the $L_1$ distance
Input:
Data Set $\mathcal{S}$ of $n$ pTrees sets $\{P_1, P_2, \ldots, P_n\}$ where each $P_i$ contains $N$ pTrees
A fixed point $C = (c_1, c_2, \ldots, c_n)$ where each $c_i$ is an $N$-bit value
Output:
pTree set $D$ contains $(N + n)$ pTrees
Variables:
integer $i$

L1Distance($\mathcal{S}, C, D$):
1.    foreach $i$ in $(1 : n)$ do
2.       $SubpTreeSet(P_i, c_i, R_i)$
3.       $AbspTreeSet(R_i, S_i)$
4.       $AddpTreeSet(D, S_i, D)$
5.    endfor
```

**Fig. 19.** Algorithm to calculate $L_1$ distance

## 4.9. Squared Euclidean Distance

Euclidean distance (also known as $L_2$ distance) between two data point $X$ and $Y$ in $n$ dimensional space where $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_n)$ is defined as:

$$d_{L_2} = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{4.2}$$

Square Euclidean Distance (SED) is the square of $L_2$ distance. So we can write:

$$SED = \sum_{i=1}^{n} (x_i - y_i)^2 \tag{4.3}$$

Using the same data set $\mathcal{S}$ and a fixed point $C = (c_1, c_2, \ldots, c_n)$ as discussed in section 4.8 we can find the SED of the data points in $\mathcal{S}$ from $C$ by the following steps for all the values of $i$ from 1 to $n$.

**Step 1:** Subtract $c_i$ from pTree set $P_i$ using the algorithm in figure 15. Assume the resultant pTree set is $R_i$.

**Step 2:** Get the absolute value pTree of $R_i$ using the algorithm in figure 11. Assume the absolute value pTree is $T_i$

**Step 3:** Get the squared value of $T_i$ by multiplying it with itself using the algorithm in figure 16. Assume the absolute value pTree is $S_i$

**Step 4:** Add $S_i$ with pTree set $D$ using the algorithm in figure 12

Algorithm in figure 20 shows these steps.

**Algorithm 11:** Calculate the SED

```
Input:
Data Set $\mathcal{S}$ of $n$ pTrees sets $\{P_1, P_2, \ldots, P_n\}$ where each $P_i$ contains $N$ pTrees
A fixed point $C = (c_1, c_2, \ldots, c_n)$ where each $c_i$ is an $N$-bit value
Output:
pTree set $D$ contains $(2N + n)$ pTrees
Variables:
integer $i$

SED($\mathcal{S}, C, D$):
1.    foreach $i$ in $(1 : n)$ do
2.        $SubpTreeSet(P_i, c_i, R_i)$
3.        $AbspTreeSet(R_i, T_i)$
4.        $MultiplypTreeSet(T_i, T_i, S_i)$
5.        $AddpTreeSet(D, S_i, D)$
6.    endfor
```

**Fig. 20.** Algorithm to calculate SED

## 4.10. Dot production calculation

For various data mining algorithm we often need to find the dot product of a set of points with a unit vector to find the shadow lengths of the points along the direction of that unit vector. Assume the data set $\mathcal{S}$ as mentioned in section 4.8 represented by $n$ pTree sets $\{P_1, P_2, \ldots, P_n\}$. Consider a direction vector $\mathcal{D} = (D_1, D_2, \ldots, D_n)$. We are interested to find the Dot product $DP$ as

$$DP = \mathcal{S} \bullet \mathcal{D}$$

$$DP = \sum_{i=1}^{n} P_i * D_i \tag{4.4}$$

We can calculate $DP$ by following the steps for $i$ from 1 to $n$

**Step 1:** Get the product of $P_i$ and $D_i$ using the algorithm in figure 16. Assume the absolute value pTree is $S_i$

**Step 2:** Add $S_i$ with pTree set $DP$ using the algorithm in figure 12

Algorithm in figure 21 shows how to calculate dot product:

## 4.11. Approximate calculation

We know in big data processing it is very vital to be able to finish its processing in reasonable time. That is why speed gain is very important. In section 3.4.2 we saw that the speed gain of pTree processing depends number of basic pTree operations performed. Now if the values in a data set are very large then each value would take a large number of bits which will make the size of the pTree set large. This large size of pTree set will cause a large number of pTree operations to perform that would eventually limit the speed gain in pTree processing. To overcome this situation we can perform some operations in approximation where an absolute accuracy is not necessary. For example, in FAUST algorithm [36] we find the dot product of the data values with a unit vector and find the range of the values of that

```
Algorithm 12: Calculate the dot product
Input:
Data Set $\mathcal{S}$ of $n$ pTrees sets $\{P_1, P_2, \dots, P_n\}$ where each $P_i$ contains $N$ pTrees
A direction vector $\mathcal{D} = (D_1, D_2, \dots, D_n)$ where each $D_i$ is an $N$-bit value
Output:
pTree set $DP$ contains $(2N + n)$ pTrees
Variables:
integer $i$

DotProduct($\mathcal{S}, \mathcal{S}, DP$):
1.    foreach $i$ in $(1 : n)$ do
2.        $MultiplypTreeSet(P_i, D_i, S_i)$
3.        $AddpTreeSet(DP, S_i, DP)$
4.    endfor
```

**Fig. 21.** Algorithm to calculate dot product

dot product. In this case we might not need to find the exact dot product values rather we can approximately calculate the dot products.

One approach to approximately perform different operations is to take a small number of most significant bits of both the operands and make the rest of the bits either 0 or 1. So in pTree set a small number of most significant pTrees will take part in the operations and rest of the pTrees will be considered either 0 or 1. Here we explain how it will work. As we know multiplication operation involve more pTree operations than addition or subtraction operation this approximation is more siginificant in case of multiplication. So we will explain how it will work for multiplication.

Suppose we have two integers $A$ and $B$. We divide $A$ into $A_x$ and $A_y$ where $A_x$ is the number formed by taking most significant $x$ bits and $A_y$ is the number formed by taking least significant $y$ bits. Similarly we divide $B$ into $B_x$ and $B_y$. So we can write

$$A = 2^y A_x + A_y \tag{4.5}$$

and

$$B = 2^y B_x + B_y \tag{4.6}$$

Now we may consider both $A_y$ and $B_y$ consist of all 0 bits which will make

$$A = 2^y A_x \tag{4.7}$$

and

$$B = 2^y B_x \tag{4.8}$$

So

$$A \times B = 2^{2y} A_x B_x \tag{4.9}$$

Now consider $A_y$ consists of all 1 bits and $B_y$ consist of all 0 bits. For $y$ number of 1's in $A_y$ makes it $2^y - 1$ and we discard -1 from here for simplicity and that makes approximate value of

$$A = 2^y (A_x + 1) \tag{4.10}$$

So

$$A \times B = 2^{2y}(A_x B_x + B_x) \tag{4.11}$$

Now consider both $A_y$ and $B_y$ consist of all 1 bits. That gives us the approximate value of

$$A = 2^y (A_x + 1) \tag{4.12}$$

and

$$B = 2^y(B_x + 1) \tag{4.13}$$

So

$$A \times B = 2^{2y}(A_xB_x + A_x + B_x + 1) \tag{4.14}$$

In pTree processing we consider $A$ and $B$ to be pTree sets. So $A_x$ and $B_x$ are pTree sets taking the most significant $x$ pTrees from $A$ and $B$ respectively. Then to implement the equations 4.9, 4.11 and 4.14 we need to multiply $A_xB_x$ by using algorithm to in figure 16. Then we need to shift this result $2y$ place left to implement 4.9. For equation 4.11 we need to add $B_x$ with $A_xB_x$ using the addition algorithm in figure 12 and then shift the result $2y$ place left. For equation 4.14 first we need to calculate $A_x+B_x+1$. To do that we can modify the addition algorithm to add two pTree sets with an initial carry 1. So in the algorithm we just need to assign $c$ to 1 in line 1. Then we need to add this result with $A_xB_x$ and finally shift the result $2y$ place left.

# CHAPTER 5. EXPERIMENTAL DESIGN AND ANALYSIS

In order to prove the effectiveness of our methods over traditional horizontal methods we ran a series of experiment. In this chapter we describe the experimental framework that we used to analyze our methods. First we propose a comprehensive experimental framework based on formal design methodology. Then based on those frameworks the experiments are carried out. Next we compare the results in a conclusive manner.

## 5.1. Formal design methodology

In formal design, an experiment is a test or series of tests where we purposefully change the input variables of a process or a system so that we can see the changes in the output responses and also be able to identify the reason for the changes that we observed [26, 29]. In this section we introduce some experimental design terminologies before we describe our experimental design scheme.

In the experimental design area, the input parameters and different assumptions about the system are called factors. The output performance measurements are called responses. An experiment is carried out in order to determine which factors affect a response. Then the important factors are identified that lead to the best possible response.

### 5.1.1. Choice of input factors and responses

We used factorial design strategy [29] to design our experiments. In factorial design strategy, experiments are run at various values and level of the factors to study theirs effects on the responses. As a result we are able to form a confidence interval for an expected response at each factor level. If the factors are quantitative we may use a graph showing the response as a function of factor levels.

Our purpose here is to explore the performance of our proposed algorithms to perform the mathematical operations using vertical data structure of pTrees on Big data. We have algorithms that perform basic mathematical operations such as addition, subtraction and multiplication. Each of them are of two categories. The algorithms doing these operations on two pTree sets as input are in category one. It includes the algorithm in figures 12, 14 and 16. Second category of algorithms are those that required one pTree set and a constant value such as in figures 13, 15 and 17. We also have algorithms that perform some data mining tasks using the basic mathematical algorithms. They are discussed in figures 19, 20 and 21. Then we have another algorithm that compare two pTree sets as shown in figure 18. Each of these algorithms are compared with their horizontal versions.

The table 1 shows all the algorithm that are considered in our experimental design and assign an ID to each of them.

**Table 1.** List of Algorithms with their assigned ID's

| Algorithm Name | Assigned ID |
|---|---|
| Algorithm to add two pTree sets | Add_P |
| Algorithm to add a constant with a pTree set | Add_C |
| Algorithm to subtract two pTree sets | Sub_P |
| Algorithm to subtract a constant from a pTree set | Sub_C |
| Algorithm to multiply two pTree sets | Mul_P |
| Algorithm to multiply a pTree set by a constant | Mul_C |
| Algorithm to compare two pTree sets | Com_P |
| Algorithm to calculate $L_1$ distance | L1D_P |
| Algorithm to calculate SED | SED_P |
| Algorithm to calculate dot product | DP_P |

These algorithms will be compared with their horizontal versions as listed in the table 2.

In our experiment design we use the following design factors: data size, bit width, bit pattern and different types algorithms. Next we fix the range of the factors. As for the data size we use 10 different data sizes as shown in table 3. Each data size is assigned a unique

**Table 2.** List of horizontal algorithms with their assigned ID's

| Algorithm Name | Assigned ID |
|---|---|
| Horizontal addition algorithm | Add_H |
| Horizontal subtraction algorithm | Sub_H |
| Horizontal multiplication algorithm | Mul_H |
| Horizontal algorithm to compare two attributes | Com_H |
| Horizontal $L_1$ distance algorithm | L1D_H |
| Horizontal SED algorithm | SED_H |
| Horizontal dot product algorithm | DP_H |

ID. Each data value can be of a different bit width. We vary this factor as shown in table 4.

**Table 3.** Data size range with their assigned ID's

| Data Size | Size ID |
|---|---|
| $0.5 \times 10^9$ | S1 |
| $1.0 \times 10^9$ | S2 |
| $1.5 \times 10^9$ | S3 |
| $2.0 \times 10^9$ | S4 |
| $2.5 \times 10^9$ | S5 |
| $3.0 \times 10^9$ | S6 |
| $3.5 \times 10^9$ | S7 |
| $4.0 \times 10^9$ | S8 |
| $4.5 \times 10^9$ | S9 |
| $5.0 \times 10^9$ | S10 |

We also assigned an ID to each bit width. Our third factor is the bit pattern used to form constant values. While dealing with constant value our algorithm differently based on the bit value of the constant. In another word, for 1's and 0's in the binary representation of the constant the algorithms run differently. So there may be an impact in the performance of the algorithm based on number of 1's and 0's. Therefore, we selected three patterns to form the constant as shown in table 5. We then generate all possible combination of the design factors and run our experiments for each of the combination. Each of this run is called a

**Table 4.** Data width range with their assigned ID's

| Bit width | Width ID |
|---|---|
| 4 bits | B1 |
| 8 bits | B2 |
| 12 bits | B3 |
| 16 bits | B4 |
| 20 bits | B5 |
| 24 bits | B6 |
| 28 bits | B7 |
| 32 bits | B8 |

**Table 5.** Bit pattern of the constant with their assigned ID's

| Pattern | Pattern ID |
|---|---|
| Best pattern with only one 1 | Best |
| Worst pattern with all 1 | Worst |
| Average pattern with equal numbers of 1's and 0's | Average |

design point [35].

As for output responses in our experiments we consider two responses. They are run time and scalability to data size. We measure run time in millisecond. We investigate scalability by observing the run time for different data size.

## 5.2. Experiment Design

In this section we will discuss the experiment we designed to compare different algorithm in our design factor. We run these algorithms for each data size S1 through S10 and for each bit width B1 through B8. For the data we used is of uniform distribution and we generate them using C++ library function to generate pseudo random numbers. For each experiment we are showing a partial result in a table. The detail results will be discussed in section 5.3 the analysis section.

### 5.2.1. Experiment 1: Comparing Addition

The algorithms involved in this experiment are Add_P, Add_C and Add_H. For algorithm Add_C we use three bit patterns Best, Worst and Average. So we have 5 different algorithms (Add_P, Add_C with Best, Add_C with Worst, Add_C with Average and Add_H) to execute for 10 data sizes and 8 bit widths. That gives us total of 400 design points. Each design point is executed 10 times and their execution time is measured in milliseconds and average execution time is calculated. Table 6 shows the partial result of the experiment that involved average execution time of these algorithms for data size S1 through S10 with data width B2.

**Table 6.** Partial Result of Experiment 1

| Size | Width | Add_P | Add_C Best | Add_C Average | Add_C Worst | Add_H |
|---|---|---|---|---|---|---|
| S1 | B2 | 84.54 | 61.94 | 62.94 | 64.00 | 1308.78 |
| S2 | B2 | 197.34 | 137.80 | 139.70 | 149.92 | 2572.44 |
| S3 | B2 | 336.62 | 222.64 | 224.32 | 226.96 | 3723.72 |
| S4 | B2 | 486.20 | 331.12 | 331.88 | 330.56 | 4931.70 |
| S5 | B2 | 598.28 | 432.42 | 436.64 | 438.24 | 6148.54 |
| S6 | B2 | 695.76 | 519.64 | 520.16 | 521.08 | 7352.84 |
| S7 | B2 | 821.86 | 604.24 | 618.80 | 628.90 | 8566.42 |
| S8 | B2 | 960.06 | 693.24 | 701.42 | 714.02 | 9778.54 |
| S9 | B2 | 1078.04 | 781.64 | 801.74 | 802.54 | 10986.30 |
| S10 | B2 | 1166.00 | 866.22 | 891.66 | 894.10 | 12212.70 |

### 5.2.2. Experiment 2: Comparing Subtraction

This experiment is designed exactly as experiment shown in subsection 5.2.1 except we use Sub_P, Sub_C and Sub_H instead of Add_P, Add_C and Add_H. So we have 400 design points for this experiment also. Likewise we execute each design point 10 times and calculate their average in millisecond. Table 7 shows the partial result of the experiment that involved average execution time of these algorithms for data size S1 through S10 with data width B2.

**Table 7.** Partial Result of Experiment 2

| Size | Width | Sub_P | Sub_C Best | Sub_C Average | Sub_C Worst | Sub_H |
|---|---|---|---|---|---|---|
| S1 | B2 | 89.78 | 61.42 | 62.38 | 65.12 | 1301.36 |
| S2 | B2 | 207.88 | 133.38 | 134.98 | 135.16 | 2512.10 |
| S3 | B2 | 346.64 | 221.80 | 222.20 | 222.80 | 3717.74 |
| S4 | B2 | 474.66 | 326.20 | 326.38 | 328.76 | 4927.00 |
| S5 | B2 | 647.38 | 420.82 | 424.10 | 427.22 | 6141.54 |
| S6 | B2 | 701.80 | 512.20 | 515.52 | 520.88 | 7351.54 |
| S7 | B2 | 805.22 | 594.16 | 600.64 | 603.98 | 8555.10 |
| S8 | B2 | 963.24 | 682.90 | 683.36 | 693.34 | 9769.84 |
| S9 | B2 | 1101.16 | 773.60 | 774.90 | 778.10 | 10965.50 |
| S10 | B2 | 1156.10 | 855.58 | 857.28 | 865.76 | 12178.00 |

### 5.2.3. Experiment 3: Comparing Multiplication

In this experiment use Mul_P, Mul_C and Mul_H algorithm. Like previous two experiments in subsection 5.2.1 and 5.2.2 we have 400 design points and we execute each design point 10 times and calculate their average in millisecond. Table 8 shows the partial result of the experiment that involved average execution time of these algorithms for data size S1 through S10 with data width B2.

**Table 8.** Partial Result of Experiment 3

| Size | Width | Mul_P | Mul_C Best | Mul_C Average | Mul_C Worst | Mul_H |
|---|---|---|---|---|---|---|
| S1 | B2 | 462.00 | 21.50 | 259.00 | 446.50 | 1427.50 |
| S2 | B2 | 978.00 | 44.00 | 563.50 | 975.00 | 2485.00 |
| S3 | B2 | 1580.50 | 69.50 | 857.50 | 1484.50 | 3623.00 |
| S4 | B2 | 2141.50 | 99.00 | 1156.50 | 1973.00 | 4861.00 |
| S5 | B2 | 2803.50 | 131.50 | 1458.50 | 2513.50 | 6039.00 |
| S6 | B2 | 3456.50 | 170.50 | 1815.50 | 3092.00 | 7302.00 |
| S7 | B2 | 4026.00 | 222.50 | 2167.50 | 3765.50 | 8448.00 |
| S8 | B2 | 4771.50 | 267.50 | 2563.50 | 4500.50 | 9630.00 |
| S9 | B2 | 5340.00 | 314.50 | 2944.50 | 5169.00 | 10788.50 |
| S10 | B2 | 6045.00 | 346.00 | 3323.50 | 5917.50 | 11974.50 |

### 5.2.4. Experiment 4: Calculate $L_1$ distance

In this experiment we calculate the $L_1$ distance of a set of points from a random point. We assume the points to be three dimensional points. That is there are three attributes in the data set. This data set is presented by three pTree sets. Then using the algorithm L1D_P as shown in figure 19 we calculate the $L_1$ distance. Then we calculate the same $L_1$ distance using horizontal algorithm L1D_H. We run these algorithms for each data size S1 through S10 and for each bit width B1 through B8. Thus we get 160 design points which we run 10 times each and get their average. Table 9 shows the partial result of the experiment that involved average execution time of these algorithms for data size S1 through S10 with data width B2.

**Table 9.** Partial Result of Experiment 4

| Size | Width | L1D_P | L1D_H |
|---|---|---|---|
| S1 | B2 | 470.00 | 6437.50 |
| S2 | B2 | 1045.00 | 12457.50 |
| S3 | B2 | 1702.00 | 18571.50 |
| S4 | B2 | 2342.50 | 24800.50 |
| S5 | B2 | 2939.50 | 31031.50 |
| S6 | B2 | 3659.50 | 37011.00 |
| S7 | B2 | 4025.00 | 43172.00 |
| S8 | B2 | 4664.00 | 49486.50 |
| S9 | B2 | 5315.00 | 55574.00 |
| S10 | B2 | 5924.50 | 61811.50 |

### 5.2.5. Experiment 5: Calculate SED

In this experiment we calculate the squared Euclidean distance (SED) of a set of points from a random point. As we assumed in section 5.2.4 the points are three dimensional points so we have three attributes in the data set which is represented by three pTree sets. Then using the algorithm SED_P as shown in figure 20 we calculate SED. Then we calculate the same SED using horizontal algorithm SED_H. We run these algorithms for each data size S1 through S10 and for each bit width B1 through B8. Thus we get 160 design points which we run 10 times each and get their average. Table 10 shows the partial result of the experiment that involved average execution time of these algorithms for data size S1 through S10 with data width B2.

**Table 10.** Partial Result of Experiment 5

| Size | Width | SED_P | SED_H |
|---|---|---|---|
| S1 | B2 | 1960.00 | 13737.50 |
| S2 | B2 | 4328.00 | 27500.00 |
| S3 | B2 | 6908.50 | 41280.50 |
| S4 | B2 | 9321.50 | 54985.50 |
| S5 | B2 | 11766.50 | 68709.00 |
| S6 | B2 | 14303.50 | 82463.00 |
| S7 | B2 | 16922.00 | 96086.50 |
| S8 | B2 | 19579.50 | 109776.00 |
| S9 | B2 | 22825.00 | 123635.00 |
| S10 | B2 | 25416.50 | 137452.00 |

### 5.2.6. Experiment 6: Calculate Dot Product

In this experiment we calculate the dot product (DP) of a set of points on a unit vector. The unit vector is chosen randomly. As we assumed in section 5.2.4 and 5.2.5 the points are three dimensional points so we have three attributes in the data set which is represented by three pTree sets. Then using the algorithm DP_P as shown in figure 21 we calculate DP. Then we calculate the same DP using horizontal algorithm DP_H. We run these algorithms for each data size S1 through S10 and for each bit width B1 through B8. Thus we get 160 design points which we run 10 times each and get their average. Table 11 shows the partial result of the experiment that involved average execution time of these algorithms for data size S1 through S10 with data width B2.

**Table 11.** Partial Result of Experiment 6

| Size | Width | DP_P | DP_H |
|---|---|---|---|
| S1 | B2 | 560.50 | 2734.00 |
| S2 | B2 | 1261.50 | 5453.50 |
| S3 | B2 | 2048.00 | 8293.50 |
| S4 | B2 | 2771.50 | 10920.50 |
| S5 | B2 | 3509.50 | 13733.50 |
| S6 | B2 | 4266.00 | 16307.00 |
| S7 | B2 | 5006.00 | 19072.50 |
| S8 | B2 | 6001.00 | 21744.50 |
| S9 | B2 | 6584.00 | 24414.00 |
| S10 | B2 | 7390.00 | 27318.00 |

### 5.2.7. Experiment 7: Compare two pTree sets

In this experiment we compare two attributes of a data set. The data set is represented in pTree sets and we use the algorithm Com_P (in figure 18)to compare them. Then we compare the attributes using horizontal algorithm Com_H. We run these algorithms for each data size S1 through S10 and for each bit width B1 through B8. Thus we get 160 design points which we run 10 times each and get their average. Table 12 shows the partial result of the experiment that involved average execution time of these algorithms for data size S1 through S10 with data width B2.

## 5.3. Analysis of the experimental results

In this section we will analyze the results of the seven experiments described from section 5.2.1 through 5.2.7. For each of the experiment we represent the results in histograms. For each of the bit width from B1 to B8 we have one histogram. Each histogram shows the size of the data set in x-axis and the response time for each algorithm involving the experiment in y-axis. Then for each experiment we will also show the speed gain (or speed loss) of pTree processing over horizontal processing for each bit width. To measure the speed

**Table 12.** Partial Result of Experiment 7

| Size | Width | Com_P | Com_H |
|---|---|---|---|
| S1 | B2 | 52.00 | 2471.00 |
| S2 | B2 | 111.50 | 4759.50 |
| S3 | B2 | 171.50 | 7015.50 |
| S4 | B2 | 240.50 | 9226.00 |
| S5 | B2 | 303.50 | 11531.50 |
| S6 | B2 | 380.50 | 13841.50 |
| S7 | B2 | 469.00 | 16091.50 |
| S8 | B2 | 539.00 | 18325.50 |
| S9 | B2 | 620.50 | 20682.00 |
| S10 | B2 | 691.50 | 23034.00 |

gain we use the following formula

$$SpeedGain = (1 - \frac{T_P}{T_H})$$

Where $T_P$ is the time taken by algorithm of pTree processing and $T_H$ is the time taken by algorithm of horizontal processing. For example, a speed gain of 80% means a pTree processing would be 80 time units faster than a horizontal processing that takes 100 time units. That is, a pTree processing would take 20 time units whereas the horizontal processing takes 100 time units. A negative value of speed gain actually refers a speed loss in which an algorithm doing pTree processing would take more time than horizontal processing. For example, a speed gain of -10% means a pTree processing would take 110 time units whereas the horizontal processing takes 100 time units. One interesting point to note here is that we did not show the speed gain for different data size because speed gain remains constant regardless the data size.

### 5.3.1. Result of experiment 1

The figure 22 and 23 show the results of experiment 1. From these figures we see that as the data size is increasing execution times of all the algorithms are also increasing linearly. But the slope for algorithm *ADD_H* is steeper than other algorithm. This proves that the pTree based algorithms will always take less time than horizontal algorithm. In table 13 we showed speed gain we have of addition operation for different bit widths. We get maximum speed gain for data width of 4 bits. As the data width increases the speed gain begins to decrease. This confirms our discussion in section 3.4.2 where we showed that speed gain depends on the number of pTree operations we have to perform and number of pTree operation increases as the data width increases. This table also shows that pTree algorithm involving one pTree execute faster than algorithm involving two pTrees.

**Table 13.** Speed gain of pTree based Addition algorithm

| Bit Width | ADD_P | ADD_C Best | ADD_C Average | ADD_C Worst |
|---|---|---|---|---|
| 4 | 95% | 96% | 96% | 96% |
| 8 | 90% | 93% | 93% | 93% |
| 12 | 86% | 90% | 90% | 90% |
| 16 | 81% | 86% | 86% | 86% |
| 20 | 77% | 83% | 83% | 83% |
| 24 | 71% | 80% | 80% | 80% |
| 28 | 68% | 76% | 76% | 76% |
| 32 | 66% | 74% | 74% | 74% |

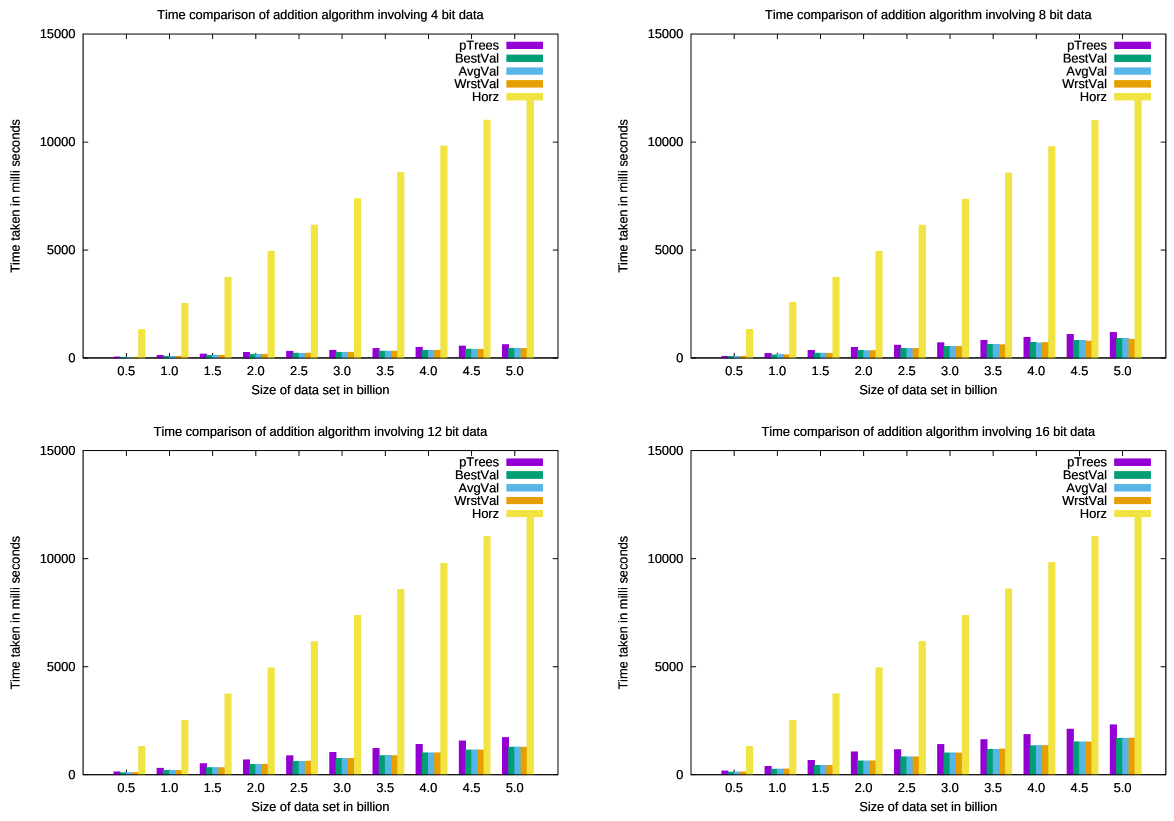

**Fig. 22.** Result of experiment 1 from 4 to 16 bit width

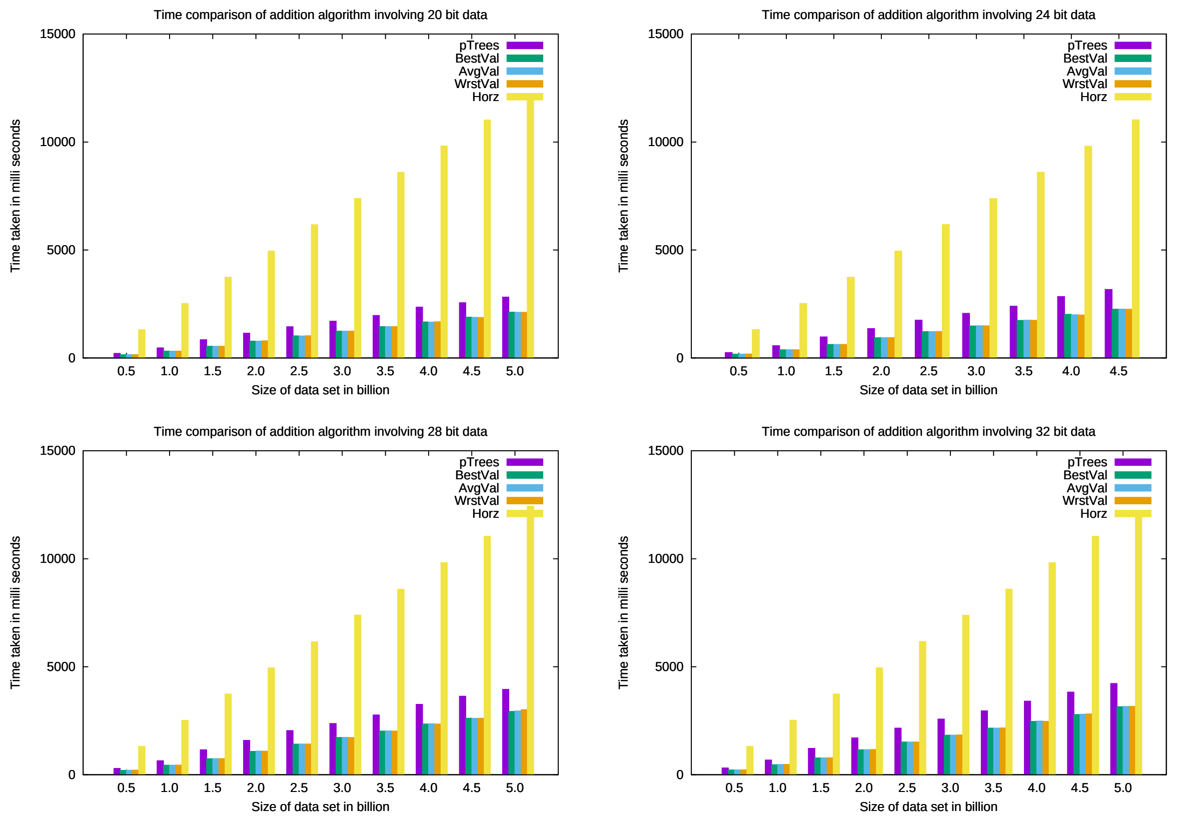

**Fig. 23.** Result of experiment 1 from 20 to 32 bit.

## 5.3.2. Result of experiment 2

The figure 24 and 25 shows the result of experiment 2. Table 14 shows the speed gain of pTree based subtraction algorithms over the horizontal processing. These results are much like the result of addition operation because addition and subtraction has the same number of pTree operations and each operation is similar to the operation of addition.

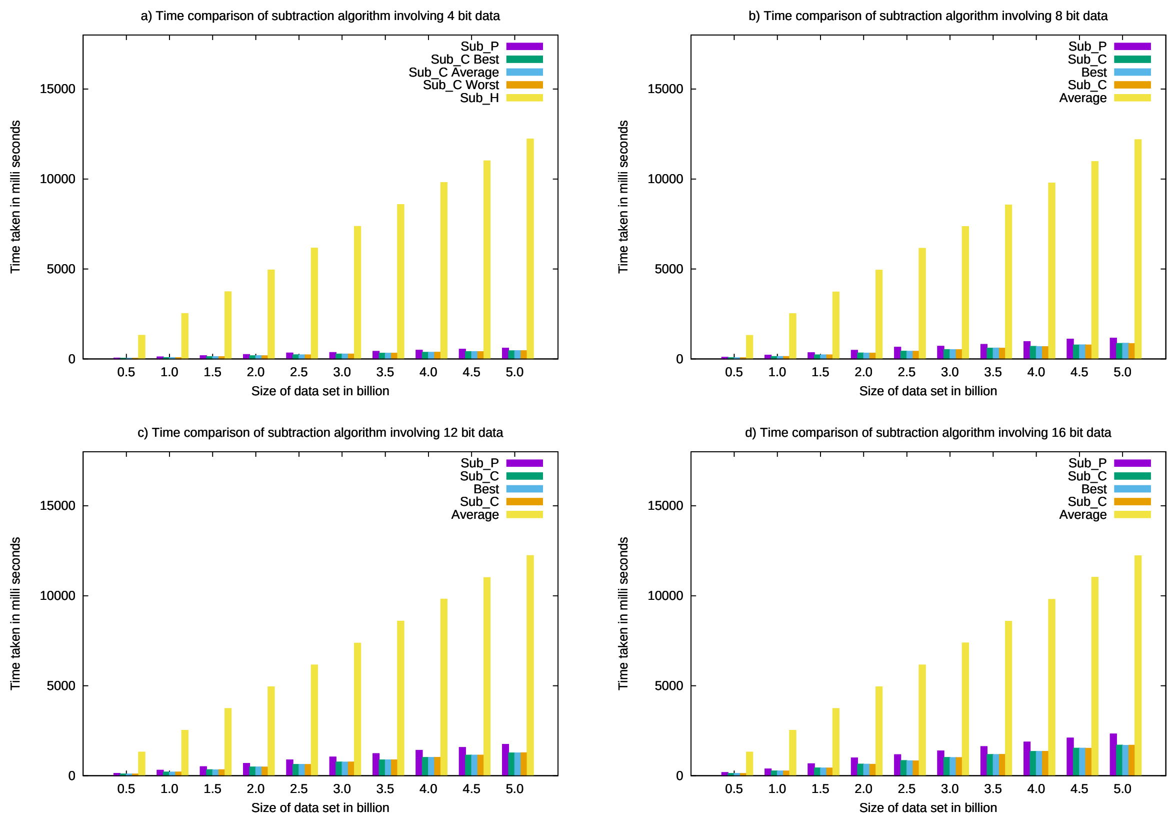

**Fig. 24.** Result of experiment 2 from 4 to 16 bit width

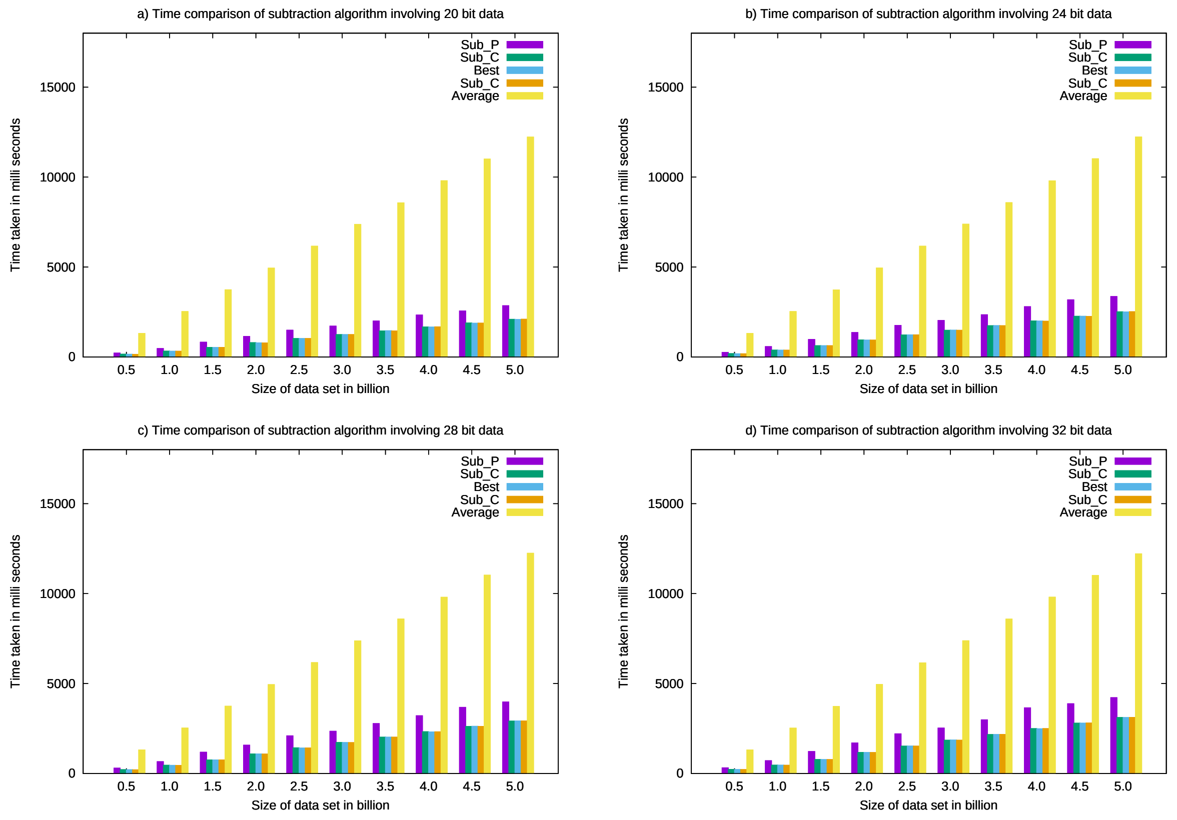

**Fig. 25.** Result of experiment 2 from 20 to 32 bit.

**Table 14.** Speed gain of pTree based Subtraction algorithm

| Bit Width | SUB_P | SUB_C Best | SUB_C Average | SUB_C Worst |
|---|---|---|---|---|
| 4 | 95% | 96% | 96% | 96% |
| 8 | 91% | 93% | 93% | 93% |
| 12 | 86% | 90% | 90% | 90% |
| 16 | 81% | 86% | 86% | 86% |
| 20 | 77% | 83% | 83% | 83% |
| 24 | 71% | 79% | 79% | 80% |
| 28 | 68% | 76% | 76% | 76% |
| 32 | 66% | 75% | 75% | 74% |

### 5.3.3. Result of experiment 3

The figure 26 and 27 shows the result of experiment 3. Table 15 shows the speed gain of pTree based multiplication algorithms. As we can see bit width of up to 12 pTree based all algorithms have positive speed gain. For bit width of 4 and 8 pTree processing is quite faster than horizontal processing. Bit width of 12 has positive gain but not quite as good as 4 and 8. But for bit width of 16 multiplication involving two pTree sets and multiplication of a pTree set with worst combination of constant have negative speed gain. That is in these two cases horizontal processing would be faster. pTree processing with best combination of constant always performs faster than horizontal processing. (Bit widths beyond 20 are not shown in the table because of negative speed gain). Our experimental results on multiplication shows that multiplication becomes an expensive operation when dealing with large bit width. But in many cases we may use approximate calculation of multiplication discussed in section 4.11 to make it faster.

**Table 15.** Speed gain of pTree based Multiplication algorithm

| Bit Width | MUL_P | MUL_C Best | MUL_C Average | MUL_C Worst |
|---|---|---|---|---|
| 4 | 91% | 99% | 95% | 93% |
| 8 | 68% | 98% | 82% | 68% |
| 12 | 24% | 97% | 58% | 22% |
| 16 | -29% | 96% | 30% | -31% |
| 20 | -113% | 95% | -14% | -117% |

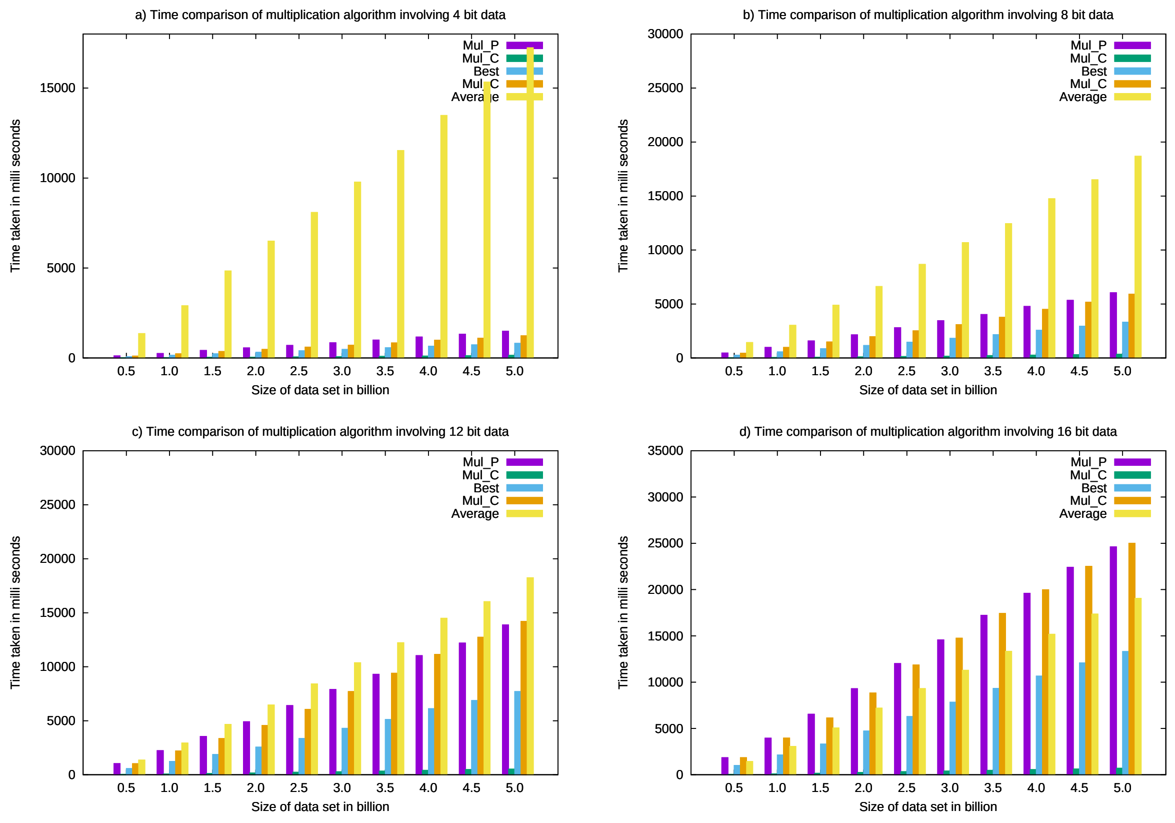

**Fig. 26.** Result of experiment 3 from 4 to 16 bit width

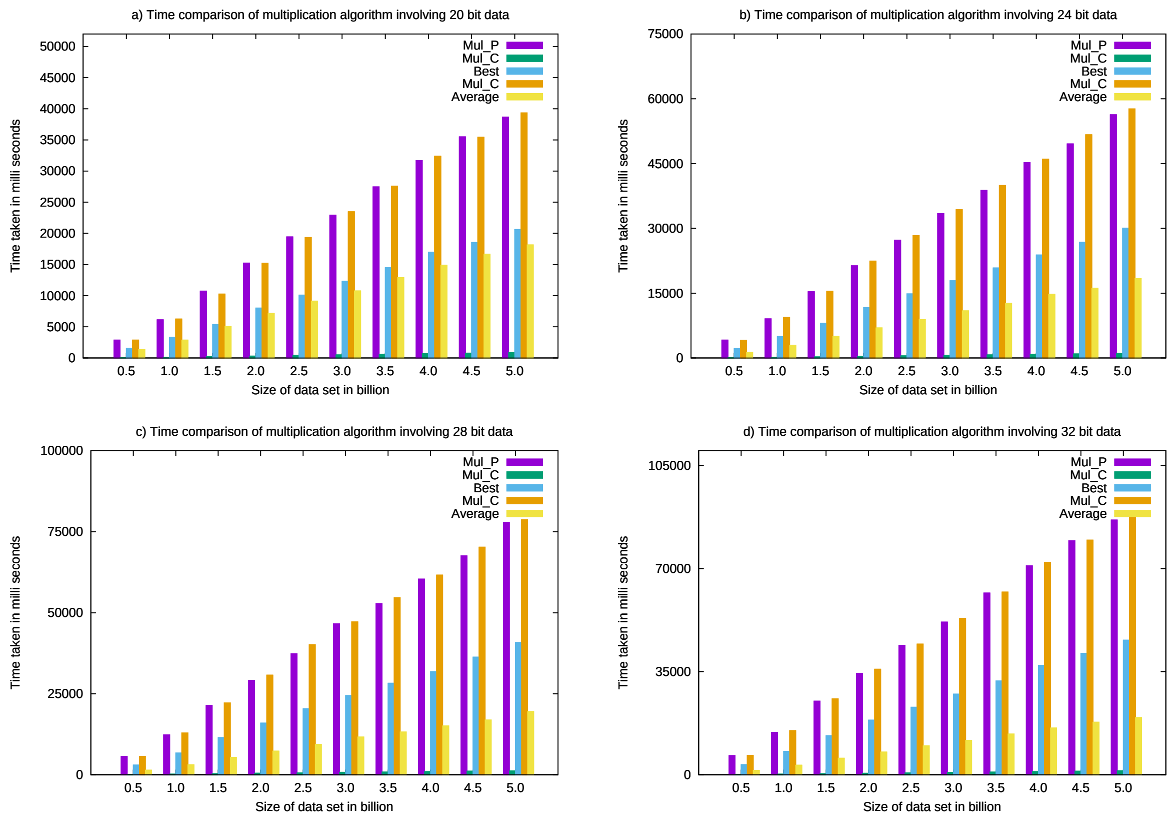

**Fig. 27.** Result of experiment 3 from 20 to 32 bit.

### 5.3.4. Result of experiment 4

The figure 28 and 29 shows the result of experiment 4. Table 16 shows the speed gain of $L_1$ Distance calculation algorithm. The table shows that for all the bit widths the pTree processing has significant speed gain over horizontal algorithm. Maximum speed gain of 95% is seen for bit width of 4 and minimum of 63% is seen for bit width of 32.

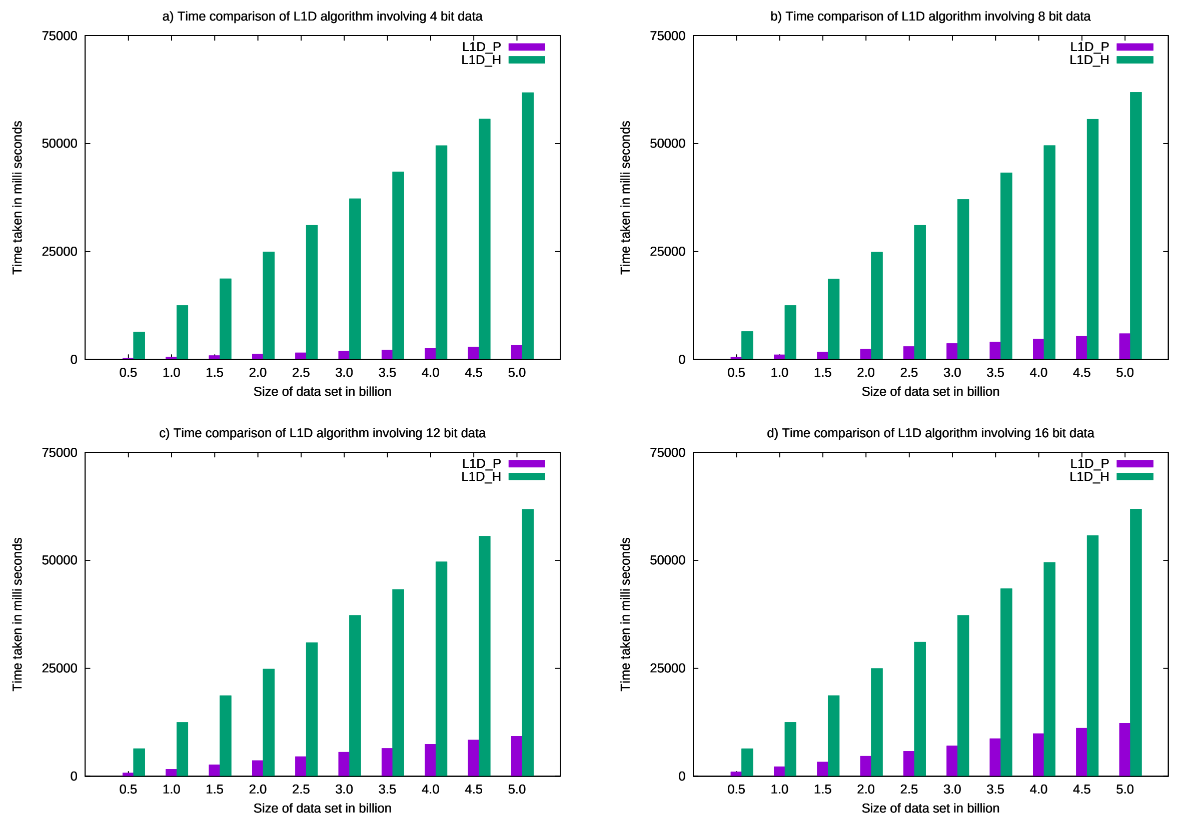

**Fig. 28.** Result of experiment 4 from 4 to 16 bit width

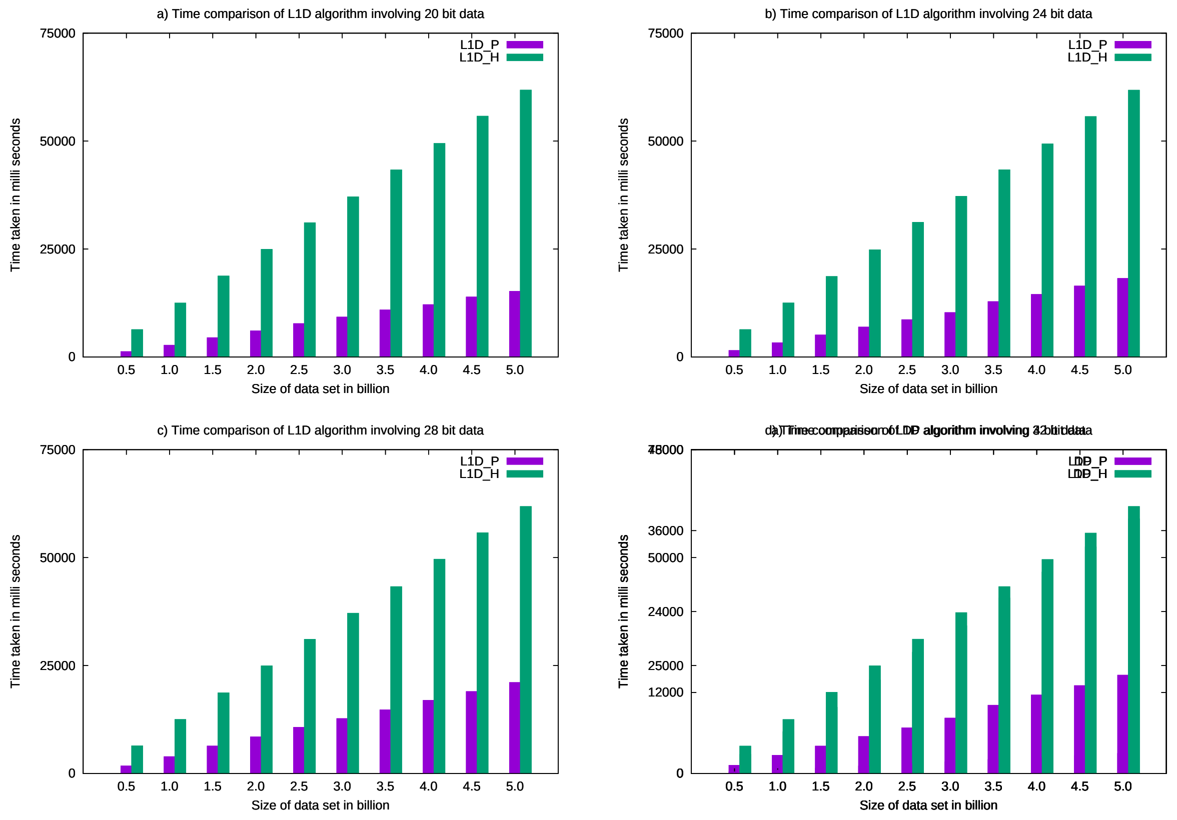

**Fig. 29.** Result of experiment 4 from 20 to 32 bit.

**Table 16.** Speed gain of pTree based $L_1$ Distance calculation algorithm

| Bit Width | L1D_P |
|---|---|
| 4 | 95% |
| 8 | 90% |
| 12 | 85% |
| 16 | 80% |
| 20 | 75% |
| 24 | 71% |
| 28 | 66% |
| 32 | 63% |

### 5.3.5. Result of experiment 5

The figure 30 and 31 shows the result of experiment 5. Table 17 shows the speed gain of SED calculation algorithm. As we can see from the table we get a 96% of speed gain over horizontal processing for bit width of 4. We get 24% of speed gain for bit width of 20. But after that we get negative speed gain of 5% for bit width of 24. So we can say up to bit width of 24 pTree processing is faster than horizontal processing.

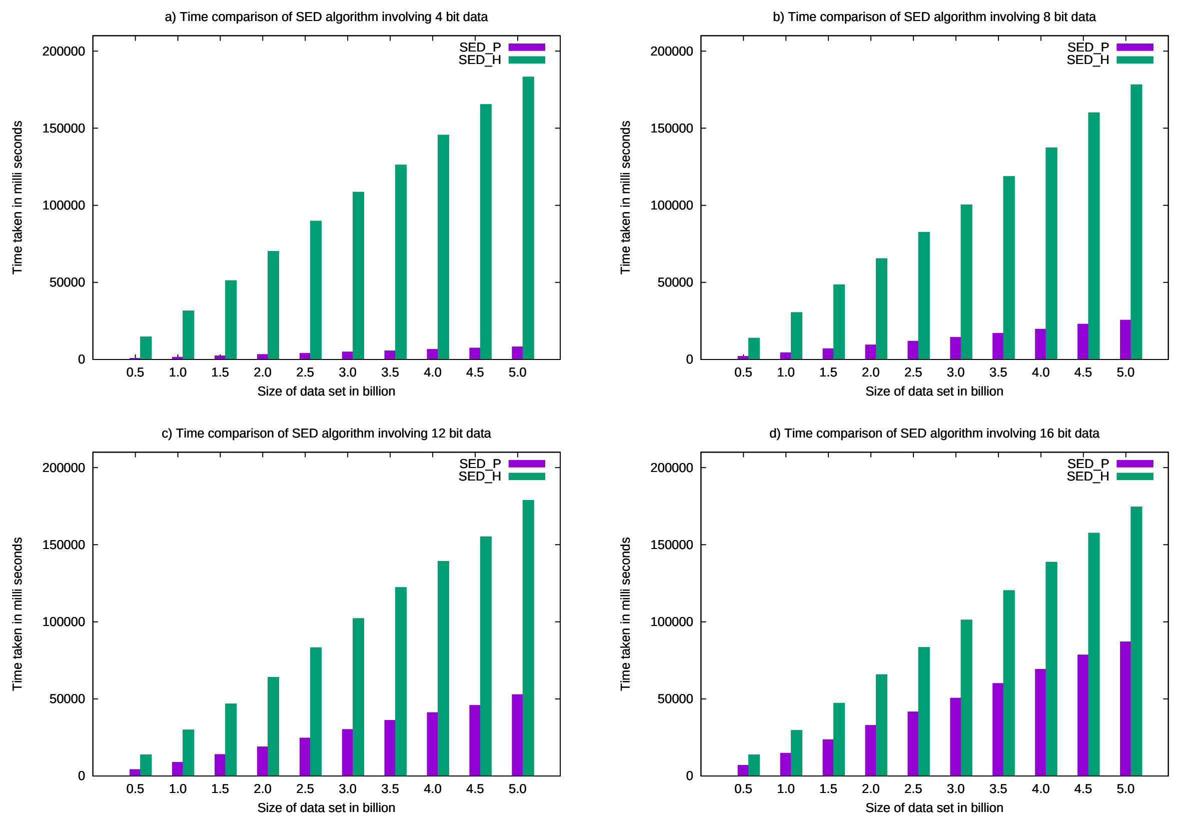

**Fig. 30.** Result of experiment 5 from 4 to 16 bit width

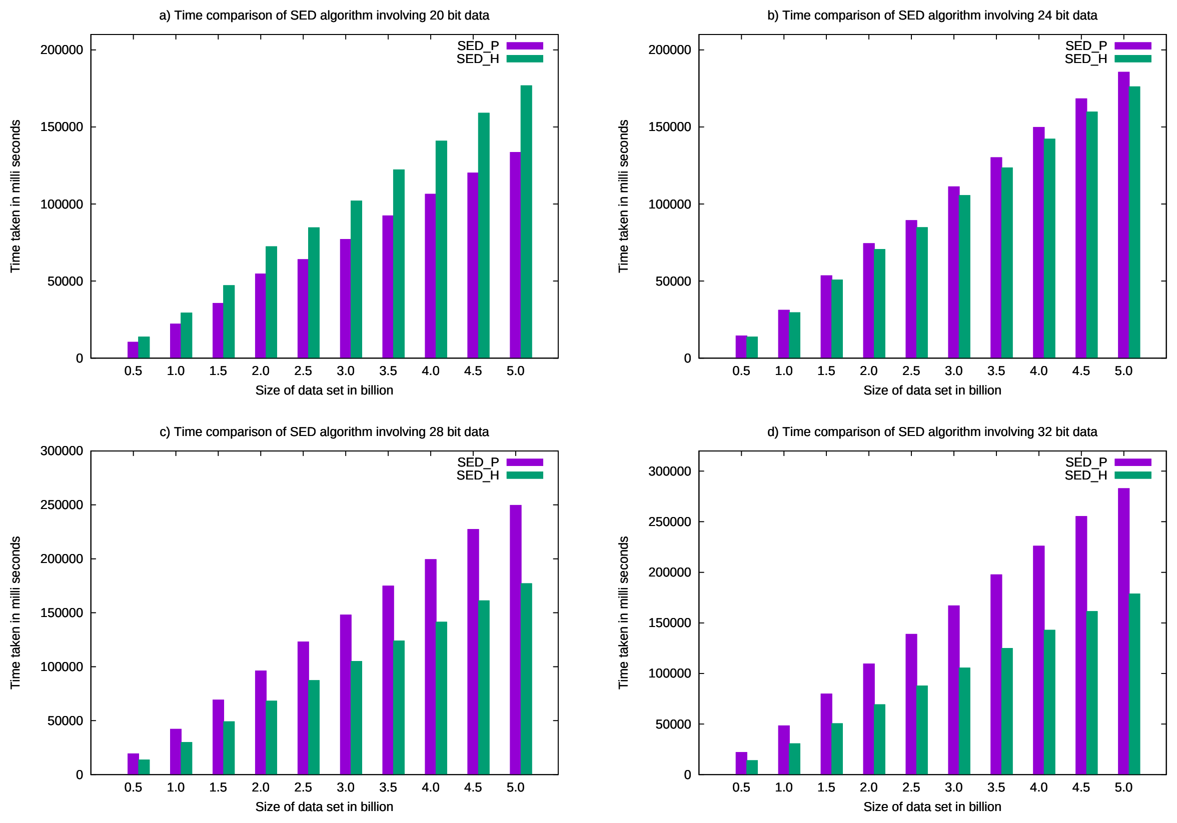

**Fig. 31.** Result of experiment 5 from 20 to 32 bit.

**Table 17.** Speed gain of pTree based SED calculation algorithm

| Bit Width | SED_P |
|---|---|
| 4 | 96% |
| 8 | 86% |
| 12 | 71% |
| 16 | 50% |
| 20 | 24% |
| 24 | -05% |

### 5.3.6. Result of experiment 6

The figure 32 and 33 shows the result of experiment 6. Table 18 shows the speed gain of dot product calculation algorithm. As we can see from the table we get a 92% of speed gain over horizontal processing for bit width of 4. We get 14% of speed gain for bit width of 20. But after that we get negative speed gain of 15% for bit width of 24. So we can say up to bit width of 24 pTree processing is faster than horizontal processing.

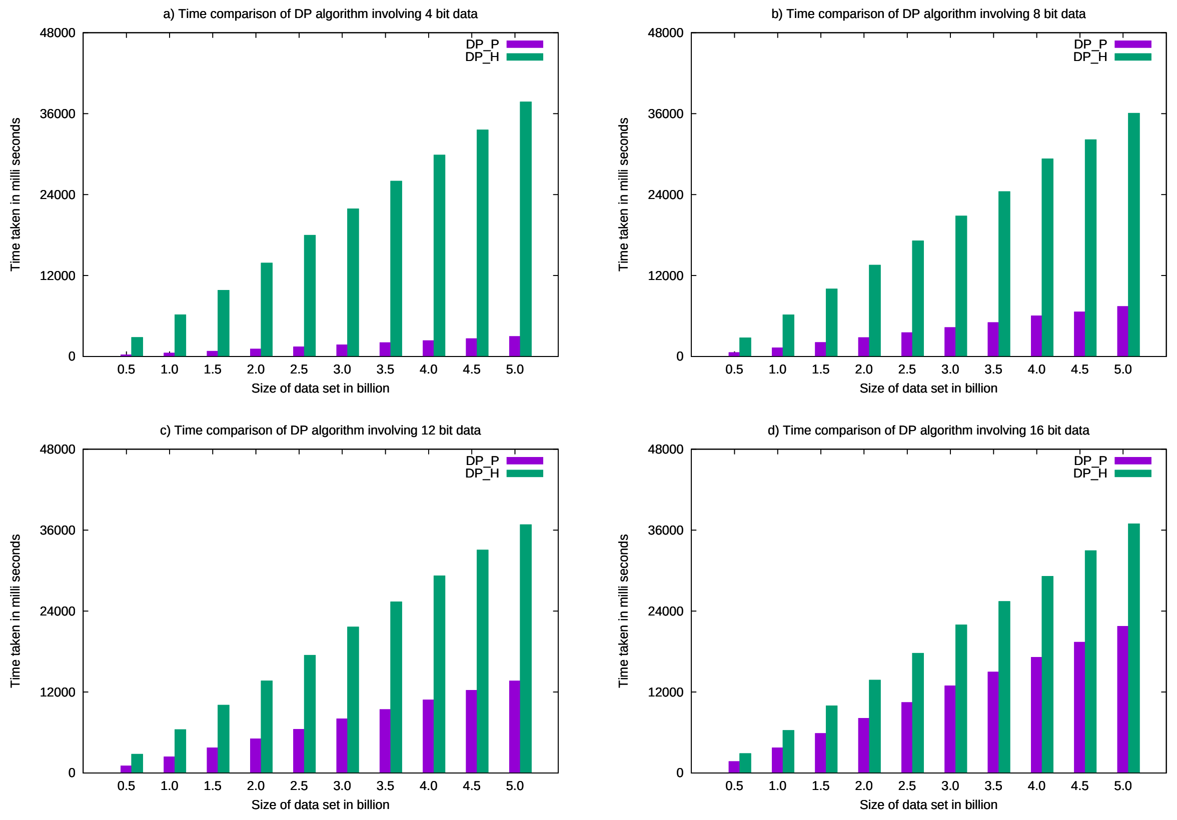

**Fig. 32.** Result of experiment 6 from 4 to 16 bit width

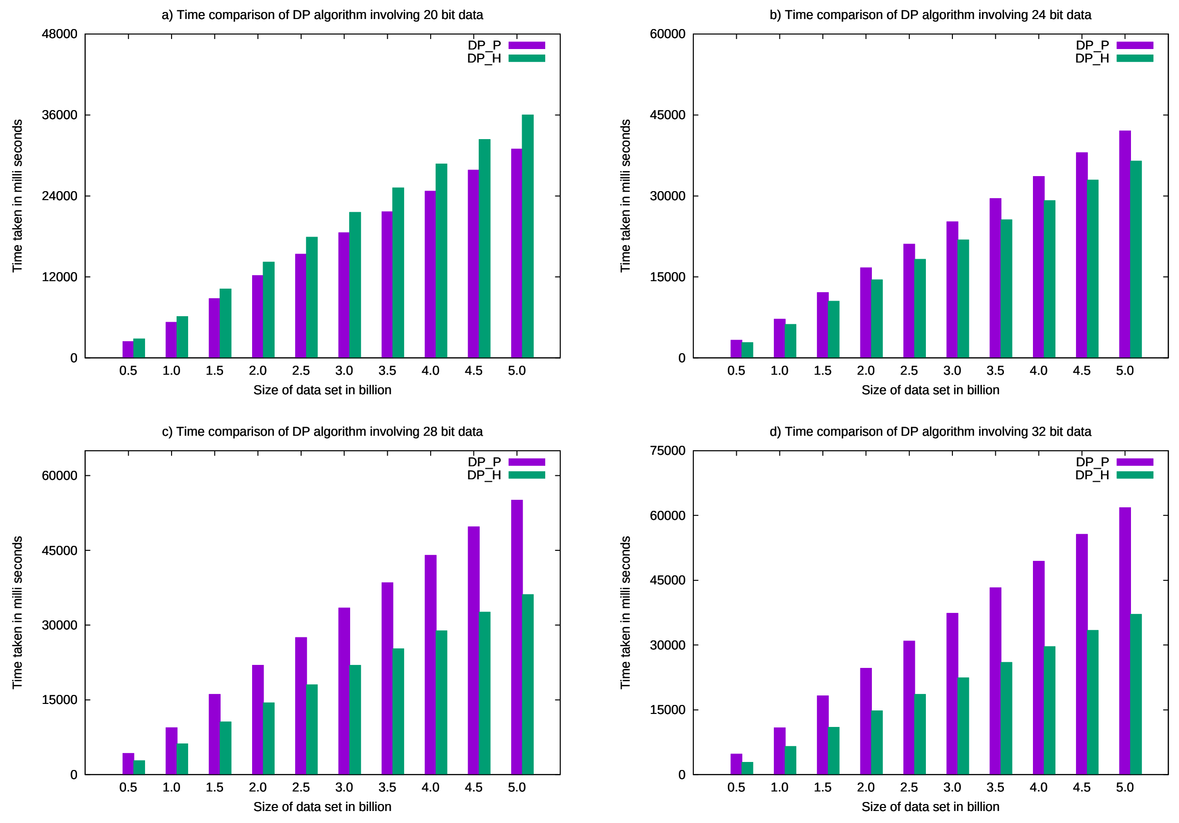

**Fig. 33.** Result of experiment 6 from 20 to 32 bit.

**Table 18.** Speed gain of pTree based Dot Product calculation algorithm

| Bit Width | DP_P |
|---|---|
| 4 | 92% |
| 8 | 79% |
| 12 | 63% |
| 16 | 41% |
| 20 | 14% |
| 24 | -15% |

### 5.3.7. Result of experiment 7

The figure 34 and 35 shows the result of experiment 7. Table 19 shows the speed gain of Comparison algorithm that compares two pTree set which is equivalent to compare two attributes of a data set. The table shows that for all the bit widths the pTree processing has significant speed gain over horizontal algorithm. Maximum speed gain of 99% is seen for bit width of 4 and minimum of 92% is seen for bit width of 32.

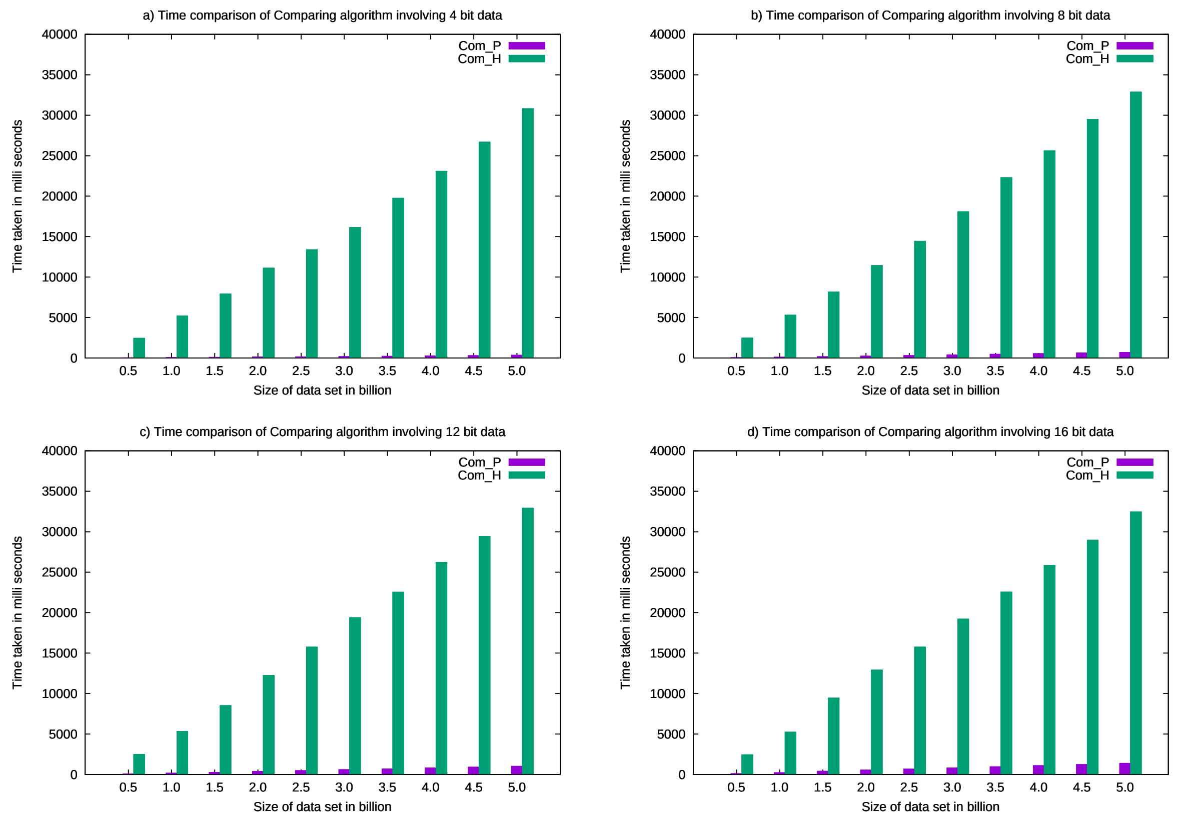

**Fig. 34.** Result of experiment 7 from 4 to 16 bit width

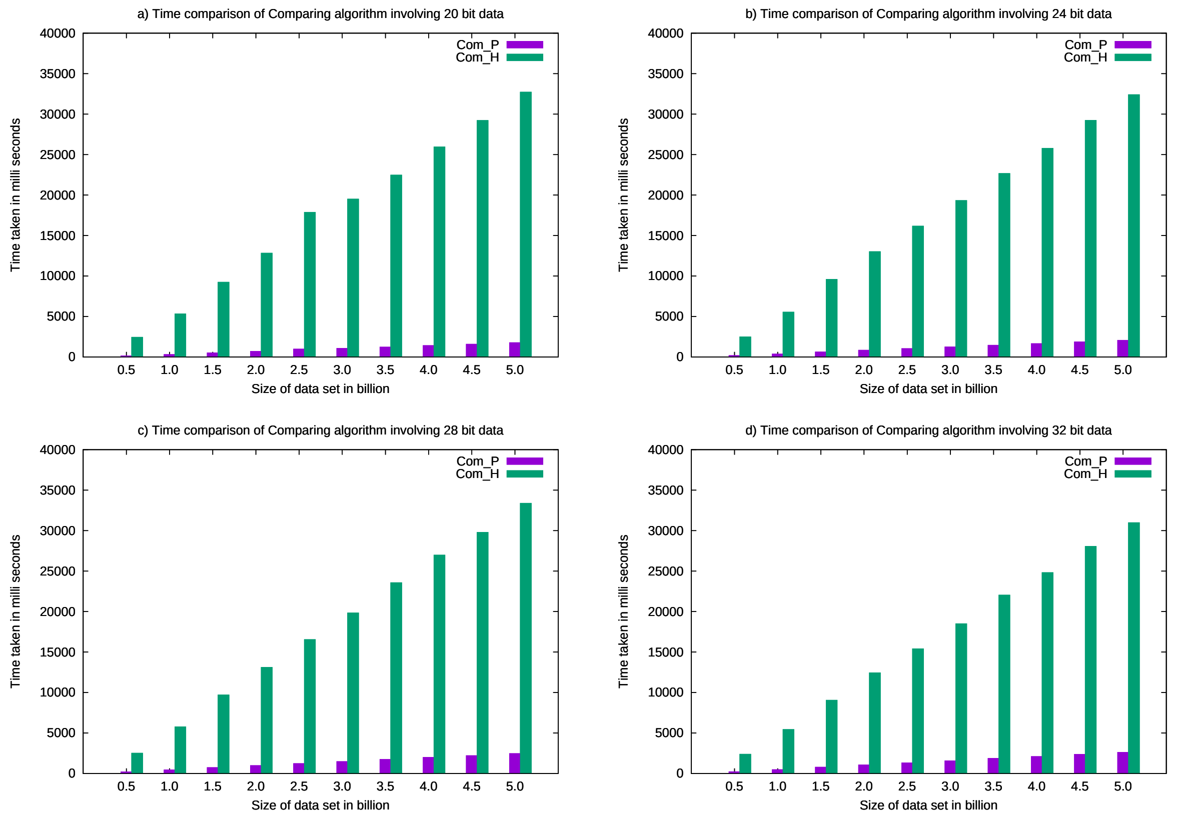

**Fig. 35.** Result of experiment 7 from 20 to 32 bit.

**Table 19.** Speed gain of pTree based Comparison algorithm

| Bit Width | Com_P |
|---|---|
| 4 | 99% |
| 8 | 98% |
| 12 | 97% |
| 16 | 96% |
| 20 | 95% |
| 24 | 94% |
| 28 | 93% |
| 32 | 92% |

## 5.4. Approximate multiplication calculation

As we discussed in section 4.11, we can approximately calculate the multiplication of two pTree sets to make it faster by sacrificing some of the accuracy. In some data mining algorithms such trade off does not impact much, specially when we need the results only to compare each other and take a decision. For example, if we need to compute SED to find the distances of the data points from the centroids to figure out which classes or clusters each point belongs to, we do not need to find the exact distances rather an approximation can serve the purpose. Another fact is that for data values of large bit width, much of the information would be carried out by the higher order bits of the data points and lower order bits will give us more minute details which might not be very significant for making the decision. In this section we will see an example how we can compute multiplication faster using approximation.

Assume we have to multiply two attributes $A$ and $B$ each has data value of 32 bits and 5 billion in size. From our experiment we saw horizontal processing of such multiplication will take 19395 ms time. Now let us divide the attributes in higher order 8 bits ($A_x$ and $B_x$) and lower order 24 bits ($A_y$ and $B_y$). So using the equation of 4.9 we can compute the $A_xB_x$ by multiplying two pTree sets of size 8. From our experiment we know this will take 6045 ms time. So in this approximation we can get 68% speed gain.

If we use the equation 4.11 we will need to add $B_x$ with $A_xB_x$ which is a pTree set of size 16. So this addition will take 2303 ms. So this approximation will take 8348 ms which is a 57% speed gain.

For the equation 4.14 we will need to add $A_x + B_x + 1$ with $A_xB_x$. Now $A_x + B_x + 1$ is an addition of pTree sets of size 8 that takes 1166 ms. So total time will be 9514 ms giving us 51% speed gain.

# CHAPTER 6. SUMMARY AND CONCLUSION

In this dissertation we have successfully implemented few very important mathematical operations to be executed in pTrees. These operations are vital in any algorithm in data mining. We have also shown in our experiments that these operations are scalable in big data environment. As the data size increases the performance of the operations remain steady. We have also studied their performance regarding the bit width of the data value. We found that for small bit width between 4 to 12, all the operations perform significantly well in comparison with the traditional horizontal processing of these operations. The operations perform moderately well for bit width between 12 to 20. For large bit width between 20 to 32, we found that not all the operations perform better than horizontal processing. Next we are summarizing our findings in brief.

- Addition and Subtraction: Addition and subtraction operations perform the same way. They perform extremely fast for data value of small bit width with more than 90% speed gain. For large bit width they also perform significantly faster with speed gain of more than 66%. Addition (or subtraction) of a constant with (or from) a pTree set is faster than that of two pTree sets. Different combinations of constant (best, worst and average) perform the same way for these two operations.
- Multiplication: Multiplication operation involves a lot of basic pTree operations. For small bit width of data value we see a significant speed gain of 68% to 91% speed gain. As the number of bit width (i.e., size of pTree set) increases, number of basic pTree operation increases very much resulting the operation taking more time. So for large bit width pTree processing of multiplication becomes slower than horizontal processing. Again multiplying a pTree set with a constant performs faster than multiplying two

pTree sets. As for different combination of constant, best combination always performs better than any other combinations. In fact this is faster than horizontal processing for all bit width with speed gain over 95%. We also see that average constant always performs faster than worst constant combination.

- Comparison: Comparing two pTree sets using pTree processing is always faster than using horizontal processing for all bit width of the data value. We have seen more than 92% speed gain for this operation.

- $L_1$ distance calculation: Computing the $L_1$ distance mainly requires addition, subtraction operations. There is no multiplication operation involve here. So the pTree processing of this calculation works faster than horizontal processing for all bit width. For small bit width we see more than 90% speed gain whereas for large bit width it is more than 63%.

- SED and DP calculation: Squared Euclidean Distance and Dot product calculation involve multiplication along with addition and subtraction operations. So with the increase of bit width their performance using pTree processing drop as in the case of multiplication. However for small bit width their performance is significantly faster than horizontal processing.

We also showed in section 5.4 that multiplication can be performed approximately for large bit width. Thus we can get faster processing of pTree based calculation sacrificing some of accuracy.

In this dissertation we have a comprehensive study of some new algorithms to perform some very important mathematical operations. Our study includes the background knowledge that are required for complete understanding of the algorithms, the detail description of the steps in the algorithms, detail performance study, etc. Our study shows that the performance of the operations are independent of the data size and are limited by bit width of the data value. However for practical purpose this limitation can be ignored because of the fact that we do not need a large bit width most of the cases and in many cases we can do approximation in our operations and get a reasonable accuracy. This makes our operations a good way for processing big data.

# BIBLIOGRAPHY

[1] Calpont infinidb concepts guide. URL http://www.calpont.com/phocadownload/documentation/2.0.0/CalpontInfiniDBConceptsGuide_20-1.pdf.

[2] Treeminer inc., the vertical data mining company. URL http://www.treeminer.com.

[3] Cmicrotek low-power design blog, August 2015. URL http://http://cmicrotek.com/wordpress_159256135/.

[4] T. Abidin and William Perrizo. Smart-tv: A fast and scalable nearest neighbor based classifier for data mining. In *Proceedings of the 21st Association of Computing Machinery Symposium on Applied Computing*, SAC-06, Dijon, France, April 23-27 2006.

[5] T. Abidin, A. Dong, H. Li, and William Perrizo. Efficient image classification on vertically decomposed data. In *IEEE International Conference on Multimedia Databases and Data Management*, MDDM-06, Atlanta, Georgia, April 8 2006.

[6] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, 22(2):207–216, 1993.

[7] P. Boncz and M. Kersten. Monet: An impressionist sketch of an advanced database system. In *Basque International Workshop on Information Technology*, San Sebastian, Spain, July 1995.

[8] P. Boncz, W. Quak, and M. Kersten. Monet and its geographical extensions: A novel approach to high-performance gis processing. In *International Conference on Extending Database Technology*, volume 1057, 1996.

[9] P. Boncz, T. Grust, M. Keulen, S. Manegold, J. Rittinger, and J. Teubner. Monetdb/xquery: A fast xquery processor powered by a relational engine. *ACM SIGMOD*, pages 479–490, 2006.

[10] Arijit Chatterjee, Mohammad K Hossain, Arjun Roy, and William Perrizo. Relational association rule mining in market basket using the rolodex model with p-tree. In *Proceedings at the ISCA 27th International Conference on Computers and Their Applications*, CATA-2012, Las Vegas, Nevada, USA, March 12-14 2012.

[11] Arijit Chatterjee, Arjun Roy, Mohammad K Hossain, and William Perrizo. Multi-hop closure theorem in rolodex model using ptrees. In *Proceedings of Software Engineering and Data Engineering*, SEDE-2012, Los Angeles, California, USA, June 2012.

[12] Surajit Chaudhuri, Umeshwar Dayal, and Vivek Narasayya. An overview of business intelligence technology. *Communications of the ACM*, 54(8):88–98, 2011.

[13] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mobile Networks and Applications*, 19(2):171–209, 2014.

[14] Yue Cui. Aggregate function computation and iceberg querying in vertical databases. *M.S. Thesis, Department of Computer Science, NDSU*, June 2005. URL http://www.cs.ndsu.nodak.edu/~perrizo/saturday/Yue_cui_master_paper.doc.

[15] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. Efficient big data processing in hadoop mapreduce. *Proceedings of the VLDB Endowment*, 5(12):2014–2015, 2012.

[16] Edd Dumbill. Making sense of big data. *Big Data*, 1(1):1–2, 2013.

[17] Chris Eaton, Dirk Deroos, Tom Deutsch, George Lapis, and Paul Zikopoulos. Understanding big data, April 2012. URL http://public.dhe.ibm.com/common/ssi/ecm/en/iml14296usen/IML14296USEN.pdf.

[18] Hanan Elazhary. Cloud computing for big data. Technical report, MAGNT Research Report, 2014.

[19] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann series in data management systems. Elsevier, 2012. ISBN 9789380931913.

[20] Mohammad K Hossain and William Perrizo. Algorithm for shifting images stored in peano mask trees. *International Journal of Electrical, Electronics and Computer Systems (IJEECS)*, 1(1):2221–7258, March 2011.

[21] Mohammad K Hossain, Arijit Chatterjee, Arjun Roy, and William Perrizo. Calculating the squared euclidean distance for vertical data represented in ptrees. In *Proceedings of Software Engineering and Data Engineering*, SEDE-2012, Los Angeles, California, USA, June 2012.

[22] Mohammad K Hossain, Arjun Roy, Arijit Chatterjee, and William Perrizo. Algorithms to calculate the manhattan (l1) distance for vertical data represented in ptrees. In *Proceedings of the 2012 ISCA 27th International Conference on Computers and Their Applications (CATA-2012)*, CATA-2012, Las Vegas, Nevada, USA, March 2012.

[23] Mohammad Kabir Hossain, Rajibul Alam, Abu Ahmed Sayeem Reaz, and William Perrizo. Bayesian classification for spatial data using p-tree. In *Multitopic Conference, 2004. Proceedings of INMIC 2004. 8th International*, pages 321–327. IEEE, 2004.

[24] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

[25] M. Khan, Q. Ding, and William Perrizo. K-nearest neighbor classification on spatial data stream using ptrees. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*, PAKDD 02, pages 517–528, Taipei, Taiwan, May 2002.

[26] A.M. Law and W.D. Kelton. *Simulation modeling and analysis*. McGraw-Hill series in industrial engineering and management science. McGraw-Hill, 2000. ISBN 9780070592926.

[27] Steve Lohr. The age of big data. *New York Times*, 11, 2012.

[28] M.M. Mano. *Digital Design*. Prentice Hall international editions. Prentice Hall, 2002. ISBN 9781888325171.

[29] D.C. Montgomery. *Design and analysis of experiments*. Wiley, 1976. ISBN 9780471614210.

[30] Kazuyo Narita and Hiroyuki Kitagawa. Outlier detection for transaction databases using association rules. In *Web-Age Information Management, 2008. WAIM'08. The Ninth International Conference on*, pages 373–380. IEEE, 2008.

[31] A. Perera, T. Abidin, M. Serazi, G. Hamer, and William Perrizo. Vertical set squared distance based clustering without prior knowledge of k. In *International Conference on Intelligent and Adaptive Systems and Software Engineering*, IASSE-05, pages 72–77, Toronto, Canada, July 2005.

[32] Gregory Piateski and William Frawley. *Knowledge discovery in databases*. MIT press, 1991.

[33] I. Rahal, D. Ren, and William Perrizo. A scalable vertical model for mining association rules. *Journal of Information and Knowledge Management (JIKM)*, 3(4):317–329, 2004.

[34] I. Rahal, M. Serazi, A. Perera, Q. Ding, F. Pan, D. Ren, W. Wu, and William Perrizo. Datamime. In *Association of Computing Machinery, Management of Data*, ACM SIGMOD 04, Paris, France, June 2004.

[35] D. Ren, B. Wang, and William Perrizo. Rdf: A density-based outlier detection method using vertical data representation. In *Proceedings of the 4th IEEE International Conference on Data Mining*, ICDM-04, pages 503–506, November 2004.

[36] Arjun Roy, Arijit Chatterjee, Mohammad K Hossain, and William Perrizo. Fast attribute-based table clustering using predicate-trees: A vertical data mining approach. *Journal of Computational Methods in Science and Engineering*, 12(1):139–146, 2012.

[37] Arjun Roy, Mohammad K Hossain, Arijit Chatterjee, and William Perrizo. Column-oriented database systems : A comparison study. In *Proceedings at the ISCA 27th International Conference on Computers and Their Applications*, CATA-2012, Las Vegas, Nevada, USA, March 2012.

[38] D. Slezak, J. Wroblewski, V. Eastwood, and P. Synak. Brighthouse: An analytic data warehouse for ad-hoc queries. *VLDB*, pages 1337–1345, 2008.

[39] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Mad-den, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and Stan Zdonik. C-store: A column oriented dbms. *VLDB*, pages 553–564, 2005.

[40] Alexander Strehl, Joydeep Ghosh, and Raymond Mooney. Impact of similarity measures

on web-page clustering. In *Workshop on Artificial Intelligence for Web Search (AAAI 2000)*, pages 58–64, 2000.

[41] Adriano Veloso, Wagner Meira, and Mohammed J Zaki. Lazy associative classification. In *Data Mining, 2006. ICDM'06. Sixth International Conference on*, pages 645–654. IEEE, 2006.

[42] E. Wang, I. Rahal, and William Perrizo. Davyd: an iterative density-based approach for clusters with varying densities. *International Journal of Computers and Their Applications (IJCTA)*, 17(1):1–14, 2010.

[43] Mohammed J Zaki and Wagner Meira Jr. *Data mining and analysis: fundamental concepts and algorithms*. Cambridge University Press, 2014.

[44] Arkady Zaslavsky, Charith Perera, and Dimitrios Georgakopoulos. Sensing as a service and big data. *arXiv preprint arXiv:1301.0159*, 2013.