2017

Parity-based Data Outsourcing: Extension, Implementation, and Evaluation

Zhenbi Hu
Iowa State University

Parity-based data outsourcing: Extension, implementation, and evaluation

by

Zhenbi Hu

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Ying Cai, Major Professor
Soma Chaudhuri
Johnny S Wong

The student author and the program of study committee are solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2017

DEDICATION

I would like to thank my advisor Dr. Cai, who helped me in the research work, and also my friends who gave me suggestions in completing the experiments. Last but not the least, I would like to thank Department of Computer Science and Office of Information Technology at Iowa State University for financial assistance during my study there.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I really appreciate that my advisor Dr. Cai, who was very patient and kind in giving me guidance for my thesis. I also want to thank Dr. Chaudhuri and Dr. Wong for being my committee members and their insightful comments and improvement suggestions. Finally, I would like to thank my family members. They have been supporting me all the way.

ABSTRACT

Our research has developed a Parity-based Data Outsourcing (PDO) model. This model outsources a set of raw data by associating it with a set of parity data and then distributing both sets of data among a number of cloud servers that are managed independently by different service providers. Users query the servers for the data of their interest and are allowed to perform both authentication and correction. The former refers to the capability of verifying if the query result they receive is correct (i.e., all data items that satisfy the query condition are received, and every data item received is original from the data owner), whereas the latter, the capability of correcting the corrupted data, if any. Existing techniques all rely on complex cryptographic techniques and require the cloud server to build verification objects. In particular, they support only query authentication, but not error correction. In contrast, our approach enables users to perform both query authentication and error correction, and does so without having to install any additional software on a cloud server, which makes it possible to take advantage of the many cloud data management services available on the market today.

This thesis makes the following contributions. 1) We extend the PDO model, which was originally designed for one-dimensional data, to handle multi-dimensional data. 2) We implement the PDO model, including parity coding, data encoding, data retrieval, query authentication and correction. 3) We evaluate the performance of the PDO model. We compare it with Merkle Hash Tree (MH-tree) and Signature Chain, two existing techniques that support query authentication, in terms of storage, communication, and computation overhead.

CHAPTER 1. OVERVIEW

The last decade has seen the rising of cloud computing and storage services. With a small cost or even free, data owners can simply upload their databases to a cloud and let it manage the data and process queries on their behalf. By releasing data owners from day-to-day management of software and hardware, cloud services have the potential to save them significant operation cost. Data owners, however, are facing with difficulty in fully trusting cloud services. After all, the data is in the hand of a third party that is beyond their own administrative domain. A cloud may be compromised by outside hackers or corrupted by malicious insiders, thus sending users wrong answers, unintentionally or intentionally.

The above problem has led to a large body of research (e.g., [9], [21], [7], [8], [32], [33], [31]) on *query authentication*, i.e., enabling users to verify if the query results they receive are indeed correct. In the proposed techniques, data owners build an authentication data structure such as Merkle Hash tree or signature chain on a database and upload both to a server. When a user queries the database, the server returns not only the query result, but also a hard-to-forge verification object as a proof that the result includes all data items in the original database that satisfy the query condition. Existing research, however, has two major limitations:

- It considers only query authentication, but not query correction. Users can only detect if a query result is wrong. There is no mechanism for users receiving a wrong query result to find the correct answer.

- Existing solutions all rely on complex cryptographic techniques and therefore incur high cost in storage, communication and computation. The authentication data structure built for a set of data items can be many times larger than the data itself. A data item can be just a 32-bit integer, yet its one-way hash value needs at least 128 bits to ensure

minimum conflict. The number of bits for a digital signature is even more, usually at least 512. This cost is excessive for small records. The authentication data associated with a query result can also greatly increase the communication cost. Furthermore, there is hefty computation such as decryption incurred during the authentication process. These undesired effects are inherent from cryptographic techniques and especially concerned to the users of battery-powered mobile devices.

- Existing approaches require the server to run the code that builds and returns verification objects. Such code is not a standard component of a typical database management system (DBMS) and can be a security concern to a service provider when installed by a data owner. Companies such as Heroku Postgres [1] and Openshift [2] provide an instance of DBMS to their customers, who upload data there and use standard SQL commands for data manipulation. In such *Database as a Service* (DaaS) business model, the customers can upload only data, but not install their own code for query authentication.

To address the above problems, we propose a novel *Parity-based Data Outsourcing* (P-DO) [29]. This scheme associates the raw data with parity data and stores them on a set of servers that are from different service providers. Specifically, the data owner encodes a set of raw data D into a number of blocks. Each block has $k+p$ data, $(x_1, x_2, \cdots, x_k, y_{k+1}, \cdots, y_{k+p})$, where x_i $(1 \leq i \leq k)$ is a raw data in D and y_{k+j} $(1 \leq j \leq p)$ is a parity data generated for the k raw data. The p parity data are generated to enable error detection and correction. That is, all $k + p$ data in the block can be recovered as long as no more than e of them are missing or wrong, where e is determined by p. The data in the blocks are then outsourced to a set of $n = k + p$ servers, where the ith data in each block is stored on the ith server. To retrieve the data of interest, a user queries all servers. By allowing the user to receive at least $k + p - e$ correct data in each of the blocks that contain some data satisfying the query condition, PDO makes possible for users to perform query authentication and correction, and achieves so without using encryption or installing any additional software on a server.

Built on top of our work [29], this thesis makes the following contributions:

- We extend the PDO model, which was originally designed for one-dimensional data, to

handle multi-dimensional data.

- We implement the PDO model, including parity coding, data encoding, data retrieval, query authentication and correction.
- We evaluate the performance of the PDO model. We compare it with Merkle Hash Tree (MH-tree) and Signature Chain, two existing techniques that support query authentication, in terms of storage, communication, and computation overhead.

The rest of this thesis is organized as follows. We discuss more related work in Chapter 2. In Chapter 3, we introduce our PDO model. We first discuss how to support one-dimensional data, and then extend for multi-dimensional data. We present our implementation in Chapter 4, and evaluation its performance in Chapter 5. The concluding remarks are given in Chapter 6.

CHAPTER 2. RELATED WORK

2.1 Query Authentication

There are three parties involved in data outsourcing, *data owner*, *cloud server*, and *data users*. The data owner has a database and asks the cloud server to manage it. Users send their queries to the server for the data of their interest. The server is considered a third-party and therefore users want a proof that the query results they receive are correct. A query result is correct if it is *sound* (i.e., every data item included is from the original database) and *complete* (i.e., all data items in the original database that satisfy the query condition are included) [9; 21].

Simple solutions such as signing each data item with a digital signature [15] can prove the soundness, but not the completeness. The work by Devanbu et al. [9; 10] was among the first to study this problem. The proposed technique supports the authentication of range query over a list of data items. The data owner first sorts the data items and computes a one-way hash value for each data item. A *Merkle Hash* tree (MH-tree) [18] is then built on top of these hash values. Figure 2.1 shows 4 data items, $r_1 \leq r_2 \leq r_3 \leq r_4$, and a corresponding MH-tree, where $H(\cdot)$ denotes the hash function and "|" the concatenation of two nodes. The root node is signed with the data owner's private key and made known to all users. In response to a query, the server returns the result and a *Verification Object* (VO), which contains a set of data items for the user to reconstruct the root digest. For example, if the query result is $\{r_3\}$, then the corresponding VO includes N_1, r_2, and r_4, and the signed root digest. The user reconstructs the root digest with these data and compares it with the root digest published by the data owner. If the two digests match, the user can be assured that r_2, r_3, and r_4 are not tempered and their order is continuous in the original list.

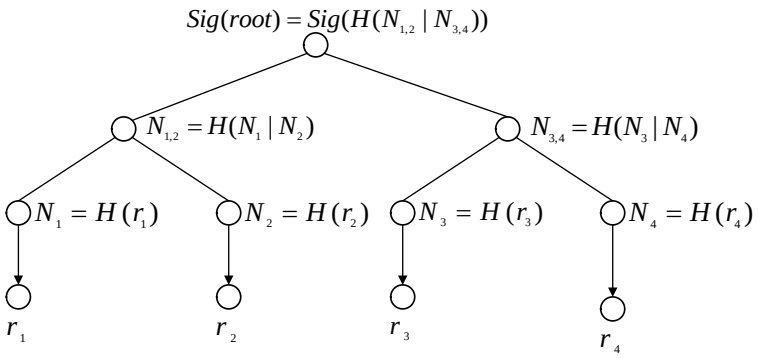


Figure 2.1: Merkle-Hash tree

In [21], Pang et al. proposed an alternative solution. Their idea is to sort the data items and then create a signature for each data item. Specifically, the signature for a record r_i is computed based on the digest of itself and the digests of its two immediate left and right neighbors. These signatures together form a *signature chain*, as illustrated in Figure 2.2. A query result $R = \{r_i \leq r_{i+1} \leq ... \leq r_j\}$ is accompanied with an VO that contains the signatures of each data item in R, plus the signatures of r_{i-1} and r_{j+1}, which are r_i's immediate predecessor and r_j's immediate successor, respectively. The chain formed by these signatures serves as the proof that for any two consecutive data items r_k and r_{k+1} in R, no data item r_x exists in the original database such that $r_k < r_x < r_{k+1}$. The completeness is verified by checking the signatures of boundary records r_{i-1} and r_{j+1}.

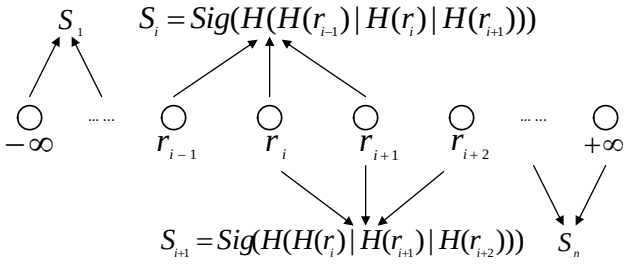


Figure 2.2: Signature chain

These two *Authentication Data Structures* (ADS) have their own advantages and disadvantages. With the MH-tree, only the root node needs to be signed, so the computation cost is low. However, $VO(q)$ needs to contain a set of tree nodes for one to reconstruct the root digest. So

the size of $VO(q)$ can be large, resulting in higher communication cost. Moreover, it is difficult to handle dynamic data. When a data item is changed, a significant part of the tree may need to be rebuilt. On the other hand, using signature chain incurs more computation cost, to both the data owner and query issuers, but has a smaller $VO(q)$ and thus less communication overhead. It is also more flexible to update data. When a data item is changed, only a few signatures need to be updated accordingly. Moreover, a user can perform boundary checking without having to know the two records that are immediate left and right to the sorted data in $R(q)$. This feature is useful in access control, since users may not be allowed to have access to all data items in the list.

The two ADS have since inspired a large body of research on query authentication. The work [7] considers multi-dimensional queries. The proposed technique maps the whole region into a multi-dimension data structure (KD-tree or R-tree). The partitions and the points in each partition space are ordered and then chained with their signatures. To process a query, the server returns all points inside the partitions that overlap with the query window and their corresponding signature chains. The work [32; 33] combines R*-tree and MH-tree to deal with spatial data. Other more complex queries such as KNN, skyline and function queries were studied in [8; 34; 30; 31]. The work [16; 22; 28; 27] considers the problem of data dynamics, where the outsourced data may keep changing. Authenticating aggregate queries such as sum and average was studied in [3]. While most solutions are software-based, where the authentication relies solely on the ADS created by the data owner, the work [4] explored tamper-proof, trusted hardware for authentication. The proposed solution can support a variety of database operations, but require hardware supports.

Regardless their differences, existing techniques support only query authentication, but not correction. Moreover, they rely on complex cryptographic techniques and thus incur high costs in storage, communication and computation.

2.2 Parity Coding

Parity coding has played a central role in the theoretical computer science [25]. The basic idea is to add some extra data to a message, which receivers can use to check consistency of

the message when received, and to recover data determined to be corrupted. Our research is inspired by error correction codes. These schemes generate parity data and add to a message such that it can be recovered by a receiver even when a number of errors (up to the capability of the code being used) were introduced. Techniques such as Hamming code [14] work on bit-level error correction. When a bit is known to be wrong, then the correct one must be its opposite. Our work is mostly related to the family of *block codes* (e.g., [24], [19]) that can correct a data (e.g., a 32-bit integer) whose value is in a large field. Block codes are similar in encoding a stream of raw data block by block, but different in the way of generating parity data. We will use the well-known Reed-Solomon codes [24] to explain how block codes work in general.

Reed-Solomon coding takes the first k raw data in the stream and then generates p parity data, where k and p are two pre-fixed values. The k raw data and p parity data together form the first *codeword*. It then takes the next k raw data from the stream and generates p parity data accordingly to form the second codeword, and so on so forth until all raw data in the stream are encoded. Let XY be a codeword, where $X = (x_1, x_2, ..., x_k)$ is a sequence of k raw data and $Y = (y_{k+1}, y_{k+2}, ..., y_{k+p})$ a sequence of p parity data generated for X. Each parity data y_{k+j} $(1 \leq j \leq p)$ is generated with a univariate polynomial function $y_{k+j} = \sum_{i=1}^{k} x_i a_j^{i-1}$, where a_j is a unique coefficient used to generate the ith parity data in each codeword. The way of generating parity data creates an equation system, where x_1, x_2, ..., and x_k are variables:

$$\begin{cases} y_1 = x_1 \\ y_2 = x_2 \\ \quad ... \\ y_k = x_k \\ y_{k+1} = x_1 + a_1 \times x_2 + a_1^2 \times x_3 + ... + a_1^{n-1} \times x_k \\ y_{k+2} = x_1 + a_2 \times x_2 + a_2^2 \times x_3 + ... + a_2^{n-1} \times x_k \\ \quad ... \\ y_{k+p} = x_1 + a_p \times x_2 + a_p^2 \times x_3 + ... + a_k^{n-1} \times x_k \end{cases} \tag{2.1}$$

The above equations form an overdetermined and independent system[1] and thus can be used for error detection and correction. Let (y_1, y_2, ..., y_{k+p}) be a codeword a user receives. The user builds the above equations and re-computes the parity data (y_{k+1}, y_{k+2}, ..., y_{k+p}), with the received (y_1, y_2, ..., y_k). If any computed y_{k+i} does not match the received y_{k+i} ($1 \leq i \leq p$), then at least one number in the codeword must be wrong. To correct the error(s), the user computes the most popular copy of $X = (x_1, x_2, ..., x_k)$. Among the $k + p$ equations, any subset of k equations can be solved to compute a copy of X. Different subsets may result in different copies of X, but as long as the number of errors in the block is not greater than $\lfloor \frac{p}{2} \rfloor$, the most popular copy must be the correct one. The simplest way to find the most popular copy is to enumerate all subsets of k equations and compute a copy of X for each subset. This brute-force approach is costly as totally there are $\binom{k}{k+p}$ subsets. Many efficient algorithms have been developed, such as Peterson decoders [12; 13], Berlekamp–Massey [5; 17], extended Euclidean algorithm [26]. Some of them (e.g., Syndrome decoding [23], maximum-likelihood decoding [6], minimum distance decoding [11]) can be used for block codes in general.

Existing parity coding techniques were designed mainly for purposes such as reliable data transmission over unreliable communication channels and reliable data storage on unreliable medias. They cannot be applied directly for query authentication and correction, which we will discuss shortly.

[1] A system is overdetermined and independent if no equation in any set of k equations can be derived algebraically from the other $k - 1$ equations in the set. A set of k equations is independent if the rank of their coefficients matrix is k [20].

CHAPTER 3. PARITY-BASED DATA OUTSOURCING

Our research has developed a novel *Parity-based Data Outsourcing* (PDO) model [29] which enables query authentication and correction without using cryptographic techniques. We use Figure 3.1 illustrates our idea. To outsource a database D, a data owner first selects a set of $k+p$ servers $\{S_1, S_2, ..., S_{k+p}\}$, each from a different cloud service provider, and decides a value of e, the maximum number of malicious servers that the system needs to tolerate. A server is said to be malicious if it does not return correct data back in response to a query (for whatever reasons, e.g., out of service, compromised by hackers, or network problems). We will assume the worst case, malicious servers may collude to forge data. The n servers are divided into two groups. The first k servers are used as *data servers* and the remaining $p = n - k$ servers as *parity servers*, where p has to do with e. The data owner then partitions D into k subsets, $\{D_1, D_2, ..., D_k\}$, and generates p sets of parity data, $\{P_1, P_2, ..., P_p\}$. D_i is uploaded to data server S_i $(1 \leq i \leq k)$ and P_j to parity server S_{k+j} $(1 \leq j \leq p)$.

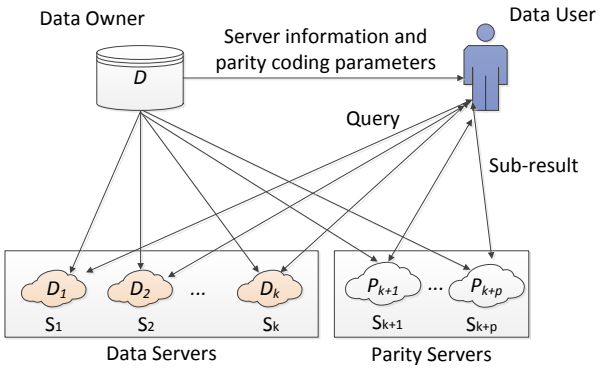


Figure 3.1: System Architecture

To retrieve the data of interest, a user sends a query to all servers. If all servers function properly, the user can simply filter out the query result R from the data returned by the k data

servers. However, some servers, up to e, may be malicious, returning no data or wrong data. So the challenge is how to enable the user to compute the correct R as long as a minimum of $n - e$ servers return correct data back. Our research is to address this challenge. To make this thesis self-contained, we first describe our solution for 1-dimensional data, which we published in [29], and then discuss how to extend it for multidimensional data.

3.1 One-dimensional Data

At the core of our solution is a novel parity coding technique, which we refer to as *Query-oriented Parity Coding* (QPC). Let XY be a codeword, where $X = (x_1, x_2, \cdots, x_k)$ is a sequence of k raw data and $Y = (y_{k+1}, y_{k+2}, ..., y_{k+p})$ a sequence of p parity data generated for X. QPC generates each parity data with a unique *coefficient vector* that consists of k numbers. Let $v_{k+i} = [a_{[i,1]}, a_{[i,2]}, ..., a_{[i,k]}]$ be the coefficient vector for y_{k+i} $(1 \leq i \leq p)$, we have $y_{k+i} = a_{[i,1]}x_1 + a_{[i,2]}x_2 + ... + a_{[i,k]}x_k$. We create the following equation system, where x_1, x_2, ..., and x_k are variables:

$$\begin{cases} y_1 = x_1 \\ y_2 = x_2 \\ \quad ... \\ y_k = x_k \\ y_{k+1} = a_{[1,1]} \times x_1 + a_{[1,2]} \times x_2 + ... + a_{[1,k]} \times x_k \\ y_{k+2} = a_{[2,1]} \times x_1 + a_{[2,2]} \times x_2 + ... + a_{[2,k]} \times x_k \\ \quad ... \\ y_{k+p} = a_{[p,1]} \times x_1 + a_{[p,2]} \times x_2 + ... + a_{[p,k]} \times x_k \end{cases} \tag{3.1}$$

Like Reed-Solomon codes, we can make the above equations form an overdetermined and independent system with appropriate coefficient vectors. However, QPC generates a parity data with a k-variate 1-degree polynomial, $y_{k+i} = a_{[i,1]}x_1 + a_{[i,2]}x_2 + ... + a_{[i,k]}x_k$ $(1 \leq i \leq p)$. If we select the coefficient vector $[a_{[i,1]}, a_{[i,2]}, ..., a_{[i,k]}]$ such that every coefficient in the vector is in between 0 and 1 and $\sum_{j=1}^{k} a_{[i,j]} = 1$, we can guarantee that the generated parity data

is always not less than the smallest number in X and not greater than the largest number in X. In other words, if all numbers in X are in between a query range (l, u), then the generated parity data must also be in the range. As such, if we store each of the $k + p$ data in XY on a different server, a user who issues a range query $q(l, u)$ to all servers can expect to receive XY together; even if some servers (up to $\lfloor \frac{p}{2} \rfloor$) are malicious, the user can still recover the original XY, thus achieving the purpose of authentication and correction.

We now discuss how to apply QPC in PDO for range query over D that is a set of N numbers. Recall in PDO, a data owner first selects a set of n servers $\{S_1, S_2, ..., S_n\}$. To tolerate at most e malicious servers at any one time, the first $k = n - 2e$ servers will be used as data servers and the remaining $p = 2e$ servers as parity servers. Each parity server S_{k+i} is also associated with a coefficient vector $[a_{[i,1]}, a_{[i,2]}, ..., a_{[i,k]}]$ $(1 \leq i \leq p)$ that meets the requirements of QPC.

Data Preparation. The data owner first sorts the numbers in D in ascending order. If $N \mod k \neq 0$, it appends a number of special token MAX (i.e., the largest number) to the sorted list to make its size divisible by k. D is then organized into sequences, each having k numbers. The first k numbers form the first sequence, the second k numbers form the second sequence, ..., and finally, the last k numbers form the last sequence. The data owner then applies QPC to encode each sequence $(x_1, x_2, \cdots, x_k)$ into a codeword $(x_1, x_2, \cdots, x_k, y_{k+1}, y_{k+2}, \cdots, y_{k+p})$, where $y_{k+j} = \sum_{j=1}^{k} a_{[i,j]} x_j$. After encoding all sequences into codewords, the data owner creates $k+p$ empty lists D_1, D_2, ..., D_k, P_1, P_2, ..., and P_p, and stores the codewords in the lists one by one, starting from the first codeword. For each codeword $(x_1, x_2, \cdots, x_k, y_{k+1}, y_{k+2}, \cdots, y_{k+p})$, store x_i in D_i $(1 \leq i \leq k)$ and y_{k+j} in P_j $(1 \leq j \leq p)$. After storing all codewords, the data owner inserts special token MIN (i.e., the smallest number) to the head and appends MAX to the end to each of the $k + p$ lists. A list may end with at most 2 MAX tokens (which happens when $N \mod k \neq 0$). Finally, the data owner uploads D_i to data server S_i $(1 \leq i \leq k)$ and P_j to parity server S_{k+j} $(1 \leq j \leq p)$.

The above procedure is illustrated in Figure 3.2, where sorted D = (1.2, 1.5, 2.0, 2.3, 2.9, 2.9, 3.6, 4.0, 4.2, 4.5, 4.8, 5.1, 5.7, 8.0). Five servers are used and one allowed to be malicious. So the first three servers are used as data servers and the remaining two as parity servers. The

coefficient vectors for the two parity servers are [0.1, 0.2, 0.7] and [0.4, 0.2, 0.4], respectively. The figure shows D is encoded into five codewords, each having two parity data, and the data list stored on each server. Note that the data lists stored on the servers remain sorted.

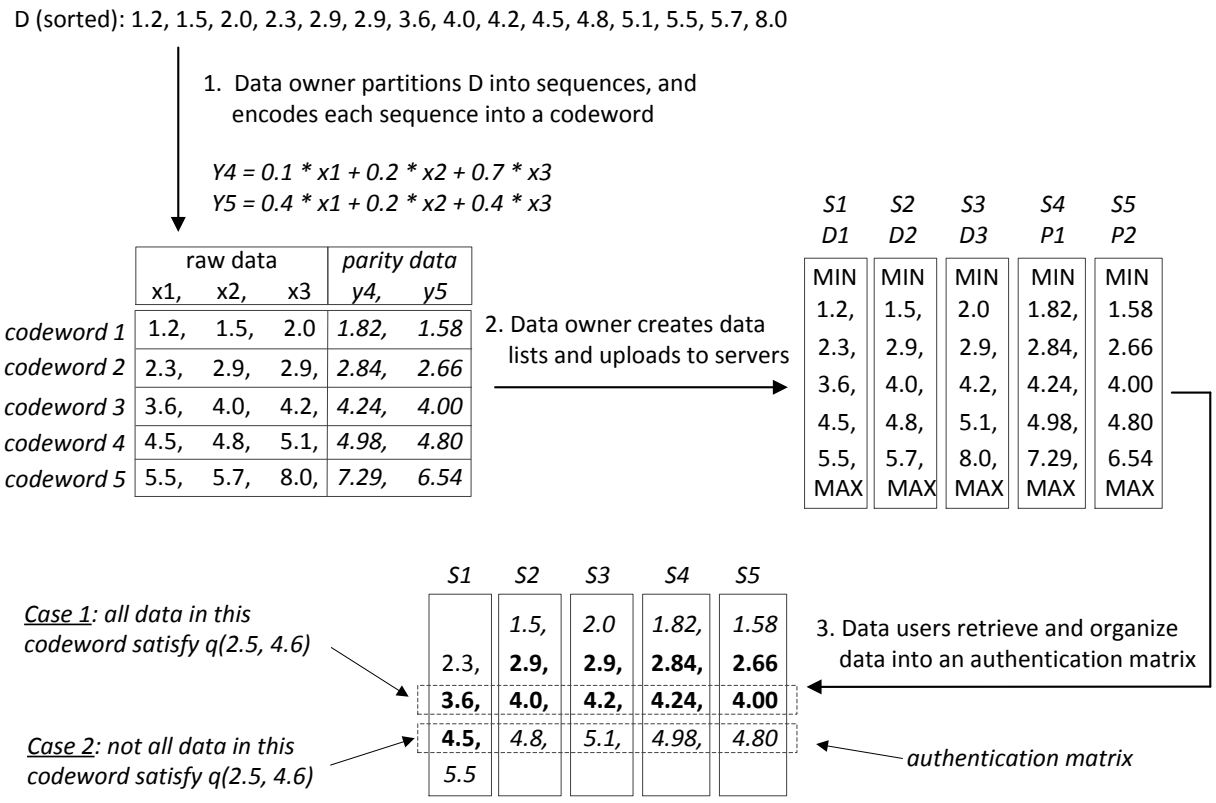


Figure 3.2: Data Preparation, Retrieval, and Authentication and Correction

Data Retrieval. To retrieve the data in between l and u, a user connects to all $k + p$ servers and ask each of them to return 1) the sublist of the data in between l and u, 2) the sublist's immediate left neighbor l' and right neighbor u', and 3) the position of l' in the list. Note all these requests can be done with standard SQL, without any special software to install and run on the server side.

Authentication and Correction. The user organizes the received data lists into an *authentication matrix*. The matrix has n columns. The list from server S_i is placed in column i, starting from either the first row or the second row, depending on the position of l'. Figure 3.2 shows the matrix created for the data lists returned for $q(2.5, 4.6)$. If all servers return correct

data back, the matrix shall have these features:

- All columns are occupied. The data in each column must be in the ascending order. The first data must be smaller than l (except when $l = MIN$), and the last data must be larger than u (except when $u = MAX$).
- Every row, except the first and last rows, are fully occupied, i.e., having $n = k + p$ data. For each fully occupied row, its first k data (raw data) must match the last p data (parity data), which can be re-calculated with their corresponding coefficient vectors.

If the matrix does not have these features, something is wrong. The user can detect the servers that are malicious and still compute the correct result. The ways of encoding D into codewords and retrieving data for a query $q(l, u)$ guarantee that, for any data in D that is in between l and u, the user will receive at least $n - e$ correct data in the codeword that contains the data and thus can recover the data if it is missing or wrong. Let $X = (x_1, x_2, \cdots, x_k)$ be the sequence that contains a data in D that is in between l and u. There are two scenarios:

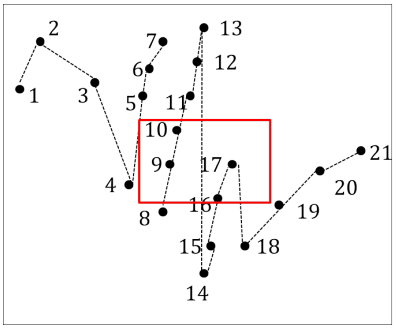
- Case 1: All data in X are in between l and u. In this case, all corresponding parity data $(y_{k+1}, y_{k+2}, \cdots, y_{k+p})$ must be also in the range. This is guaranteed by QPC. Since all servers are asked to return the data in between l and u and at least $n - e$ of them are trustworthy, the user will receive at least $n - e$ correct numbers in the codeword XY.
- Case 2: Some data, but not all, in X are in the range of l and u. In this case, the user will still receive at least $n - e$ correct numbers in the corresponding codeword, because all servers are asked to return two additional data, l's immediate left neighbor and u's immediate right neighbor.

The above two scenarios are illustrated in Figure 3.2. Note that the first row and the last row in an authentication matrix may not be fully occupied. This is fine since they won't contain any number in between l and u.

3.2 Multi-dimensional Data

We now consider how to extend our original solution for multidimensional data. To simplify our presentation, we will assume two-dimensional data. It will be easy to see that our technique can be applied for higher dimension spaces. The overall process of outsourcing and querying is similar: The data owner partitions the raw data and calculates the parity data, then distributes the raw data and parity data to n servers; Users queries all servers simultaneously for the data of their interest and authenticate, and correct if necessary, the query results returned from each server. Handling multi-dimensional data, however, is more challenging, because it cannot be sorted like 1-dimensional data.

Figure 3.3 shows 21 data items indicated by an assigned *id* number. The dataset is sorted by the horizontal coordinate, and the order is marked by the polylines. The table in the right side is the data distribution over 3 data servers, and each row is related with a codeword (the parity servers are not shown). Given the window query (marked as red), the correct query result is supposed to be $r = \{id_9, id_{10}, id_{16}, id_{17}\}$. According to the 1-dimensional PDO model, each server should return two extra items, the immediate left and right neighbor of the sub-result. For example, the sub-result for the server S_1 is the list $\{d_7, d_{10}, d_{16}, d_{19}\}$, in which d_{10} and d_{16} are in the window query while d_7 and d_{19} are the two extras. Similarly, server S_2 returns $\{d_{14}, d_{17}, d_{20}\}$, and server S_3 returns $\{d_6, d_9, d_{12}\}$. But apparently the returned objects are not sufficient for the data user to reconstruct complete codewords.



S1	S2	S3
1	2	3
4	5	6
7	8	9
10	11	12
13	14	15
16	17	18
19	20	21

Figure 3.3: Index Example

To solve the problem, we propose indexing data with *Grid File*. The data owner partitions the dataset into cells by horizontal and vertical lines. The grid cell is defined by these partition

lines, and each cell contains several data items or none. A very important feature about the structure of the Grid File is that the coordinate value of the partition lines in each dimension are independent, and can be treated as an one-dimensional dataset. For example, in Figure 3.4, the horizontal lines can be expressed as an one-dimensional dataset $s_h = \{0, 5, 10, 15\}$ and vertical lines $s_v = \{0, 5, 10, 15\}$. The basic idea is that the data items within a grid cell can be sorted by a certain coordinate, then our 1-dimensional technique can be applied to compute parity data, which are also within the grid cell. For a user's query, each cloud server returns all overlapped grid cells back. That is, if a grid cell overlaps with the window query, then all data items in it are returned. We explain as follow.

For each dimension of the Grid File, the data owner applies the 1-dimensional PDO technique to outsource the partition lines. If the number of data items in a cell is not a multiple of the number of data servers, we add some dummy data items in the cell. This is to ensure that we can generate complete codewords for the cell. We will discuss how to compute the parity data shortly. After the codewords are created in a cell, they are distributed among the multiple servers. The same process is applied for all cells.

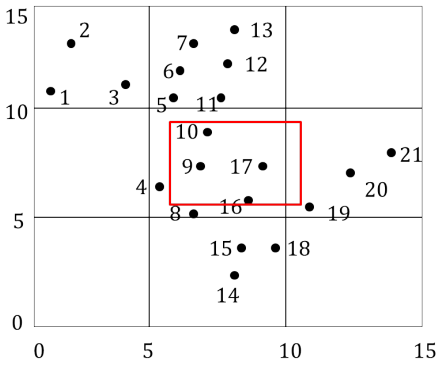
Taking a two-dimensional dataset for example, the equation system can be expressed in equation 3.2. Let $X = \{d_1, d_2, ..., d_k\}$ be a group of raw data. For each data item d_i, we use $(d_i.val_1, d_i.val_2)$ to denote d_i's values in the first dimension and the second dimension, respectively. Let $Y = \{y_{k+1}, y_{k+2}, ...y_{k+p}\}$ denote the parity set generated for X. Let $V = \{v_{k+1}, v_{k+2}, ..., v_{k+p}\}$ be the chosen coefficient vectors for Y, and $v_{k+i} = \{a_{[i,1]}, a_{[i,2]}, ..., a_{[i,k]}\}$. Both dimensions share the same set of coefficient vector, so $y[k+i]$ contains a two-dimensional value field. The first dimension value $y_{k+i}.val_1 = d_1.val_1 \times a_{[i,1]} + d_2.val_1 \times a_{[i,2]} + ... + d_k.val_1 \times a_{[i,k]}$, and second dimension value $y_{k+i}.val_2 = d_1.val_2 \times a_{[i,1]} + d_2.val_2 \times a_{[i,2]} + ... + d_k.val_2 \times a_{[i,k]}$. We have the equation $y_{k+i}(val_1, val_2) = [d_1(val_1, val_2), d_2(val_1, val_2), ..., d_k(val_1, val_2)] \times (v_{k+i})^T$:

$$\begin{cases} y_1(val_1, val_2) = d_1(val_1, val_2) \\ y_2(val_1, val_2) = d_2(val_1, val_2) \\ \quad\quad ... \\ y_k(val_1, val_2) = d_k(val_1, val_2) \\ y_{k+1}(val_1, val_2) = [d_1(val_1, val_2), d_2(val_1, val_2), ..., d_k(val_1, val_2)] \times (v_{k+1})^T \\ y_{k+2}(val_1, val_2) = [d_1(val_1, val_2), d_2(val_1, val_2), ..., d_k(val_1, val_2)] \times (v_{k+2})^T \\ \quad\quad ... \\ y_{k+p}(val_1, val_2) = [d_1(val_1, val_2), d_2(val_1, val_2), ..., d_k(val_1, val_2)] \times (v_{k+p})^T \end{cases} \tag{3.2}$$

Users retrieve data in two steps. Let $q = (< l_1, u_1 >, < l_2, u_2 >)$ be a query, i.e., retrieving all data items whose values in the given window. The first step is to identify q's minimum boundary cells (MBC), i.e., the minimum set of cells which covers the query window. As a consequence, the user knows that the sets of partition lines in each dimension. For example, among the coordinate values in the first dimension, assume $l_1^{'}$ is the maximum number which is no bigger than l_1, and $u_1^{'}$ is the minimum number which is no less than u_1, then all numbers between the range $l_1^{'}$ and $u_1^{'}$ form the set of partition lines for the first dimension. We denote the set as s_1. Note that s_1 can be retrieved by sending a range query to all servers, and verified with our 1-dimensional technique. In the same way, the set of partition lines for the second dimension can be retrieved and verified and we denote this set as s_2. As such, the MBC for q is formed by two sets s_1 and s_2. The authentication and verification of the two sets can be done with the one-dimensional data outsourcing technique.

The second step is to retrieve the data items in MBC. The boundary of each grid cell in MBC can be used as the range of a window query. The authentication and verification are performed separately for each cell. Users sort the data items according to a certain dimension, and organize them into authentication matrix for authentication and correction.

Figure 3.4 illustrates the above procedure. The space is partitioned by horizontal lines $s_h = \{0, 5, 10, 15\}$ and vertical lines $s_v = \{0, 5, 10, 15\}$. The data owner partitions the set s_h, calculates their parity data, and then uploads the dataset to the the corresponding servers.



S1	S2	S3
1	2	3
5	6	7
11	12	13
4	8	9
10	16	17
19	20	21
14	15	18

Figure 3.4: Grid File Partition

The process for s_h works in the same way. The data items in each cell are sorted by horizontal coordinate, partitioned for three data servers (the parity servers are not showed). The window query is the rectangle in the figure. It overlaps with two cells $C_{[5,10];[5,10]}$ and $C_{[10,15];[5,10]}$, where the index are the partition lines in horizontal and vertical directions. The MBC is defined by two sets of partition lines $s_1 = \{5, 10, 15\}$ in the horizontal direction and $s_2 = \{5, 10, 15\}$ in the vertical direction. Set s_1 can be obtained by range queries from the authentication structure of dataset s_h, while s_2 can be retrieved from s_v. From the table on the right side, we can notice that the $C_{[5,10];[5,10]}$ contains two codewords, while $C_{[10,15];[5,10]}$ contains one codeword. These codewords can be obtained from the two cells independently. For instance, the boundary of $C_{[5,10];[5,10]}$ can be used as a window query to retrieve the two codewords in it.

CHAPTER 4. IMPLEMENTATION

We have implemented PDO for both 1-dimensional data and multi-dimensional data, using Java language. For purpose of performance study, we have also implement two related techniques, MH-tree and signature chain. We describe as follows.

4.1 PDO for One-dimensional Data

The outsourcing model mainly contains three parts, building the authentication structure, authenticating query result, and correcting if needed. The model structure is displayed in Figure 4.1.

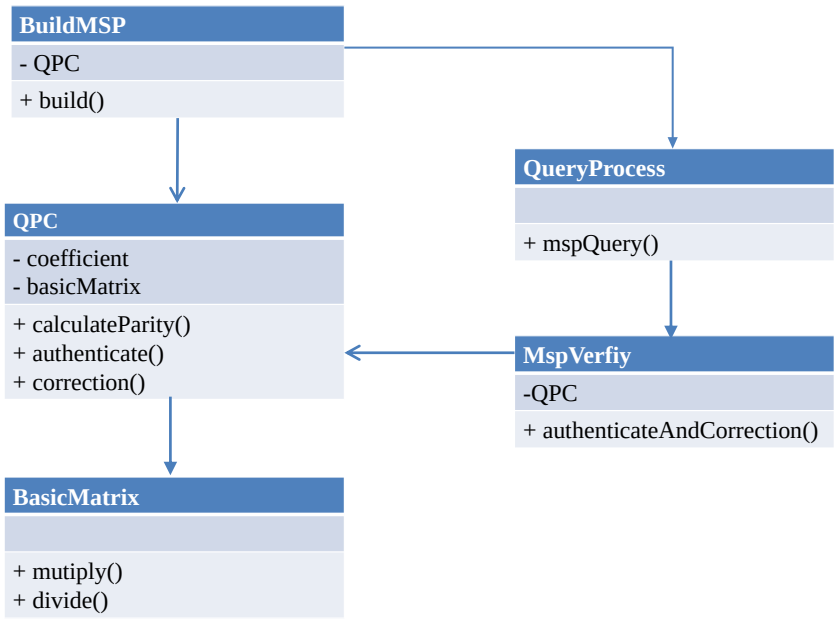


Figure 4.1: One Dimensional Parity Coding Class Design

- Class **BuildMSP** is for the data owner to partition the raw data, and prepare the parity

data, and distribute the dataset among the corresponding cloud servers. The number of data servers and parity servers are given in a configuration file, which is shared by all the classes in the project. The building process is explained in Algorithm 1. In addition, to prepare the parity data part of a codeword, the function calls the parity calculation function in Class **QPC**.

Algorithm 1 Build()

```
1: AddHead()          //adding the special starting token
2: For each codeword, do          //create codeword one by one
3:     PrepareRawData()          //prepare the raw data part
4:     PrepareParityData()          //prepare the parity data part
5: End For
6: AddEnd()          //adding the special ending token
```

- Class **QueryProcess** is used by cloud servers to receive queries from users. After processing queries, they return the results to users. Each server processes the queries independently, and returns a sub-result for a query.

- Class **MspVerify** is used by users to verify the received query results. At first, a user authenticates the query result. If the sub-results from cloud servers do not match with each other, it starts the correction process. The algorithm is given in Algorithm 2. In line 1, all the sub-results for a query is organized in the authentication matrix. The authentication of each codeword is checked in line 3, and the correction is executed in line 5.

Algorithm 2 authentionOrCorrect(*subResults*, *query*)

```
1: Organize each sub-result in a matrix
2: For each codeword, do
3:     If it is authenticated, then adding to result
4:     Else recover the real sub-result.
5: End For
```

- Class **QPC** is responsible for the implementation of calculating parity data, authenticating query result, and correcting if needed. The functions are explained as follows:

- *calculateParity*() mainly receives the raw data, and multiplies the coefficient vector. It returns the computed parity data.
- *authenticate*() receives the organized codeword, and recalculates the equation system to check whether the values of the elements match with each other. It returns a boolean data type.
- *correction*() recovers the real result if there exists inconsistence in authentication. The function is invoked by the methods in class **MspVerify**, and it contains two important algorithms. One is to find out all the combinations of k of $k+p$ equations, since every k equations decide a copy of X in equation system. The collection of all the subsets is recursive process, which display in Algorithm 3. The final result is returned in *subsetsList*. Another algorithm is to count the most popular subset in Algorithm 4. It uses a HashMap to count the copy of solutions and the corresponding frequency, then sorts the elements in the HashMap to get the most popular with the highest frequency.

Algorithm 3 subset(*subsetsList*, *currentList*, *index*, *equations*[])

1: If currentList.size = k, then add *currentList* to *subsetList*
2: While $index < k + p$, do
3: Add $equations[index + 1]$ to *currentList*
4: subset(*subsetList*, *currentList*, $index + 1$, *equations*[])
5: Remove $equations[index + 1]$ from *currentList*
6: $index++$
7: End While

Algorithm 4 mostPopularSolution(*subsetsList*)

1: create a HashMap *solutionMap*, which stores the copy of solution and the frequency
2: For each *subset* in *subsetsList*, do
3: Add to *solutionMap*, and increase the $frequency$ by one
4: End For
5: Use HeapSort to order the copies of solutions by $frequency$
6: Return the most popular one

- Class **BasicMatrix** contains the basic matrix calculation, such as multiplication and division operations. These methods are called by the class **QPC**.

4.2 PDO for Multi-dimensional Data

Recall that we use Grid File to index the multi-dimensional dataset. To construct the Grid File index, we need to specify three parameters. The first one is the boundary of the Grid File, which can cover all the dataset. The second one is the dimension of the dataset, apparently the dimension of the structure is the same as the dataset. The third one is the capacity of the grid cell. After the three parameters are fixed, the data items are added into the Grid one by one. At the beginning, the whole Grid is treated as a big cell. When the number of data items added reaches to the capacity of the big cell, the cell is cut into two new cells. The data items are reassigned into the corresponding new cells. In this way, the data items from the original dataset keep inserting into the grid cells, and a cell is divided by a cutting plane (a cutting line in two-dimensional case) once its capacity is full.

When the Grid File structure is built, the parity coding technique is applied on for data items among these cells. Here, each cell is treated as an independent unit, and each dimension of the data items is independent too. First, if the number of items in a cell is not divisible by the number of data servers k, then dummy items are added. Second, the data items are sorted by the first dimension value. Then every k data items can be used to calculate a group of parity data. Accordingly, a codeword can be created by combining the k raw data and the related parity data. Finally, all codewords are uploaded into the cloud servers.

The class designed for the parity-based multi-dimensional data outsourcing is displayed in Figure 4.2. It includes the building of the Grid File, query processing and the performance measurement of authentication time and correction time.

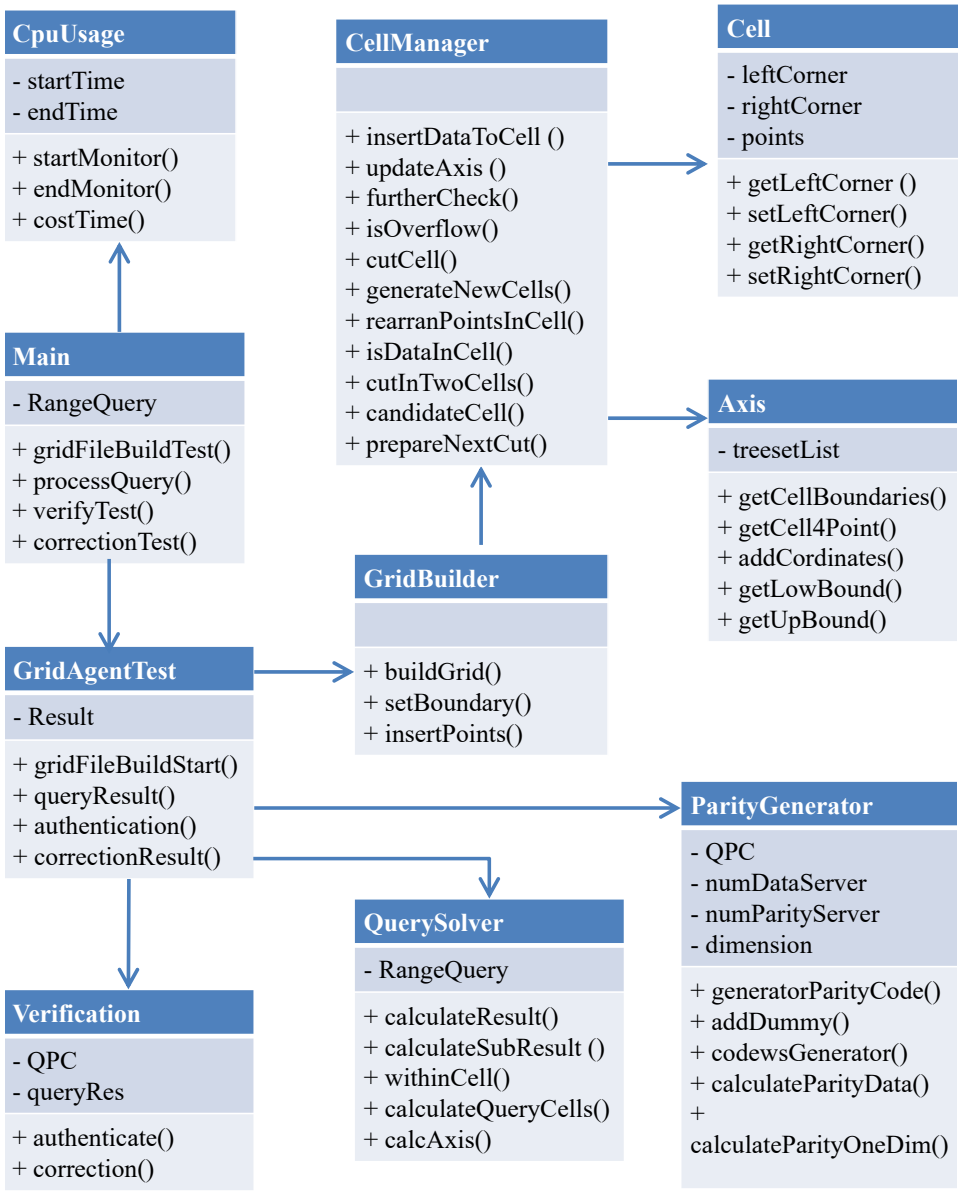


Figure 4.2: Multi-dimensional Data Outsourcing Class Design

The definition of each class and the related algorithms are explained as follows:

- Class **Main** is the entry point for testing the computation time, authentication time and correction time.

- Class **GridAgentTest** is the agent which receives the query and processes the tasks from the class Main. The primary attributes and functions are as follows:
  - Result: stores the query result after query processing.
  - gridFileBuildTest() builds the authentication structure. It contains two steps, the first one is creating the Grid File index, another is generating the parity data among the grid cells.
  - processQuery() receives the window queries, and returns the query results.
  - authenticate() checks the query result whether it is sound and completeness.
  - correctionResult() corrects the query result.
- Class **GridBuilder** is responsible for creating the Grid File index.
  - buildGrid() implements of building the Grid File structure.
  - setBoundary() sets the boundary of the Grid File.
  - insertPoints() inserts the raw data items to the Grid File.
- Class **ParityGenerator** calculates the parity data and generates the codewords after the establishment of the Grid File index. The algorithm is given in Algorithm 5.
  - QPC: implements the parity data calculation.
  - numDataServer: the number of data servers.
  - numParityServer: the number of parity servers.
  - dimension: the dimension of the dataset.
  - generatorParityCode() traverses through all the cells, and sorts the data items in each cell to prepare for the generation of the codewords.
  - addDummy() adds dummy items to a cell if necessary.
  - codewsGenerator() generates the codewords within a cell. The specific calculation is in $calculateParityData()$.

- calculateParityData() prepares the parity data for a group of raw data items. For each dimension, it invokes the function *calculateParityOneDim*().
- calculateParityOneDim() calculates the parity data for a certain dimension of the data items.

Algorithm 5 ParityGenerator(*QPC*, *numDataServer*, *numParityServer*)

```
For each cell in Grid File, do
    If the number of items in cell is not divisible by numDataServer, then addDummy()
    Sort all the items in cell by the first dimension
    For each numDataServer items, do
        For each dimension of these items, do
            Calculate parity data
        End For
        Combine the raw data and parity data into a codeword
    End For
End For
```

- Class **CellManager** is the manager of building the Grid File structure, including the updating of the Grid File structure, updating of the coordinate axis. The functions in details are as follows:
  - insertDataToCell() inserts a data item into the Grid. The inserting process might cause the overflow of a grid cell, consequently it leads to the cutting requirement of a series of cells.
  - updateAxis()updates the set of axis. When a cell is cut, a new coordinate value would appear, and it should be added into class Axis.
  - furtherCheck() double-checks whether the new divided cell overflowed or not, after a cell is cut, .
  - isOverflow() checks whether a cell overflow, basically it checks whether the number of data items exceeds the threshold of the cell capacity.
  - cutCell() cuts a series of affected cells which overlap with the cutting plane.
  - generateNewCells() updates the set of cells in CellManager, since there are new cells generated.

  - rearranPointsInCell() rearranges the data items in the new cells after the original cell is cut.
  - isDataInCell() checks whether a data item is in a cell.
  - cutInTwoCells() implements the procedure of partitioning a cell into two.
  - candidateCell() finds out all the cells affected by the cutting plane.
  - prepareNextCut(): indicates the cutting rules.

- Class **Cell** is used for creating the object of the grid cell. It has three attributes:
  - leftCorner: indicates the coordinate of the left corner.
  - rightCorner: indicates the coordinate of the right corner.
  - points: contains all the data items in the current cell.

- Class **Axis** keeps all the coordinate values in all dimensions, and provides the functions related with coordinate value.
  - getCellBoundaries() identifies the MBC for the query when a window query is given.
  - getCell4Point() inputs a data item, and outputs the cell which can cover it.
  - addCordinates() updates the coordinates in Axis.
  - getLowBound() returns the lower boundary of a value.
  - getUpBound() returns the upper boundary of a value.

- Class **QuerySolver** communicates with the third-party to fetch the query result.
  - RangeQuery: stores the query.
  - calculateResult() collects all the sub-results from all the cloud servers.
  - calculateSubResult() gets the query result from a certain server.
  - withinCell() identifies the cell for a record received from a server, since it can contains extra information, such as dummy mark.

- calculateQueryCells(): after the MBC is obtained, assembles all the related cells. The implementation is a recursive procedure, which is given in Algorithm 6.
- calcAxis(): returns all the overlapping coordinate values.

Algorithm 6 calculateQueryCells(*cellsResult*, *current*, *axises*, *level*)

```
1: If level == axises.size, then add current to cellsResult.
2: For each value in level-th element in axises, do
3:     Add value in current
4:     calculateQueryCells(cellsResult, current, axises, level + 1)
5:     Delete value from current
6: End For
```

- Class **Verification** gives the functions of authentication and correction of query results.
  - QPC: calls the verification operations in it.
  - queryRes: the returned result from cloud servers.
  - authenticate() checks whether the calculation based on the codeword matches with the received values. The process of authenticating a query result is given in Algorithm 7
  - correction() corrects the query result if there are errors in it.

Algorithm 7 authenticate(*QPC*, *queryRes*, *axises*, *level*)

```
1: For each subResults in queryRes, do
2:     Organize the subResult into columns
3:     For each codeword, do
4:         QPC.authenticate(codeword)      //check authentication
5:     End For
6: End For
```

4.3 Implementation of Related Work

We compare the parity-based data outsourcing in one-dimensional dataset with the two existing techniques, MH-tree and Signature Chain. The performance comparison includes the time cost in computation and authentication. The computation cost refers to the time it requires to build the authentication structure. That is, the computation cost is in the data

owner side. The authentication cost is the time which the user uses to verify the query result. We start from the MH-tree scheme, the constructing of the MH-tree can be implemented layer by layer. It is a binary tree and each ancestor node is built upon its two child nodes. For simplicity, we design a full binary tree. In order to facilitate the query processing through the tree, each internal node contains the range of attribute it covers. Therefore, the query process and the constructing of the VO can be done at the same time. In the Signature Chain, two dummy nodes should be added at the two chain ending. The query result is continue partial chain from the original Signature Chain. In order to prove that there are no data missing at the boundary of the query result, the VO needs to include two extra nodes beside the real result. In the PDO scheme, there are also some places which need to be noticed in implementation. The original dataset should be sorted in ascending order before applying the parity coding technique. A certain number of dummy items are added if the size of original dataset is not divisible by the number of data servers.

The class design is displayed in Figure 4.3. The process starts from building structure, then query processing, verification at last. Another feature is that in the class VerifyScheme, the two existing schemes support only authentication, but not correction.

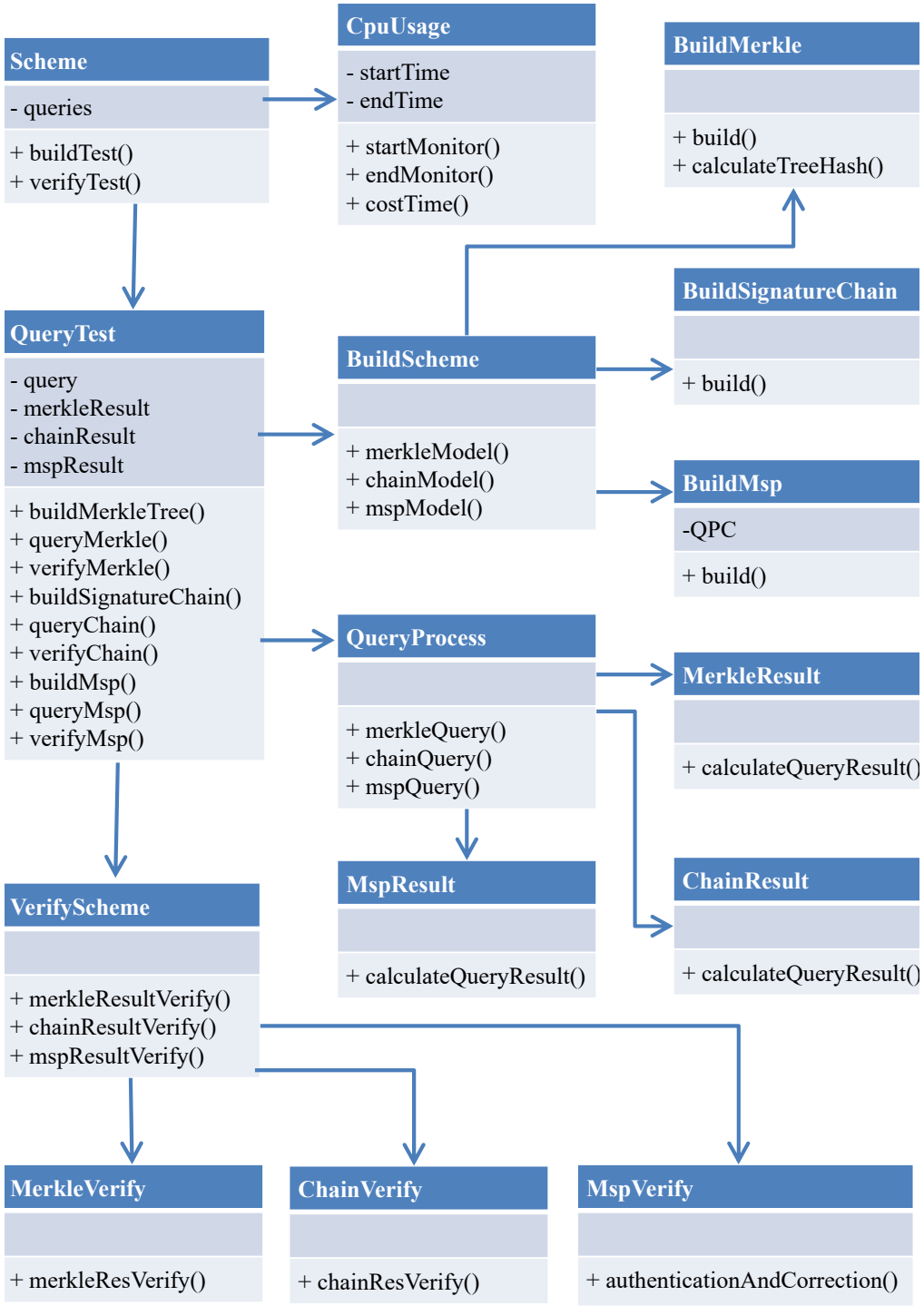


Figure 4.3: Performance Comparison Class Design

The classes and the related algorithms are explained as follows:

- Class **Scheme** is the access entry to test the time used for building an authentication structure, and verifying a query cost. It receives queries from the users.
- Class **CpuUsage** is used by the class **Scheme** to record the starting time, ending time and time cost of an action.
- Class **QueryTest** provides the whole access points for building structure, query processing, verification.
- Class **BuildScheme** is called by the class **QueryTest**. And it contains all the three building model: $merkleModel$, $chainModel$, $mspModel$.
- Class **BuildMerkle** provides the implementation of $merkleModel$. The tree is built from the bottom to the root. For each leaf node, it stores the real value and the hash value of the node. For each internal node, it stores the range, which the child nodes cover, and also the hash value. Only the the root of the tree is digitally signed. The building procedure is given in Algorithm 8. To calculate the hash value of each node, the Algorithm 9 involves in a tree postorder traverse.

Algorithm 8 buildMerkle(*records*)

```
For each level from the bottom to up, do
    For each node, do
        Calculate the range it covers, and store in the node
    End For
End For
Traverse the tree, and calculate the hash value of each node
Sign the root
```

Algorithm 9 calculateTreeHash(root)

```
If root == NULL, then return
calculateTreeHash(root.left)
calculateTreeHash(root.right)
Calculate the Hash value of the root
```

- Class **BuildSignatureChain** provides the implementation of $chainModel$. The process is given in Algorithm 10. First, generate two dummy nodes at the two ending, with a

special token value. Then, for each node, compute the hash value, and sign a signature over itself and its two neighbors.

Algorithm 10 buildChain(records)

```
addDummy(MIN)
addDummy(MAX)
For each record, do
    Calculate the hash value H(record) of the presented node
End For
For each chain node, do
    Sign the node, Sig(H(node)) = Sig(H(H(node.left)|H(record)|H(node.right)))
End For
```

- Class **BuildMsp** is explained in previous section.
- Class **QueryProcess** processes the queries representing cloud servers, and returns the query result to the class $QueryTest$.
- Class **VerifyScheme** receives the query result from **QueryTest**, and passes to the three different verification module, $merkleResultVerify$, $chainResultVerify$, $mspResultVerify$.
- Class **MerkleVerify** provides the implementation of the module $merkleResultVerify$. The process is similar to the process of building the authentication structure, starting from the bottom, reconstructing the hash value of the root, and comparing with the signed one.
- Class **ChainVerify** provides the implementation of $chainResultVerify$, which is described in Algorithm 11.

Algorithm 11 chainResVerify($Query$, $Result$, $VO(query)$)

```
For each sig(node) in VO, do
    If Sig^{-1}(Sig(node)) ≠ H(H(node.left)|H(node)|H(node.right)), reject Result.
    Else if node.value is not in Query range, Reject Result.
End For
Accept Result.
```

- Class **MspVerify** is described in previous section.

CHAPTER 5. PERFORMANCE STUDY

We compare the performance of PDO, MH-tree, and Signature Chain, through both analysis and experiments. For all three techniques, the data owner needs to pre-process the original dataset before delegation to the cloud servers. The cost for the data owner includes that (1) Storage overhead, defined to be the ratio between the size of the data that needs to be uploaded to the servers and the size of the original data; (2) Preparation cost, i.e., the time incurred in building an authentication data structure. To data users, the main overhead includes (1) Communication cost, which is the size of the data received from the servers; (2) Authentication cost, which is the time for the users to authenticate the received query result; (3) Correction cost, which is the time incurred in correcting a wrong query result.

In addition to analysis, we have studied their performance through experiments, where we are our code on a server with Intel Xeon 4-core CPU 2.67GHz, 16GB RAM.

5.1 Analysis

Storage overhead. Let $|D|$ be the number of records in the database D, k is the number of data servers, and $p = n - k$ is the number of parity servers, thus $e = \lfloor \frac{p}{2} \rfloor$ is the maximum number of malicious servers allowed. Let t denote the size of a data record. The length of a hash value can be varying, e.g. 128 bits, 256 bits, 512 bits, depending on the practical algorithm used. Let $|H|$ be the length of a hash value. The length of a digital signature is normally longer than a hash value, it can be 1024 bits or 2014 bits and so on. Let $|sign|$ be the length of a signature. The storage cost for each of the scheme is as follows:

- *Parity Coding*: the database D will be encoded into $\lceil \frac{|D|}{n-2e} \rceil$ codewords, each consisting k raw data and p parity data records. Since the parity data has the same size as the raw

data, the total size of raw data and parity data is $(\lceil \frac{|D|}{n-2e} \rceil) \cdot n \cdot t$.

- *MH-tree*: each node in the tree (leaf or internal node) stores the range it represented, a hash value. We assume that the size of the rangee is the same as a data value. Besides, the root has a signature. Therefore, the total size of authentication structure is $2 \cdot |D| \cdot (t + |H|) + |sign|$.

- *Signature Chain*: in this scheme, every chain node requires not only a data value and hash value, but also a signature based on itself and its two neighbors. Plus, two dummy nodes are added in the two ending. In consequence, the total size is $(|D| + 2) \cdot (t + |H| + |sign|)$.

Communication cost. Let $|q|$ be the query size, specifically the number of records which satisfy the query condition. The communication cost for each of the scheme is as follows:

- *Parity Coding*: besides the records within the query range, each server will send back two more extra records. Accordingly the data records received from each server is $\lceil \frac{|q|}{n-2e} \rceil + 2$. So the total communication cost is $(\lceil \frac{|q|}{n-2e} \rceil + 2) \cdot n \cdot t$.

- *MH-tree*: the query result is a contiguous leaf nodes, and VO contains two extra leaf nodes in the boundary, the auxiliary hash values of the nodes along in the search path, and the signature of root. Basically the size of the query result is about $(|q| + 2) \cdot t$. Since the height of the tree is $log|D|$, the size of hash values in VO is around $2 \cdot log|D| \cdot |H|$. Thus, the total is $(|q| + 2) \cdot t + 2 \cdot log|D| \cdot |H| + |sign|$

- *Signature Chain*: the items sent back includes a consecutive chain nodes which are in the query range and the two extra nodes at the boundary. So the cost is $(|q| + 2) \cdot (t + |sign|)$.

5.2 Experiments

We compare the preparation cost, authentication cost and correction cost using simulation, and we will show how these cost are impacted by the size of the dataset, query size, and the number of servers.

Impact of Dataset Size. Given a dataset, we study the impact of datasize size on preparation cost. For the existing schemes, we record the time of building a MH-tree and Signature Chain.

For our approach, we use 5 servers in total, and have two configurations on the maximum number of malicious servers: $e = 1$ and $e = 2$. Accordingly, we identify the two settings as PDO-5-1 and PDO-5-2. We increase the data size (i.e., the total number of records in the dataset) from 2^{10} to 2^{17}. The results are plotted in Figure 5.1, and it shows that Signature Chain is the worst performer. This is not surprising, since this scheme needs one-way hash on every number and builds a signature for every hash value. MH-tree also requires to one-way hash every node of the binary tree. However, it needs to create only one signature for the tree root. Our result shows that MH-tree outperforms Signature Chain in 2-3 orders of magnitude in generating the authentication structure. The two PDO schemes outperform MH-tree in another 3 orders of magnitude. These results convince that generating parity data, which is to compute a multivariate linear polynomial function, incurs much less computation time than computing one-way hash function and digital signatures. Our results also show that tolerating more malicious servers result in more preparation cost, but not in a significant way.

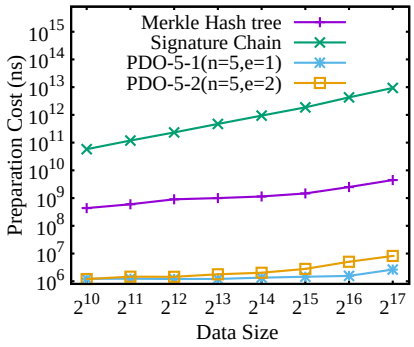


Figure 5.1: Impact of Data Size

Impact of Query Size. In this study, we first compare PDO with MH-tree and Signature Chain in terms of authentication cost. Again, we use 5 servers in PDO but have two configurations of the maximum number of malicious servers, $e = 1$ and $e = 2$. In each simulation, we apply the three techniques in authenticating a query result, the size of which varies from 1000 to 5000. We consider the worst scenario: the query result is correct so each technique needs to check every number. In reality, the checking stops whenever an error is found. The results are plotted in Figure 5.2 (a). It shows that Signature Chain incurs most time in authentication. Again, this scheme needs to recompute all hash values and verify their signatures. MH-tree

performs better than Signature Chain. It also requires to recompute all hash values but needs to verify only one signature. Note that their performance gap is not as great as in Figure 5.1. This is due to the fact that verifying a signature is less computation-intensive than creating a signature. The two PDO schemes outperform the two existing approaches in 2-3 orders of magnitude. In PDO, the authentication is done by recomputing the parity data. Again, this computation cost is much less than computing one-way hash values, not mentioning computing digital signatures.

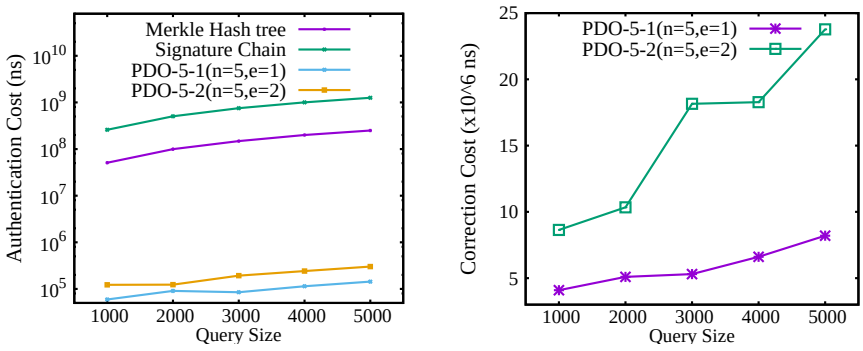


Figure 5.2: Impact of Query Size

In this study, we also look at the cost of correcting a wrong query result. MH-tree and Signature Chain do not support correction, so we focus on our own technique. In our simulation, we randomly remove one or two data sets (depending on the settings of the number of malicious servers) returned from the servers and apply our technique to recover them. We vary the query size from 1000 to 5000 and plot the results in Figure 5.2 (b). As the query size increases, the correct cost increases, since more data needs to be recovered. For the same reason, more malicious servers result in more correction cost, which is showed in the figure.

Impact of Server Number. In PDO, a data owner outsources data to a number of servers. We are interested in how the number of servers impacts the performance of this scheme. For this purpose, we generate a data set of 2^{30} numbers and vary the number of servers n from 3 to 10. We simulate two settings of malicious servers, $e = 1$ and $e = 2$, which requires n to be at least 3 and 5, respectively. Figure 5.3 (a) illustrates the preparation costs. It shows

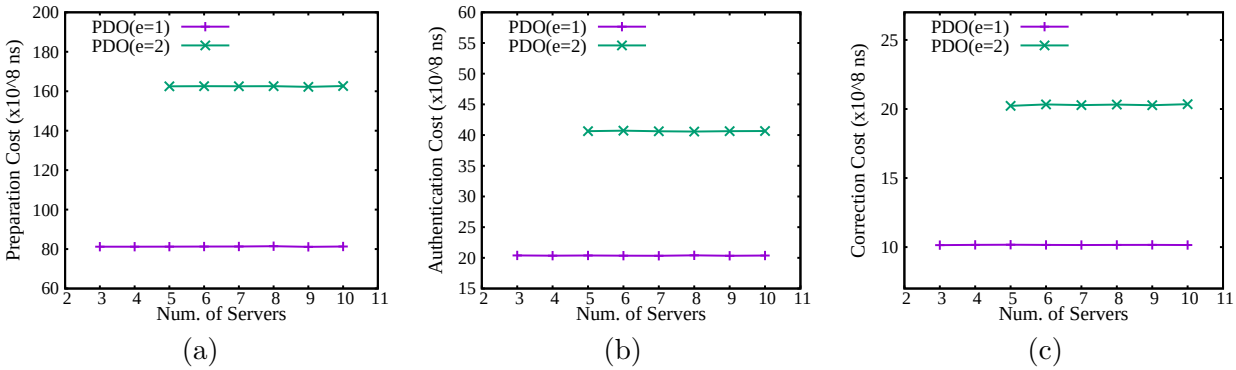


Figure 5.3: Impact of Server Number

that this cost is insensitive to the number of servers. Both curves are nearly flat. This can be explained as follows. When e is fixed, increasing n results in a larger k (the number of data servers) and thus, a smaller redundancy rate $\frac{2e}{n-2e}$. The result is less number of parity data to be generated. However, each parity data is now generated with k raw data. In other words, the number of parity data reduces but the cost of generating a parity data increases. Apparently the two factors offset each other and result in a nearly constant preparation. For the same reason, we expect the number of servers to have little impact on the authentication cost and correction. This is confirmed in Figure 5.3 (b) and (c), respectively. Note that when n is fixed, but e increases, we have more parity data to generate out of a same set of data, thus the computation time increases, which we can observe in the figures.

5.3 Extension

The analysis and simulation above compare the PDO scheme with the existing techniques. But when the dataset turns into multi-dimensional, it causes significant difference, due to the fact that the original dataset cannot be index in the same way as in one dimensional case. The introduction of Grid File is for solving the sorting problem. To build the authentication structure, the data owner needs to create the Grid File index first, then generates the parity coding among the Grid cells. Here, we will show the performance result of our technique in multi-dimensional case, in terms of preparation cost, authentication cost, correction cost.

Preparation Cost. The result is displayed in Figure 5.4 (a). To avoid the special case, we use 7 servers in total, and have two configurations in the maximum number of malicious servers: $e = 1$ and $e = 2$. In this simulation, the data size increase from 2^{10} to 2^{17}. From the two sets of output, we can see that there is no very obvious difference in the beginning. But with continuous accretion of the data size, it proves the conclusion that tolerating more malicious servers lead to more preparation cost.

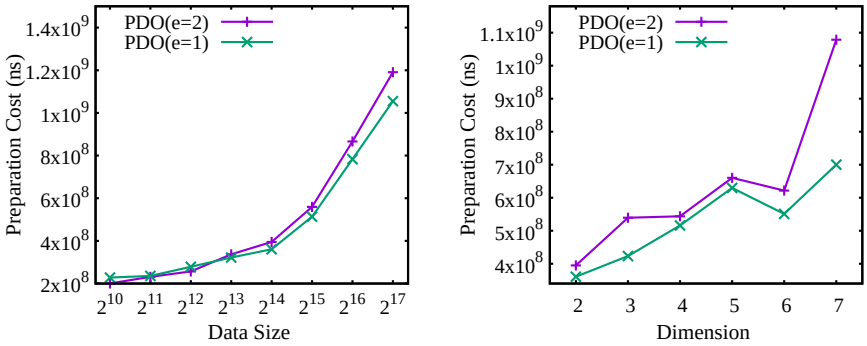


Figure 5.4: Preparation Cost

To study the impact of the dimension on the preparation cost, we set the data size 2^{14}, and the dimension varies from 2 to 7. The result is displayed in Figure 5.4 (b). As the number of dimension increases, on the one hand, it incurs more computation time in generating the parity data; On the other hand, building the Grid File index contains randomness in partition the Grid cells.

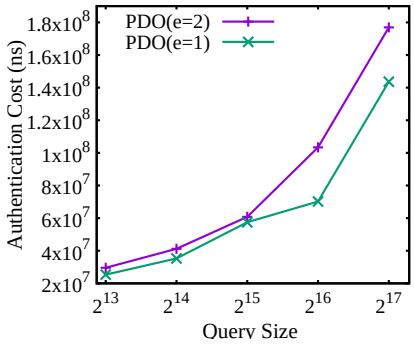


Figure 5.5: Authentication Cost

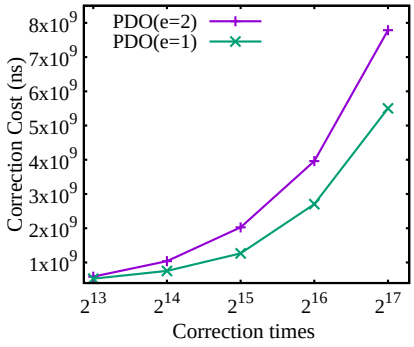


Figure 5.6: Correction Cost

Authentication Cost. To retrieve the query result, the users first need to find out the MBC of the window query, then authenticate the MBC. If MBC is correct, then for each candidate cell, the users send the boundary of the cell (used as window query) to each server. Considering that the first step is similar to the process in one dimensional case, we only test the time cost in the second step, which authenticates the sub-result in each candidate cell. The authentication result is displayed in Figure 5.5. The query size varies from 2^{13} to 2^{17}. Basically, the time cost increases as the query size increases.

Correction Cost. Similar to the testing in authentication cost, we only consider the second step. To correct a codeword, the users need to find out the most popular solution. However, in reality, if a server is marked as malicious, then the returned data is just removed from the equation system. This implies that the correction procedure does not need any more for the remaining codewords, once the malicious server is confirmed. Besides, the number of equations in the equation system reduced accordingly. In this simulation, to test the average time for correction, we repeat the correction operation on each codeword received. The result is displayed in Figure 5.6. We use the query size as the correction time from 2^{13} to 2^{17}. The time increases almost in a linear increasing manner.

CHAPTER 6. CONCLUSION

We have described a Parity-based Data Outsourcing (PDO) model. In contrast to existing techniques that use cryptography and store data on a single server, our approach is to pair the raw data with parity data and store them on a number of servers from different vendors. We show that this approach allows data users to verify if the query results they receive are indeed sound and complete, but also correct if any data is wrong. Our model has a few other desired features. 1) PDO is fault-tolerant. Unless more than e servers are malicious, users can always have access to their data and receive correct query results. 2) PDO is simple and practical. Customers simply upload a smaller database to a cloud server and query it with standard SQL. There is no need for them to develop and/or install any extra software. 3) PDO is cost-effective and load balanced by its nature. The extra storage overhead is the parity data. In our implementation, each server is allocated with a similar size of data. The communication cost of receiving a query result has a similar redundant rate. While the original PDO is designed to support one-dimensional data, we have extended it for multi-dimensional data. We have implemented our techniques and two existing techniques, namely, MH-tree and Signature Chain, and studied their performance through both analysis and experiments.

BIBLIOGRAPHY

[1] Heroku Postgres Free Cloud Data Management Services. Web page at https://www.heroku.com/postgres.

[2] Openshift cloud hosting. Web page at https://www.openshift.com.

[3] B. Thompson and S. Haber and W. Horne and T. Sander and D. Yao. Privacy-Preserving Computation and Verification of Aggregate Queries on Outsourced Databases. *Privacy Enhancing Technologies: Lecture Notes in Computer Science*, 5672:185–201, 2009.

[4] S. Bajaj and R. Sion. Correctdb: Sql engine with practical query authentication. *The VLDB Journal*, 6(7):529–540, 2013.

[5] Elwyn R. Berlekamp. *Algebraic Coding Theory.* McGraw-Hill, New York, NY, 1968.

[6] David Chase. Code combining–a maximum-likelihood decoding approach for combining an arbitrary number of noisy packets. *Communications, IEEE Transactions on*, 33(5):385–393, 1985.

[7] Weiwei Cheng, HweeHwa Pang, and Kian-Lee Tan. Authenticating multi-dimensional query results in data publishing. In *Data and Applications Security XX*, pages 60–73. Springer, 2006.

[8] Cheng, W. and Tan, K. Authenticating kNN Query Results in Data Publishing. *Secure Data Management: Lecture Notes in Computer Science*, 4721:47–63, 2007.

[9] P. Devanbu, M. Gertz, Martel, and Stubblebine C. Authentic third-party data publishing. In *Fourteenth IFIP 11.3 Conference on Database Security*, 2000.

[10] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic data publication over the internet. *Journal of Computer Security*, 11(3):291–314, 2003.

[11] G David Forney. Generalized minimum distance decoding. *Information Theory, IEEE Transactions on*, 12(2):125–131, 1966.

[12] G.D. Forney. Authentication of k nearest neighbor query on road networks. *IEEE Transactions on Information Theory*, 11:549 – 557, Oct 1965.

[13] Daniel Gorenstein, W Wesley Peterson, and Neal Zierler. Two-error correcting bose-chaudhuri codes are quasi-perfect. *Information and Control*, 3(3):291–294, 1960.

[14] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[15] Jonathan Katz and Yehudi Lindell. *Introduction to Modern Cryptography: Principles and Protocols*. Chapman and Hall Cryptography and Network Security Series. Chapman and Hall, 2007.

[16] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of SIGMOD'06*, pages 121–132. ACM, 2006.

[17] James L Massey. Shift-register synthesis and bch decoding. *Information Theory, IEEE Transactions on*, 15(1):122–127, 1969.

[18] Ralph C Merkle. A certified digital signature. In *Advances in Cryptology – CRYPTO'89 Proceedings*, pages 218–238. Springer, 1990.

[19] D. E. Muller. Application of boolean algebra to switching circuit design and to error detection. *IRE Transactions on Electronic Computers*, 3:6–12, 1954.

[20] Ben Noble and James W Daniel. *Applied linear algebra*, volume 3. Prentice-Hall New Jersey, 1988.

[21] HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of SIGMOD'05*, pages 407–418. ACM, 2005.

[22] HweeHwa Pang, Jilian Zhang, and Kyriakos Mouratidis. Scalable verification for outsourced dynamic databases. *Proceedings of VLDB'09*, 2(1):802–813, 2009.

[23] W Wesley Peterson. Encoding and error-correction procedures for the bose-chaudhuri codes. *Information Theory, IRE Transactions on*, 6(4):459–470, 1960.

[24] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics (SIAM)*, 8(2):300–304, 1960.

[25] Madhu Sudan. Coding theory: Tutorial & survey, 2001.

[26] Yasuo Sugiyama, Masao Kasahara, Shigeichi Hirasawa, and Toshihiko Namekawa. A method for solving key equation for decoding goppa codes. *Information and Control*, 27(1):87–99, 1975.

[27] Yuzhe Tang, Ling Liu, Ting Wang, Xin Hu, Reiner Sailer, and Peter Pietzuch. Outsourcing multi-version key-value stores with verifiable data freshness. In *Proceedings of ICDE'14*, pages 1214–1217. IEEE, 2014.

[28] Yuzhe Tang, Ting Wang, Xin Hu, Jiyong Jang, Ling Liu, and Peter Pietzuch. Authentication of freshness for outsourced multi-version key-value stores, 2013.

[29] Shixin Tian, Ying Cai, and Zhenbi Hu. A parity-based data outsourcing model for query authentication and correction. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 395–404. IEEE, 2016.

[30] X. Lin and J. Xu and H. Hu and W. Lee. Authenticating Location-Based Skyline Queries in Arbitrary Subspaces. *IEEE Transactions on Knowledge and Data Engineering*, 26(6):1479 – 1493, 2014.

[31] Guolei Yang, Ying Cai, and Zhenbi Hu. Authentic publishing of mathematic functions. In *review by ICDE'16*, 2016.

[32] Yin Yang, Stavros Papadopoulos, Dimitris Papadias, and George Kollios. Spatial outsourcing for location-based services. In *Proceedings of ICDE'08*, pages 1082–1091. IEEE, 2008.

[33] Yin Yang, Stavros Papadopoulos, Dimitris Papadias, and George Kollios. Authenticated indexing for outsourced spatial databases. *The VLDB Journal*, 18(3):631–648, 2009.

[34] Yiu, M. and Lo, E. and Yung, D. Authentication of moving kNN queries. In *Proc. of ICDE'11*, pages 565 – 576, April 2011.