Best Practices of Big Data Analytics Applied to PII Security

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Citable link</td>
<td><a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:33825785">http://nrs.harvard.edu/urn-3:HUL.InstRepos:33825785</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA">http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA</a></td>
</tr>
</tbody>
</table>
Best Practices of Big Data Analytics Applied to PII Security

Lori Higham

A Thesis in the Field of Information Technology
for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

November 2016
Abstract

Personally identifiable information (PII) is frequently compromised. This is a crucial issue because compromised PII can and does result in identity theft. Yet the storage and query techniques currently used to process PII facilitate, rather than guard against, breaches to such information. Although research and development of distributed systems has greatly improved the performance of Big Data analytics, the best practices implemented within that context have not been extended to security standards for personal information. As a result, the breach of a single database can expose millions of full sets of PII. In this research, we design a structure of individually secured nodes. Then, taking a set of PII, we strategically group specific elements of personal information together and distribute those subsets across our structure, applying best practices as seen in other fields to PII security. In so doing, we demonstrate how the well-established concepts of data and server isolation can be extended to PII.

Keywords: Personally identifiable information; PII; privacy; breach; data security; data isolation; server isolation; security technologies; Big Data principles
Acknowledgments

Special thanks to Dr. Stratos Idreos, whose many excellent papers not only served as a source of inspiration for my thesis but also provided invaluable insight into encryption, and whose advice guided me throughout the entire process of forming and writing my thesis. In every way, he has been an excellent instructor and coach. I am also grateful to Dr. Jeff Parker, whose quick responses and input served to initially challenge my goals and inspire new focus. In expressing my gratitude, I cannot overlook my instructors whose classes first helped me to realize that PII security could indeed be greatly improved; for that, thank you especially to Scott Bradner and Ben Gaucherin.

Last but not least, thanks to my various family members, all of whom have consistently encouraged me in my endeavor to pursue my studies in the field of computer science—especially my dad, who has always believed in my ability to accomplish whatever I chose to pursue.
# Table of Contents

Abstract .............................................................................................................................. iii
Acknowledgments.............................................................................................................. iv
Table of Contents ................................................................................................................ v
List of Figures ................................................................................................................... vii
Chapter 1 Introduction ........................................................................................................ 1
  1.1 Motivations ............................................................................................................... 1
    1.1.1 Target of Attacks ............................................................................................... 1
    1.1.2 Current Strategies for PII Protection ................................................................. 2
  1.2 Thesis Overview ....................................................................................................... 3
    1.2.1 PII Considerations: Data Types and Workloads ................................................ 3
    1.2.2 Research Methodology ...................................................................................... 4
    1.2.3 Thesis Structure ................................................................................................. 5
Chapter 2 Background ........................................................................................................ 7
  2.1 System Architectures and Database Selection .......................................................... 7
  2.2 Encoding ................................................................................................................. 11
    2.2.1 Integers ............................................................................................................. 12
    2.2.2 Strings .............................................................................................................. 13
    2.2.3 Biometric Data and Images .............................................................................. 16
  2.3 Encryption ............................................................................................................... 16
    2.3.1 Integers ............................................................................................................. 16
    2.3.2 Strings .............................................................................................................. 18
    2.3.3 Biometric Data and Images .............................................................................. 21
  2.4 Chapter Summary ................................................................................................... 23
Chapter 3 Prior Work ........................................................................................................ 25
  3.1 The Principle of Isolation ....................................................................................... 25
    3.1.1 Specific Focus ................................................................................................. 25
List of Figures

Figure 1. PII as target in hacking/malware attacks ............................................................. 1
Figure 2. Sensitive PII ........................................................................................................ 3
Figure 3. Strengths and weaknesses of major systems ....................................................... 8
Figure 4. Full record insert ............................................................................................... 34
Figure 5. Criminal history updated ................................................................................... 35
Figure 6. Last name updated ............................................................................................ 35
Figure 7. Select range scan ............................................................................................... 36
Figure 8. Cost model ........................................................................................................ 37
Figure 9. Select criminals and group by city ................................................................. 38
Figure 10. Constant group-by cost .................................................................................. 39
Figure 11. Security versus overhead ............................................................................... 47
Chapter 1  Introduction

1.1 Motivations

1.1.1 Target of Attacks

In 2015, the account information of 80 million individuals was stolen via Anthem's massive data breach, with hackers gaining access to names, birth dates, medical IDs, and social security numbers, among other data (Weise, 2015). The previous year, U.S. government databases holding personnel records and security-clearance files for more than 22 million people were breached (Nakashima, 2015). These and other massive data leaks demonstrate both the frequent target of attacks and the problematic structure of current data systems used to store personally identifiable information (PII).

![Figure 1. PII as target in hacking/malware attacks](image)

Source: Huq, 2015, p. 6
Given both the high value that hackers evidently place on PII and the obvious failures to protect it, in this thesis, we review options for a distributed system that stores subsets of PII across individually secured nodes. This is important to ensure that a single server breach cannot result in the exposure of full sets of PII. Such a system must also include a secure encryption scheme such that sets of data retrieved through queries remain encrypted, with only the authorized client able to decrypt a set and view the values.

1.1.2 Current Strategies for PII Protection

Homeland Security (2011) wrote a fact sheet designed to help safeguard PII in paper and electronic form, outlining how to collect, access, use, share, and dispose of PII. In requiring proper storage, it was noted that "sensitive PII may only be saved, stored, or hosted on DHS-approved portable electronic devices (PEDs), such as laptops, USB flash drives, and external hard drives, all of which must be encrypted" (p. 2). In addition, several organizations have defined the various data elements considered PII, with requirements to store incident documents at a secure facility, lock laptops containing PII, and wipe rental computers before returning them to vendors (General Services Administration, 2014; Homeland Security, 2012; McCallister, Grance, & Scarfone, 2010; NRCG, 2012).

Although these standards do require that electronic devices, files, and attachments containing PII be encrypted, to our knowledge there is no detailed guidance on what proper encryption and storage of the data itself should entail. For example, we find no mention of guidelines for encrypting individual components within a set of PII, strategies for storing subsets of these elements across nodes, or even a standard on encrypted query
processing for retrieval of such data. Thus, it seems there is neither a sound, multilayered approach to PII security in place nor any real provision for protecting the data in the event that a database server is actually compromised.

1.2 Thesis Overview

1.2.1 PII Considerations: Data Types and Workloads

To ascertain which literature on database systems, encryption and storage methods, and other strategies are most pertinent to our research, we examine the data types and workloads that a data system needs to handle. Although PII records contain many data elements, not all are considered sensitive. It makes sense to consider national standards in evaluating which data elements should be encrypted individually.

![What is PII?](image)

*Figure 2. Sensitive PII*


The question of which elements to group together and distribute as subsets across nodes is one we consider later. Regardless of subset groupings, all PII elements marked as sensitive by Homeland Security need to be encrypted, which means we must consider
encryption for integer, string, and image data types. Not all PII is retrieved for every type of query, but when retrieved, each of the above elements must remain encrypted throughout the process.

We also consider the types of workloads a data system needs to handle. The storage and query processes in this case would consist only of disseminating and aggregating related data, with no support required for complex analytics tasks. Our objectives are to encrypt and encode individual components; compartmentalize subsets of data and distribute them securely across nodes; and execute queries over the encrypted data. The background that would best serve as a foundation for our system, then, includes literature and systems demonstrating sound encryption and encoding methods for integer, string, and image data types as well as clear comparisons of the various types of database systems.

1.2.2 Research Methodology

In this thesis, we highlight the problem of frequent PII exposure through data breaches and propose a solution that extends concepts already well established in Big Data analytics to PII security. Such concepts include encoding and encryption of individual PII components. Proper encryption, maintained not only during storage but also throughout the query process, prevents both eavesdropping by malicious adversaries and snooping by the server itself. Existing literature clearly supports the need for data encryption, and in our recommendations for database design, we assume such foundations are in place. Data isolation, implemented by distributing PII subsets across nodes, further supports security in that even if a server/database is compromised and encryption somehow fails, the entire set of PII is not exposed.
Because the increased security measures described above could introduce prohibitive overhead such as slow query performance and increased storage requirements as a side effect, we also recommend encoding some of the encrypted data (another concept found in Big Data analytics). As our paper demonstrates, we find evidence from the literature that the strategies outlined here could greatly enhance PII security while maintaining good performance during query processing.

This thesis is not intended to serve as the final word on PII security but rather aims to apply previously developed Big Data analytics concepts to the storage and retrieval of PII and then contribute database design options for enhanced data security that could be easily implemented and monitored with existing technology. Our hope is that this paper will serve as inspiration for later developments in this field.

1.2.3 Thesis Structure

In Chapter 2, we examine background issues that should be in place in a multifaceted approach to data security, noting existing contributions related to database selection, encoding, and encryption as a foundation. We freely acknowledge that no single layer of security, including the well-established concept of data and server isolation that we extend to PII in later chapters, is sufficient to achieve data security, and we recommend careful application of each protective measure described in this chapter. Readers with such foundations already in place may wish to skip to the next chapter, but we sketch an outline of these concepts, as could be applied to PII, for those interested.

In Chapter 3, we present a brief overview of existing works that clearly demonstrate the principle of data and server isolation, which is the main focus of our paper. This is the concept that we extend to PII in some detail. In Chapter 4, we present
three database design options, strategically dividing a limited set of data into subsets that could be distributed across various nodes. Chapter 5 outlines our experiments, and in Chapter 6, we discuss the results. Finally, Chapter 7 presents the summary and limitations, with a focus on future work.
Chapter 2  Background

In this thesis, we assume the implementation of the best security measures found in existing systems and in the literature. We base this research on the foundation of an appropriate database system, sound encryption standards, and well-established encoding principles. Thus, before moving to our contribution, in which we apply the principle of data and server isolation to PII, we carefully present an overview of these equally important concepts in the field of cybersecurity.

2.1 System Architectures and Database Selection

One method of categorizing database systems is four-system classification: Classic Parallel, Columnar, MapReduce, and Dataflow (Babu & Herodotou, 2012). A few examples of each classification are listed below.

- **Classic parallel**: DB2 Parallel Edition, Greenplum, Teradata
- **Columnar**: Amazon RedShift, MonetDB, Vertica
- **MapReduce**: Hadoop, Hive, Pig
- **Dataflow**: Pregel, Spark, Dremel

We compare some of the known strengths and weaknesses of each system type, seen in the figure below:
<table>
<thead>
<tr>
<th>Source</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classic Parallel (Row)</td>
<td>- Read and write optimized</td>
<td>- Not scalable</td>
</tr>
<tr>
<td></td>
<td>- Ideal for frequent transactions</td>
<td>- Not well suited for queries, especially against huge databases</td>
</tr>
<tr>
<td></td>
<td>- Excels at managing and processing structured relational data</td>
<td></td>
</tr>
<tr>
<td>Columnar</td>
<td>- Scalable</td>
<td>- For write operations, tuples must be split into component attributes, with each attribute written separately</td>
</tr>
<tr>
<td></td>
<td>- Fast query support</td>
<td>- Compressed data layout increases difficulty of making insertions to tuples within a disk block</td>
</tr>
<tr>
<td></td>
<td>- Good compression</td>
<td>- Not well suited for transactions (import and export speed issues)</td>
</tr>
<tr>
<td></td>
<td>- Good join performance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Good indexing</td>
<td></td>
</tr>
<tr>
<td>MapReduce</td>
<td>- Scales linearly</td>
<td>- No built-in support for indexes</td>
</tr>
<tr>
<td></td>
<td>- Excellent load times</td>
<td>- High parsing overhead</td>
</tr>
<tr>
<td></td>
<td>- Good fault tolerance</td>
<td>- Poor compression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Not ideal for joins</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Master node can be single point of failure</td>
</tr>
<tr>
<td>Dataflow</td>
<td>- Works well with independent distributed file systems</td>
<td>- Complex structure, with layers nested within or built on top of the typical distributed file systems; can be complicated</td>
</tr>
<tr>
<td></td>
<td>- Can use automatic resource management and performance optimization</td>
<td>- Many Dataflow-type software stacks still use MapReduce elements</td>
</tr>
<tr>
<td></td>
<td>- Can function in both batch mode and real time, easily moving, processing, and analyzing massive amounts of info</td>
<td>- Often language specific</td>
</tr>
<tr>
<td></td>
<td>- Varied storage and query options</td>
<td>- Still very new</td>
</tr>
</tbody>
</table>

**Figure 3. Strengths and weaknesses of major systems**

Sources: Babu and Herodotou, 2013; Cloud Dataflow, 2015; Madden, 2009; Walker, 2012

As seen in Figure 3, the Columnar database system is more scalable and better suited to queries than its counterpart, the Classic (row-based) Parallel database system. Although write operations and transactions are easier with the Classic Parallel system (Walker, 2012), a number of the most sensitive elements—PII (e.g., SSN, DOB, mother's maiden name)—rarely need to be changed and are not characterized by transaction-type operations, while frequent queries are an essential function within our system. In addition, the design should be scalable given that the amount of PII may increase or decrease at any time. Finally, Columnar systems have much better join performance than Classic Parallel systems, as noted in Figure 3.
In considering MapReduce systems, we are likewise concerned about the issue of join performance and about the fact that MapReduce is not as well suited for indexing or compression (Madden, 2009).

Finally, in reviewing the Dataflow systems, we note that many of this type of architecture's main selling points (e.g., ability to move and process huge data streams, with support for complex analytics tasks) are generally irrelevant to our task at hand—to implement a well-defined, compact, relational system in which PII elements can be securely stored and retrieved. In addition, many Dataflow-type software stacks still use MapReduce elements. The storage structure of Dataflow systems is often complex, with layers nested within or built on top of the typical distributed file systems; however, nesting and obfuscation of distributed file systems does not make them entirely insusceptible to infiltration by malicious adversaries. Thus, sound encryption and storage strategies as well as query processing considerations are the most crucial factors in terms of security, regardless of the system selected.

We are impressed with the processing method of Dataflow systems. Although the system structures vary, the processing of data "follows a common theme and consists of (a) a set of computational vertices that define local processing on a partition of the data, and (b) a set of communication edges that define data transfers or transformations between the vertices" (Babu & Herodotou, 2012, pp. 63–64). The overall processing concept adheres to graph theory principles and might one day be the best option for secure storage and retrieval of data, especially if combined with the capability-based principles suggested in "A Survey of Distributed Capability File Systems and Their Application to Cloud Environments" (Naval Postgraduate School, 2014). Currently,
however, the Dataflow system structure is somewhat new, has complex architectures, and is designed primarily for good performance in batch and streaming data operations; our focus is the security of relational data in individual, fairly simple queries. For that reason, we have narrowed our options to Columnar database systems.

Various popular options exist among Columnar database systems, such as Amazon RedShift, Vertica, Infobright, and MonetDB (Amazon Web Services, 2016; Lamb et al., 2012; Ślęzak & Eastwood, 2009; Idreos et al., 2012). We limit our options to open-source databases, however, because (a) an open-source system can be adapted to the needs at hand, while commercial systems cannot be freely altered and (b) given that its code is open to public critique and improvement, an open-source system may be better tested and developed from a wider range of perspectives. After careful consideration of Infobright, C-Store, and others, we ultimately recommend MonetDB for those with extensive experience and knowledge of that system, but MySQL as the otherwise more practical option due to its wide use and popularity (DB-engines, 2015), especially given its existing implementations (e.g., AWS's RDS DB engines, MySQL Workbench) that lend well to visual demonstrations.

Some of the features we particularly like about MonetDB are the following:

- Supports remote tables (MonetDB, 2015)
- Supports foreign keys, secondary indexes, joins, triggers, and stored procedures (DB-engines, 2015; MonetDB, 2015)
- Converts relational algebra and SQL into columnar algebra in keeping with its layout (Babu & Herodotou, 2012; Idreos et al., 2012)
- Uses order-preserving compression techniques (Idreos et al., 2012)
- Supports late tuple reconstruction (Babu & Herodotou, 2012)
- Supports database cracking (Idreos, 2010)
• Supports automatic indexing and run-time query optimization (Boncz et al., 2006)
• Supports array processing and storage through SciQL (Babu & Herodotou, 2012)
• Runs on a rich spectrum of operating systems and with numerous programming interfaces (DB-engines, 2015; MonetDB, 2015)
• Accommodates various query languages through its proprietary algebraic-language, MonetDB Assembly Language (MonetDB, 2015)

Although MonetDB may lend well to more extensive use of algorithms (MonetDB.org, 2015) and has more options for joins (Graphiq, 2014), it is similar to MySQL, overall, in many ways (Graphiq, 2014). An exhaustive review of other open-source options is beyond the scope of this research, but it is notable that not all run on multiple operating systems, support triggers and secondary indexes, or use order-preserving dictionary-based compression techniques to the extent that both of these systems do. For our demonstrations, we use MySQL due to the visual metrics that already exist in AWS's MySQL implementation (seen in RDS) and in MySQL Workbench.

2.2 Encoding

Database management systems can build indexes for ciphertexts the same way they do for plaintexts; by extension, comparisons and various other operations can be performed between encrypted data in the same way as between plaintexts (Idreos, 2016; Popa, Redfield, Zeldovich, & Balakrishnan, 2011). Thus, encryption does not necessarily affect data compartmentalization and storage strategies.

Conversely, the encoding schemes used for data storage greatly affect both compartmentalization and storage, which in turn relate to joins and the overall workloads. Furthermore, because the data type for a given PII element can sometimes vary, the
selection of encryption methods can be affected by compartmentalization strategies. Thus, encryption, encoding, and compartmentalization all require careful consideration to ensure efficient implementation of any of them.

As an example of an inconsistent PII element, in some states the driver's license number contains only integers, whereas in others it consists of both letters and numbers. The same issue occurs with alien registration numbers, which can serve as an alternative to the social security number (SSN) for immigrants. Because of the string portion in some of the "numbers," not all can be simply stored as integers. One possible solution is to isolate the string and integer components from each other and use different methods of encryption for each. The encryption strategy itself can vary based on whether or not the elements are to be indexed, among other factors, which in turn can relate to encoding. Therefore, we first separate the various data into integer, string, and image data types and examine encoding and compartmentalization options for each before considering the encryption strategies for each type of PII element.

2.2.1 Integers

Integers typically require less storage space than string or image data types. However, among the integers found in PII are lengthy integers that might be logically subdivided into more manageable units. For example, the first three digits of a SSN represent the area in which the number was assigned, the next two digits represent the group number, and the final four digits are the unique identifier within a given area/group number, so the larger SSN integer structure can be subdivided into three units, with each stored separately. The same basic concept can be applied to the date of birth (DOB).
In both cases, however, the integer encryption methods to be used greatly affect whether or not encoding should be considered for this data type. For instance, in the above example, the subdivided integers can be stored using dictionary encoding as described in Chen, Gehrke, and Korn (2001), reducing overhead. However, to encode integers in this way could result in ordered storage, meaning that if one element is accessed, information can be leaked about the adjacent elements based on their location. For this reason, and because integers require less storage space than the other data types we consider, we opt to avoid encoding integers in this case.

2.2.2 Strings

In "Query Optimization in Compressed Database Systems," Chen et al. (2001) discussed dictionary encoding, in which values that would normally consume many bytes are encoded in only a few bits. To reduce overhead, we recommend this principle for storing skewed string PII components (e.g., last name, city, criminal charge, sentence). Lee et al. (2014) expanded on the same concept, discussing how to retain encoding throughout the join process. They also examined the uses of Raman's et al. (2008) frequency partitioning scheme, in which more frequent values are encoded in fewer bits, and less frequent values are encoded in more bits. Such a scheme employs fixed-length codes and per-column encoding, building a columnal, partitioned histogram of the value occurrences according to value frequency. Then all values in a given partition are encoded using the same number of bits (Lee et al., 2014). A simple case of skewed data with frequent repetitions is mother's maiden name; we view the strategy described above as optimal for such data.
Among the elements listed as "always sensitive" by Homeland Security (2012, p. 5), the data type most obviously affected by encoding strategies is string data, but not all cases of string data should be handled the same way. In some cases, in fact, what first appears to be unmanageable string data should be broken down and compartmentalized (e.g., criminal history and medical information). For example, with regard to whether a given individual has a criminal history, yes and no can be encoded in one bit. For cases of existing criminal history, the standardized descriptions of the charges are skewed, with frequent repetitions (e.g., "assault with deadly weapon," "malicious bodily injury," or "armed robbery"), and thus can also employ the frequency partitioning scheme described above for mother's maiden name. Sentencing records (as part of criminal history) tend to use standardized terminology as well (e.g., "6 months county jail," "fines and restitution," or "3 years formal probation"). Importantly, both charge and sentencing components might be logically subdivided; some terms, such as "6 months," include both an integer and a string value and can be compartmentalized accordingly. Even text strings that do not include an integer should be split into keywords, after which the frequency partitioning scheme can be applied within the various nested trees.

The date of offense can be stored as an integer, with months, days, and years subdivided and encoded, and the three options for plea can be encoded in two bits to represent guilty, not guilty, or no contest.

Criminal reports as retrieved through queries often do not include images (Evictionrecords.com, n.d.); this is significant as relates to joins and queries, discussed later. Nonetheless, the storage of an entire criminal history requires many additional partitions not discussed in this paper (e.g., court notes, case numbers, disposition,
photos). Essentially, criminal history as a PII element can be subdivided and compartmentalized by data type as its own nested tree within the larger tree of PII, with the same storage, encoding, encryption, and query processing considerations applied.

The same approach can be taken with medical information; predefined sets of medical conditions appear in medical reports as either present or not present for a given individual (one-bit encoding), and standardized strings can be subdivided before encoding the data. As with criminal history, detailed medical information often includes images (e.g., X-ray, MRI); in addition, long notes and detailed information, where necessary or included in a report for query processing, could be scanned, encrypted, and saved as images. In short, medical information can be subdivided and compartmentalized by data type as its own nested tree, applying the same logical partitions and encoding discussed above for criminal history.

One PII string component deserves special consideration: account passwords. As noted in the literature (Coates, 2014; Ragan, 2013), account passwords should be hashed and should never be retrievable through queries. In addition, they are often associated with user accounts rather than with individuals (meaning there is not a one-to-one relationship; a single person may have several user accounts); thus, although they do require secure hashing and storage, account passwords fall outside the scope of PII data to be encoded for fast encrypted query processing. Ample literature exists on modern cryptographic hashes for passwords (Boswell, 2012; Defuse Security, 2014). In this research, we limit our focus to PII data that is intended to be retrieved through legitimate queries; thus, we exclude account passwords from our analysis.
2.2.3 Biometric Data and Images

Although only one category of PII items listed by Homeland Security as "always sensitive" specifically relates to images (biometric identifiers), image data is also found within the nested trees of criminal history and medical information, discussed above. At the present time, biometric and other image data are retrieved in routine queries far less frequently than some other PII elements and thus may not require the same level of join and performance considerations that some other types of data require. For our purposes, secure storage (versus a strong focus on query optimization) is the most important concern for PII components not intended to be frequently accessed. This is reinforced by the fact that most of the strategies related to biometric data that we have found in the literature discuss storing the encrypted key rather than the biometric template itself, and encoding is not generally addressed.

Because biometric data is increasingly used as a personal identifier, we devote a later section to sound encryption and storage techniques; however, we do not include it in the smaller dataset carefully scrutinized for the join performance most often required for noninvasive, routine business queries.

2.3 Encryption

2.3.1 Integers

The greatest randomness and most thorough encryption can be achieved for integers (versus dictionary-based string indexing) through mathematical algorithms; therefore, even aside from the generally lower overhead of integers, we prefer to store sensitive components as integers when possible to achieve greater security. However, integer encryption presents a challenge. Although various hashing and encryption
methods allow arbitrary computations over encrypted data via homomorphic encryption (Gentry, 2009), most rely on an expensive public key cryptosystem (Gentry, Halevi, & Smart, 2012; Popa et al., 2011) and do not allow for indexing of the encrypted data (Idreos, 2016).

In response to the problem of high overhead and the need to index the encrypted data, Idreos (2016) proposed a novel solution for adaptive indexing of encrypted integers. This strategy includes copying each original column into a cracker index column that can be continuously reorganized, using on-demand indexing while disallowing comparisons at the server except to perform carefully circumscribed inequality checks among ciphertexts. Idreos's (2016) solution ensures that information about the order of tuples is always obscured from the server, both at storage and during query processing.

Representing numerical values as short vectors, Idreos's encryption scheme employs simple linear-algebra operations for encryption and decryption. Noise addition, scalar multiplication operations, and matrix multiplication take place on the client side, while the server conducts the other computations as it would do with an unencrypted database. It is important to note that only the client should have access to decryption keys (Idreos, 2016; Popa et al., 2011). This is an important element of maintaining security and is supported through the balanced authority structure. The server processes the queries, initiated from the client side, over the encrypted data and then delivers the encrypted query results to the client. The client then decrypts the data to obtain the actual results (Idreos, 2016).

Aspects that we particularly like about Idreos's solution are that (a) it enables computation over encrypted data; (b) it is lightweight, without prohibitive overhead; (c) it
allows for adaptive indexing; and (d) authority capabilities are distributed between server and client. No superuser or server has unrestricted control over or even access to all capabilities or all data; thus, none can be used by a malicious adversary to gain full control and tamper with the data or processes. As noted by the Naval Postgraduate School (2014), ambient authority—with programs having access to permissions beyond those needed for their own carefully defined operations—tends to enable hackers to "hijack systems, escalate their privileges, and perform their malicious intent" (p. 65).

That problem is avoided through Idreos's (2016) encryption structure. For that reason, we reserve this encryption design for the storage and retrieval of sensitive numerical data.

Details about Idreos's (2016) encryption scheme, including specifics regarding the noise addition, scalar multiplication operations, and matrix multiplication applied to obscure attribute values throughout the storage and query process, can be found in his paper titled "Adaptive Indexing over Encrypted Numeric Data." Importantly, although this scheme was intended specifically for integers, some underlying principles apply to other data types as well.

2.3.2 Strings

Various options exist for executing secure queries over encrypted strings. One is to divide the storage functionality between a block store, which holds file data and most metadata and allows for storage and retrieval by cryptographic hash, and a consistency server. Data security is ensured because the user's files can only be accessed through signed messages that are valid under the user's public key (Li, Krohn, Mazières, & Shasha, 2004). However, this solution does not work well for various systems, including "database-backed websites that process queries to generate data for the user" and
"applications that compute over large amounts of data" (Popa et al., 2011, p. 85). Because these types of applications may need to process PII data, we reject this approach.

Another option is fully homomorphic encryption, which allows arbitrary computations over encrypted data (Gentry, 2009), with only clients viewing the decrypted data (Popa et al., 2011). However, as with integer data, such schemes rely on an expensive public key cryptosystem (Gentry et al., 2012; Popa et al., 2011), which would be prohibitively slow unless the server were given access to the decryption keys, so we reject that option as well.

The encryption system we prefer for string data is CryptDB, which consists of not only a DBMS server but also a separate application server that runs the application code and issues DBMS queries over encrypted data on behalf of the clients (Popa et al., 2011). A key security function within CryptDB's strategy is the interception of SQL queries using a database proxy. The proxy translates the queries so they can be executed on the encrypted data, changing some query operators but not the semantics of the query.

Working through a proxy ensures that the server does not receive the decryption keys and thus cannot gain access to sensitive data. For our system, we want to avoid having superusers or servers with unrestricted control over (or even access to) all capabilities or all data. As noted previously, allowing any entity access to permissions beyond those needed for its own carefully defined operations creates the possibility that a malicious adversary could gain full control of that entity and tamper with the data or processes. CryptDB provides a distributed authority structure while executing queries over encrypted data; thus, it is the system we select to handle string data.

As noted in Popa et al. (2011), some of CryptDB's key strategies are as follows:
• In executing SQL queries, CryptDB uses an SQL-aware encryption strategy that adapts well-defined primitive operators such as equality checks, order comparisons, and joins in such a way that the DBMS executes queries over cryptographically transformed data instead of plaintext.

• CryptDB adapts its encryption scheme based on the SQL queries observed at runtime using what the authors term "onions of encryption" (p. 86). In so doing, it compactly stores nested ciphertexts, avoiding expensive re-encryptions.

• CryptDB chains encryption keys to user passwords so that even if the DBMS and application server are compromised, an adversary cannot decrypt the data unless he or she knows the respective user's password or the user is logged in. It also allows for policy annotations to specify who has access to each data element.

CryptDB describes an "encryption onion" in which it first splits text strings into keywords, with standard delimiters, and then randomly permutes the strings, encrypting each word according to Song, Wagner, and Perrig's (2000) scheme and padding them to the same size (Popa et al., 2011). This isolation of individual keywords found in strings, with each keyword stored separately, facilitates dictionary encoding for skewed strings, as discussed earlier. The overall encryption process, one that CryptDB developed specifically to search for keywords within text data, is called the SEARCH onion.

Although many PII elements can be stored as integers, among the data typically processed for routine queries, (a) the last name or letter element of some states' driver's license structure and (b) various nested elements within criminal history would need to be encrypted, stored, and processed as strings. While Idreos's encryption strategy, which we prefer for integers, relies heavily on mathematical algorithms, CryptDB's solution, our choice for strings, closely correlates its strategies with SQL's well-established query structure. Both solutions employ the concept of multiple encryption layers, and both provide a distributed authority structure in which no server or superuser has access to all
capabilities or all data, protecting against the possibility of an adversary gaining full control and tampering with the data or processes.

2.3.3 Biometric Data and Images

NIST uses PKI encryption to store and retrieve biometric identifiers (NIST, n.d.). Extensive discussions of biometric security can be found in the NIST publications and online (NIST, 2015). Although we do not prefer expensive PKI encryption for biometric identifiers, we do feel that it offers excellent options for storing other types of images not closely associated with personally identifiable features (e.g., X-rays, MRIs, extensive notes) because such images are much less frequently joined with other data (and may never need to be accessed outside of a medical context). Hence, we include NIST’s storage and query process in our study.

Biometric identifiers are verifiable in a way that other types of data are not: by comparison with the live subject. This means the exposure of biometric identifiers does not always equate to malicious adversaries gaining the capability to use those identifiers, making the use of this type of identity verification at least to some degree a protection against identity theft in itself.

That said, in the case of thieves with knowledge of how to exploit the use of a stored (unencrypted) biometric template, the compromise of such data can be much more damaging than that of many other PII components (Cavoukian & Stoianov, 2007; Cherry, 2007). And, importantly, biometric data can be and has been stolen (Peterson, 2015), which is a serious problem because the biometric identifiers themselves, such as fingerprints and iris patterns, cannot be changed. The dilemma, then, is how to store
biometric data in such a way that the breach of a database in which biometric templates are stored does not result in irrevocable access to and use of the biometric data itself.

A logical solution might be to apply hashing to each biometric identifier, storing only the hash values and not the biometric templates themselves, so that compromised templates can be revoked and new ones generated in the case of a database breach. Essentially, then, with this method, a biometrically encrypted key of each identifier, rather than the actual biometric template, is created and stored. This technique is referred to as biometric encryption and has been researched and documented in various articles (Cavoukian & Stoianov, 2007; Cherry, 2007; Soutar, Roberge, Stoianov, Gilroy, & Kumar, 1999).

For example, as explained in "Personal Biometrics, Private Data" (Cherry, 2007), a form of biometric encryption called privID applies a cryptographic hash function along with noise estimation and adaptive analog-to-digital conversion "to reduce both the noise in and size of the template. It then employs error-correction codes similar to those used in scratch-tolerant CDs to give a noise-free representation of the biometric template" (p. 6). The author further discussed the importance of inserting auxiliary information into the template before hashing it (with a result resembling a salted hash) to enable revocation and replacement of any compromised (hashed) identifier. In this strategy, the underlying concept safeguarding the data is simply that it is not the biometric identifier alone that is encrypted but rather the biometric identifier plus added components. Thus, in the event of a breach, simply changing the added components changes the template.

There are other versions of the same basic concept as well, with earlier strategies combining the biometric image with a digital key to enable (a) continued use of the
biometric if the key is compromised and (b) modification of the key, if required (Soutar, Roberge, Stoianov, Gilroy, & Kumar, 1999).

Keeping in mind the underlying concepts of the various approaches, a combination of noise reduction, salted hash functions, and error-correction codes, as mentioned in Cherry (2007)—combined with storing only the encrypted key associated with the biometric identifier and not the identifier itself—is the method we recommend to encrypt biometric identifiers and protect against breaches such as that which occurred in the Office of Personnel Management (Peterson, 2015). Additional details about this type of storage and query process can be found in the articles referenced earlier.

With regard to images as found within criminal history and medical information, pictures of facial structure or other identifiable features could be stored using the same principles outlined above for biometric identifiers. Other types of images that describe but do not identify the individual (e.g., X-rays, longer explanatory notes) could be scanned and stored using PKI encryption because, as noted earlier, they rarely need to be accessed outside of a medical or government context and thus may not require query optimization. Conversely, while biometric identifiers are not currently included in routine queries, they are increasingly used in access control, payment systems, and attendance recording (Cavoukian & Stoianov, 2007) and thus need to be securely accessible for legitimate purposes.

2.4 Chapter Summary

In this chapter, we considered the various categories of databases and reasons for specific database selection. We also explored both the data types found in PII and the types of encoding and encryption that might logically be applied to each. In addition, we
noted that although every element of data deemed "sensitive" by Homeland Security should be encrypted individually, not all sensitive PII is retrieved for every type of query. This is an important point because our own contribution relates primarily to those data retrieved regularly for routine business queries; thus, our design considers only a limited set of PII.

As with other concepts seen thus far—such as in CryptDB's onions of encryption and in Idreos's use of noise addition, scalar multiplication operations, and matrix multiplication to obscure attribute values—PII security should consist of multiple layers. In considering the background described in this chapter, we primarily seek to establish that guidelines do exist for database selection, sound encoding and encryption schemes, and distributed storage in general, with principles that can and should be applied to PII security. Moving to our own contribution, we assume that all of these security measures are in place. Our focus in later chapters is then one layer of a multifaceted approach to security: a database/node design structure that strategically separates subsets of PII so that data exposure is minimized in the event of a breach.
Chapter 3  Prior Work

3.1 The Principle of Isolation

The concept of data isolation as a security measure is well established. This is true for data in all of its states, whether at rest, in use, or in transit. In the literature, more attention may be devoted to isolation of data at rest and data in transit than to isolation of data in use (Colley, 2016; Microsoft, 2009; Vengurlekar, 2012); however, Cloud computing by its very nature can blur the lines between the various states. For example, although the reason for using a virtual machine (VM), container, or combination of both (e.g., Intel, 2016) may initially be isolated storage of a particular application (Zomaya, 2016), a side benefit includes isolation of the associated data while it is in use as well.

3.1.1 Specific Focus

Although isolation is worth considering for data in all of its states, in this paper, our primary focus is secure, isolated storage of data at rest, specifically through server isolation. This concept is established as a security measure not only as analyzed in the literature (Harris, n.d.; Microsoft, 2009; Northcutt, 2009) but also in storage architectures themselves (Erl, 2016). Various strategies exist for achieving server isolation. One method involves domains and group permissions as well as secured, authenticated connections, briefly described later with regard to Microsoft's Active Directory domain. Another method is to isolate a server from the traffic of other virtual servers through the use of an additional virtual switch, as illustrated in Erl (2016). These strategies, among
others, essentially restrict the isolated server's connections to only those secured, authenticated networks with which the server actually needs to communicate.

3.1.2 Existing Implementations of Isolated Data at Rest

Several works have examined specific implementations of storage isolation as a significant factor in security. For example, Elish, Deng, Yao, and Kafura (2013) described device-based isolation whereby the secrecy of highly sensitive data such as private keys is protected. This strategy ensures that cryptographic keys are maintained in a dedicated device kept separate from the general computing environment. Whereas some other storage methods allow hosts to request the unencrypted key value itself, this method never allows the private key to be sent to another host; only a message signed with the protected private key may be requested and retrieved. By maintaining device-based isolation, never passing the clear-text keys to any other host, the potential for a breach is greatly limited.

Microsoft detailed the security benefits of server isolation, describing the role of Active Directory in limiting specific server access to domain members (Harris, n.d.; Microsoft, 2009). This strategy essentially utilizes network policies to isolate servers that contain sensitive information from non-domain-member computers. An additional layer of security then limits server access even among domain members to only those users that belong to a particular security group.

Other security solutions exist as well, some of which include the use of IPSec and group policy (Microsoft, 2005) or isolated PVLANs (Cain, 2013). As with the earlier examples, the underlying principle is to restrict the isolated network or server's connections, thus limiting the attack surface.
3.2 Extension of Existing Implementations

While these and other works provide a foundation that clearly depicts the same concept we discuss in this paper—the security benefits of server isolation—we extend this principle beyond isolation by data category. Rather than store an entire dataset on a single server and attempt to isolate that server, which, if somehow breached, could still expose all associated sensitive data, we divide the data into subsets and then distribute those subsets as partial records across multiple servers. Our strategy is to thereby minimize data exposure to only a subset rather than an entire set of PII in the event of any single server breach. Another difference between our contribution and prior work is the underlying assumption of each. Prior work seems to rely on the assurance that a particular server can never be breached, whereas our assumption—and as a result, our approach—is that indeed any server might be breached. Our aim is to therefore plan for the worst, distributing related data across multiple individually protected servers so that when a server is breached, only a subset of any individual's PII is exposed.
Chapter 4  Design

4.1 Preliminary Considerations

In implementing a design to distribute subsets of PII across multiple individually protected servers, we first selected the specific PII elements to use. We limited our study to the personal information that is typically retrieved for routine business queries. Second, we strategically divided that set of data into subsets that can be distributed over various nodes. The purpose of isolating small groups of PII, and in so doing, separating specific PII elements from each other, was to mitigate the risk posed by data aggregation, which occurs when multiple pieces of information are combined (Cobb, 2011). We also included in our design plan an additional node in which to isolate the stored procedures whereby we would populate and update our databases, create searches, and benchmark the performance of each algorithm and query over the various database structures.

For the purpose of this study, based on the literature (Bassett, 2011; Thompson, n.d.), we associated an individual with his or her name, address, SSN, DOB, state ID, criminal history, and a few other associated elements for routine business/employment queries. Having selected these specific personal information components for our study, we created three designs, two of which demonstrate data isolation by maintaining separation between subsets of PII. An additional design included all PII elements in a single node but maintained separation between the node storing the database and the one in which we kept our stored procedures.
4.2 Design Options

4.2.1 Three-Node Design

In Option 1, the three-node design, we isolated the data in subsets as follows.

- **Node 1**: first name, last name, and address
- **Node 2**: ID (SSN or alien registration number), ID status, and criminal status (and related info in the case of a criminal record)
- **Node 3**: state ID, state ID status, and DOB

In addition to these basic data elements, each node maintained create_time and update_time fields for integrity checks. Finally, a hashID field was common across all nodes, with each hashID unique to a single record while serving as a join column for stored procedures, statements, and queries. The strategy for this three-node design was to exclude PII from the database storing the names and locations of individuals because a number (SSN, DOB, state ID) means less when not attached to a name. This strategy also separated the DOB from the SSN.

In the above three-node design, we grouped the DOB with the state ID, given that an individual's DOB appears directly on his or her state ID card. Just as the DOB is verified whenever a state ID is required, the same verifications would tend to occur in a query or computer check of the state ID, so it made sense to store those two pieces of PII together.

Importantly, in many scenarios of a state ID check, the SSN is not required. On the other hand, for employment as a self-employed contractor, the SSN is always queried and usually accompanies a criminal background check, so we stored the SSN and criminal history items together. It is worth mentioning that queries requiring only the
SSN but not the state ID or DOB (or vice versa) could be set to search only two of the three nodes, reducing the amount of PII accessed in the query process.

4.2.2 Two-Node Design

For Option 2, the two-node design, we eliminated node 1 and added its contents—first name, last name, and address—to the remaining two nodes. The argument for this option was that in legitimate searches of an SSN or state ID, the associated individual's nonsensitive personal information is already known, so eliminating the nonsensitive information as a separate node could reduce both the number of connections required and CPU utilization.

On the other hand, in full-record queries, the duplication of three string fields (first name, last name, and address) in two nodes is inefficient and increases overhead. However, this structure did create two smaller subsets of PII that could each function as a one-node option for searches that, in scenarios such as those described in Section 4.2.1, require only the SSN and related info, or only the state ID and related info, but not both.

4.2.3 One-Node Design

Option 3 stored all personal data in one node. Option 3a simply included all of the personal info listed above for the three-node design in a single node, while Options 3b and 3c were equivalent to the two nodes described in the two-node design, each including only the information necessary for verification of select PII elements, as already detailed above.
4.3 Summary

Although we term these designs the one-node, two-node, and three-node options, each design requires \( n + 1 \) nodes because, as noted previously, we use an additional, separate node for the storage of all transaction and procedure algorithms. This is in keeping with the Big Data principle of data isolation to the greatest extent possible as a standard security measure.

Even before any tests of the designs, it was obvious that some increase in overhead would occur with every node addition. First, to correctly create and run queries and stored procedures over more than one node simultaneously, and to easily perform integrity checks, we duplicated the create\_time, update\_time, and hashID fields across nodes. This meant three additional columns would be searched \( n - 1 \) extra times for every added node. Second, each node addition increased not only the number of nodes but also the overall system size, number of connections, and possibly time and CPU utilization.

On the other hand, an important factor in data security is data isolation, which was the primary purpose in separating the name and address from the associated PII and of separating certain PII elements from each other by storing them across nodes. It is worth mentioning that not only can each database be individually encrypted and password protected, but each individual server can be uniquely secured as well, using a hardware security module (HSM) and other state-of-the-art security measures.
Chapter 5  Experiments

5.1 Setup

To analyze performance and cost, we tested our designs over Amazon's AWS preconfigured MySQL default engine, using RDS micro-instances with two vCPUs, 4 GiB of RAM, and 10GB of allocated storage within a single VPC and using general-purpose SSD storage rather than disks.

We assigned data distribution across nodes as described in the previous section. In creating our tables, the auto-incrementing primary key was the hashID, indexed as such in each structure, with each insert wrapped in a transaction that called the last insert ID as the hashID for elements of the same record across additional nodes. We also indexed the create_time/update_time fields as a single timestamps index across nodes.

We then identified eight common real-life occurrences in the maintenance of PII databases: one full-record insert scenario, three partial-record update scenarios, and four specific search queries based on various criteria. We then planned out and created stored routines with which to insert 310,000 records into each of the one-node, two-node, and three-node structures; perform various updates; and run queries. To test the performance of each workload, we pasted each of these stored routines into our previously created benchmarking tool, which itself consisted of two stored procedures and with which we were able to measure the increased cost in terms of CPU utilization and time when
running the same workloads across one, two, and three nodes. We stored the two procedures in the "benchmark" database/node, along with other stored routines.

The main procedure in this benchmarking tool processed the insert, update, or select routine that was pasted into it and then simply called a second, previously defined procedure, termed testProcedure, that selected 1+2 into the result and repeated the process defined within the main procedure, seen below, x number of times:

```
CREATE DEFINER=`root`@`%` PROCEDURE `mybenchmark`(setNum INT, benchmark varchar(255))
BEGIN
  DECLARE stamp1, stamp2, stampdiff, num INT;
  SET stamp1 = UNIX_TIMESTAMP();
  SET num = setNum;
  WHILE num > 0 DO
    START TRANSACTION;
    [stored insert, update, or select routine pasted here]
    COMMIT;
    SET num = num - 1;
  END WHILE;
  SET stamp2 = UNIX_TIMESTAMP();
  SET stampdiff = stamp2 - stamp1;
  SELECT stampdiff;
END

call benchmark.mybenchmark(10000,'CALL benchmark.testProcedure()');
```

Having created our structures of one-node, two-node, and three-node designs, we used the benchmarking tool described above to insert 190,000 identical records into each structure as a base on which to build.

5.2 Exact Workloads

5.2.1 Workload 1: Insert Full Record

For our first measured experiment, we inserted the same workload, 12 sets of 10,000 records, into each of the three designs, bringing the records count of each
structure to 310,000. Using Amazon's CloudWatch monitor and the CPU utilization monitoring tools in RDS, in addition to our benchmarking tool, we calculated the CPU utilization and the time required to insert the 12 sets of 10,000 records into each of the three designs. In the first figure below, the CPU utilization of the two-node design was 93.8% of that of the three-node design, while the one-node design consumed 3.3% less than that (at 90.5% compared with the three-node design). The results of previous inserts were not 100% consistent with those of this measured experiment; in some cases, we observed the CPU utilization of the two-node design somewhat closer to that of the three-node design, matching the pattern of the average time cost seen below. Overall, for all observed inserts, there was less than 10% difference in overhead between the one-node and three-node designs, with a slight increase with each node addition. It is important to note that although the differences in performance may appear significant from this zoomed-in perspective (to clearly observe where the greatest changes occur), in reality, the number values of each are fairly similar.

![Figure 4. Full record insert](image)

5.2.2 Workload 2: Update Criminal History

A common real-life workload for PII database systems might entail updating an individual's criminal history. In the case of no criminal history, the charge, plea, and sentence fields were null, but all of those could change if the criminal status changed. In
the criminal history update that we performed across our three designs, the status (1 bit), charge (varchar 45), plea (2 bit), and sentence (varchar 45) were modified, with most changes occurring in node 2. Here again, we have zoomed in to note where the greatest changes occur.

Figure 5. Criminal history updated

5.2.3 Workload 3: Update Last Name

Another common update is a last name change, as with a marriage. This change occurred in node 1 in the three-node design and in all nodes in the one-node and two-node designs. Note that in this case, we have not zoomed in on time cost or CPU utilization but rather kept the base at 0. This more accurately depicts the overall cost differences between designs, whereas the previous charts more clearly highlight where the greatest changes tended to occur.

Figure 6. Last name updated
5.2.4 Workload 4: Select Range Scan of DOB for Individuals > 17 Years of Age

While most PII queries would typically select only a single record, a real-life scenario that could require a range scan would be a search for individuals qualifying for a given privilege based on age. In the query seen below, we searched for all individuals born after December 31, 1998, resulting in 110,000 records being selected and returned. The average time and CPU costs for this workload are shown below. In this case, even with the base kept at 0, the differences between the results for each design were significant for both time and CPU utilization.

![Average time to select range across nodes](image1)

![Average CPU utilization across nodes](image2)

*Figure 7. Select range scan*

To further elaborate on the above results, having used our benchmarking tool and Amazon's CloudWatch and RDS monitoring tools to measure the cost when running the same query over each design, we observed the following statistics:

- **Three-node design** – duration: 0.984 seconds / fetch average: 31.95 seconds / CPU utilization: 29.98 units over 11 minutes
- **Two-node design** – duration: 0.797 seconds / fetch average: 27.2 seconds / CPU utilization: 24.01 units over 9 minutes
- **One-node design** – duration: 0.593 seconds / fetch average: 16.42 seconds / CPU Utilization: 15.37 units over 7 minutes
The CPU utilization of the one-node and two-node designs was 51.27% and 80%, respectively, of that of the three-node design. The time cost of the one-node and two-node designs was roughly 60% and 81%, respectively, of that of the three-node design. It is important to note, however, that this type of search would not likely be among the most common searches. Routine business queries would usually focus on selecting specific records rather than ranges.

We were particularly interested in the overhead results for searches (versus inserts and updates) because this would be the type of workload performed most often by the public, so we examined not only total CPU utilization over time but also the cost of the query itself, based on Oracle's cost model (Sandstå, 2014). The following outlines the cost model for the above query, a range scan.

Above, the simple explanation of query cost is that with the particular database system we used, for a single node, the read cost = 1.0 * (# records in range), and CPU cost = N(0.2 * (# records in range)), where $N$ refers to the number of evaluations required for the query. For each additional node, there is an additional cost of $x + y$, where $x = (\text{filter from the previous node(s)}) * (# \text{ total records})$ and $y = x * 0.2 * (\text{filter on current node})$.

![Figure 8. Cost model](image)
5.2.5 Workload 5: Select Criminals in [City]; Group by Last Name

Thus far, the pattern has been similar in terms of both time and CPU cost percentages for the one-node, two-node, and three-node options across all workloads: inserts, updates, and selects. In a search for all individuals with a criminal record living in the city of Miami, grouped by last name, however, the pattern changed. Overall, the overhead still increased with each node addition because of increased connections and the greater cost involved with running a larger system with more machines. However, the duration/fetch time and CPU utilization of the select process itself actually decreased with each node addition, as seen below.

![Figure 9](image)

*Figure 9. Select criminals and group by city*

Similar to the range scan described in 5.2.4 above, we returned all fields for this search. Notably, only the hashID and timestamps fields were duplicated across databases/nodes in the three-node option, whereas the two-node option duplicated the first name, last name, and address fields (in addition to hashID and timestamps) in the two databases.

It is also worth noting that this particular workload added a cost to the overall CPU utilization that remained constant regardless of node additions. This constant cost,
depicted in the figure below, is the temp table/filesort created through the "group by" clause.

Figure 10. Constant group-by cost

The constant overhead of the group-by clause was equivalent to (total % filters on databases/nodes) * (# records). We explored the filter percentages in this particular query and in a few other queries using the explain statement in Command Prompt, but due to limited time, we were unable to conduct further tests to evaluate to what extent this and other types of constant costs may affect CPU utilization changes in relation to added nodes.

5.2.6 Other Workloads

We identified three additional common, real-life workloads in the maintenance of PII databases, as noted at the beginning of Section 5.1. One was an update, adding a state ID to an existing record, as might occur when a child becomes an adult. For this workload, the overhead pattern was similar to those of the early experiments.

For our two additional search workloads, we performed simple queries, searching for specific individuals based on state ID for the first query and on SSN for the second query. In both cases we selected only four fields total instead of retrieving all fields as in
the previous workloads. In addition, we were able to eliminate the need for a subset of PII in each case. To elaborate, our original three-node design had been structured as follows:

- **Node 1**: first name, last name, and address
- **Node 2**: ID (SSN or alien registration number), ID status, and criminal status (and related info in the case of a criminal record)
- **Node 3**: state ID, state ID status, and DOB

As noted in Section 4.2, our two-node design simply eliminated node 1 from the three-node design and added its contents to both node 2 and node 3. Thus, each of these two nodes could function as the one-node option when the additional data in either node 2 or node 3 was not required for a particular query, as was the case now. Furthermore, our two-node option would now query only node 1 and node 3 from our original three-node design when searching based on state ID and only node 1 and node 2 when searching based on SSN.

Having eliminated a subset of PII for each of these two final queries, as described above, the two-node option took less time and demonstrated less CPU utilization than the one-node option; only two fields were retrieved from each of the two nodes queried, and both nodes were queried simultaneously. The results were similar to those of Section 5.2.5.

Notably, regardless of variations in time cost or CPU utilization, the overall overhead was never actually reduced for any workload through node additions because the connection and evaluation costs still increased. There was also the practical cost involved in running additional machines. All of these issues were factors in our analysis of the increase in overhead with every node addition.
Chapter 6  Discussion and Analysis

6.1 Overview of Experiments and Results

In our experiments, all nodes were located on the same network, minimizing the latency and communication costs that could otherwise occur. It is important to also note that these experiments do not address the additional security issues and overhead that would be encountered in expanding the structure. For example, replication across sites would necessitate multiplication of every node rather than of a single database, incurring additional costs. Such expansion would also require additional security considerations for data in transit; these experiments focus primarily on data at rest. However, as we discuss later in this section, the same security considerations should be in place regardless of the design, and the benefits of data isolation in separating subsets of PII across nodes greatly outweigh the drawbacks.

In our experiments and results, we have presented worst-case scenarios; performance might usually be much better than we demonstrate here, given that most queries on PII would involve only specific fields in single records and not range scans of all fields of the records in range. As such, a search using a stored procedure that queries multiple nodes simultaneously, selecting one or two fields from each database/node, might actually have better performance than the same search of all fields performed over a single database/node. The location of the nodes in relation to each other would be a factor; servers stored together on the same subnet, for example, might demonstrate
excellent performance, whereas servers communicating from different regions might not. At this time, we have not conducted sufficient testing to form solid conclusions about these issues, but the few experiments we did conduct showed promising performance for this type of query given the abovementioned conditions.

The overall concept behind our design included wrapping every insert statement or query in a transaction. For inserts, a new hashID—the primary key common to all tables—was created only for the first node, and LAST_INSERT_ID() was called for each successive part of the insert to ensure that the same hashID was used for the same record across nodes. Updates and searches could then easily select a record from multiple nodes with the hashID as the common identifying factor.

Another important feature of our design entailed the use of stored procedures. The various transactions that we created were generally stored in routines/procedures, and we isolated all stored routines/procedures in a node separate from all other databases. This is in keeping with data isolation as a security measure. The use of stored procedures could also be employed to limit clients' access to only the needed PII fields based on specific permissions/policies.

For these experiments, we isolated only three subsets of PII, based on the types of queries we have observed and on the principle of least privilege, and stored them across nodes. In some cases, however, the ideal number of data subsets may vary; and later in the discussion, we present a method of balancing performance/overhead with security in terms of selecting an appropriate number of nodes over which to distribute subgroups of data.
6.2 Security Benefits of Multiple-Node Structure

These experiments were geared solely toward evaluating the increase in overhead due to node additions and not to evaluating the increase in security that such a structure provides; however, there is ample support for the underlying security benefits we assert. First, the concept of data/server isolation as a security measure is already well established (Ellingwood, 2015; Northcutt, 2009). Second, this paper does not introduce new security concepts but rather extends existing principles that are accepted in other areas to a new field. For example, various researchers have affirmed that not only columns but also discrete elements can and should be individually encrypted as a security measure before and during storage and throughout the query process (Idreos, 2016; Popa et al., 2011; Song et al., 2000). An extension of this principle entails also securing each database, which can easily be done through encryption and password protection, as seen in Microsoft's "Encrypting a Database" (2016) and "Password Protecting a Database" (2016).

For several reasons, we view the need to secure individual servers as even more important than database encryption. One reason is simply that extensive technologies exist for protecting servers—one example being the hardware security module (HSM). Another reason is that a huge percentage of malicious attacks are mounted against servers (Goodin, 2016; Keizer, 2009; Maximo, 2010). Thus, a logical security measure is to ensure that (a) no single server hosts all sensitive data and (b) each server hosting subsets of data is individually secured. By extension of the same logic, the technology used to secure each server should be unique so that if a hacker is somehow able to compromise one server, he or she does not gain access to the others.
A third reason to distribute subsets of data across servers is that doing so prepares for advances in technology. As concern mounts regarding attacks against servers, new technologies will continue to be developed to protect them. Only if server isolation is already in place can database administrators take advantage of these advances in technology by applying new, unique security measures to each server, further preventing intruders from accessing entire sets of data.

This is all in addition to the fact that the design of data distributed in isolated subsets across nodes itself opens the way for establishing barriers, with potential for permission policies granted to users at specific levels according to need.

6.3 Distributed Data Options

While a particular node structure may function well for certain types of data, the same structure may not be ideal in other situations. In this paper, we have demonstrated the overhead that must be considered in dividing PII into subsets and distributing those subsets over multiple nodes. Certain costs are constant, whereas others are not. For example, depending on the system structure and types of searches, the time required to process queries may actually decrease with node additions. However, the CPU and I/O costs as well as the number of connections and machines required consistently increase with each node added.

In terms of security gains, with each node addition (assuming this entails further subdividing and redistributing the data), the principle of data isolation is established to a greater extent. Not only is a smaller amount of data exposed in the event of a breach, but the very structure enables the use of new security measures for each individual node as they are developed.
In our examples, as noted previously, we associated every record with a hashID, using a transaction to process each record so that the subdivisions of the record were always associated with the same hashID across nodes. The structure best demonstrating the principle of data isolation—and, in our opinion, the best and most secure option—exemplified in our experiments is the three-node design. This design maintains a separate database for nonsensitive information such as name and address and then divides sensitive PII into two parts: (a) SSN/criminal history and associated information and (b) DOB/state ID and associated information. Although nonsensitive information itself might easily be obtained from property records or an address book, it could be damaging when associated with sensitive PII. Based on the same rationale, identifying numbers such as the SSN and state ID mean less when not associated with a name/address.

As discussed in previous sections, the basis for how we subdivided PII relates to the types of information required in common business queries. Our design is only one example, however. In reality, every column could be stored in its own database/node, with each element from the same record associated with the same hashID across nodes. A greater extent of data isolation—the isolation of every element, in such a case—would obviously allow for even less data exposure in the event of a server breach. The decision as to how much data should be stored on a single node is therefore a matter of balancing increased overhead with increased security.

It is worth noting that the same security measures could be applied with other types of data as well and that the ideal design, balancing overhead and security, might vary according to the needs of a particular client. To that end, we have prepared a basic
input form that might enable any client to make these decisions, along with a chart showing sample output.

The basic information required to make such a decision might include the following:

1. **How many unique fields exist in my database(s)?**

   (This is \( f \). The maximum number of nodes that could be used, based on our design, would be \( f + 1 \).)

2. **Are certain fields logically queried together consistently, with some departments/clients querying only subsets of the data?**

   (Storing the fields together that are queried together cuts back on CPU, connection, and I/O costs while isolating these fields from data not needed for those particular queries.)

3a. **If the answer to #2 is yes, then to obtain a "good" (medium) level of security, determine the following:**

   i. Which fields are common to all (or most) queries?
   
   ii. How many groups of fields are searched only for a limited type of query?

   **Note that the number of subsets used to obtain medium security could range widely, depending on database size, with a minimum size of 3–6 fields per node. See 3b for a rough idea of what this might look like. Here, the objective is to group together the least number of fields typically queried together while isolating as a separate node(s) those fields that tend to be common to all queries.**

3b. **If the answer to #2 is no, then form divisions as follows for medium security:**

   The basic equation for any level of security is \( N = (f/n) + 1 \), where \( N \) = # nodes, \( f = \) # unique fields, and \( n = \) # subsets. In this equation, at the lowest level of security, \( n = f \);
while at the highest level, \( n = 1 \). For medium security, we recommend that if \( f \leq 8 \), then \( n = 2 \). For \( 8 < f \leq 60 \), increment \( n \) by 1 approximately every additional five fields until \( n = 12 \) (adjust as needed; for a database of 60 fields, \( n = 12 \)). To achieve medium security for large databases, the number of fields in each node could be revised, with each subset containing roughly \( 1/12 \) of the total unique fields.

The graph below depicts a database with exactly 12 unique fields.

![Figure 11. Security versus overhead](image)

That is all of the information we need as input. We use the client's responses to questions 1, 2, and 3 to determine the value of the variables in our graph, as previously detailed. Our explanation to the client is then the following:

Certain costs are involved in setting up and running any system, including normal expenses such as database administrator, network, and so on. In addition, our design, without any isolation of data subsets across nodes, requires two nodes for setup: one in which to keep all stored procedures and functions and one as the initial one-database starting point. We rank this initial structure as low security because it entails the minimum requirements for our system; if the server storing the single database is breached, all data is potentially exposed.
Medium security is roughly estimated using the equation as explained in 3b above, with 3a as the ideal scenario. An example of this type of security is our three-node design: fields typically necessary for any PII query include the nonsensitive name and address associated with the record (common fields = node 1). This is true regardless of whether the query verifies only the SSN and criminal history (node 2), as in a W2 form submission, or only the state ID and DOB (node 3), as in an age-verification check. In a full-record check, all or most fields of all three nodes might be queried, but logical subsets can be created for when a full-record scan is not required.

High security entails storing every field in a separate node, associating each element with a hash ID that is unique to all fields of that record. This level of security not only allows for encryption of each element, each database, and each server for every single field but also, by its very structure, ensures that every new security measure invented, present or future, can be applied to each server individually. This is an important point because servers are a major point of attack (Goodin, 2016; Keizer, 2009; Maximo, 2010), and new server protections are continually being developed.

Several points might be identified between these three levels of low, medium, and high security. For example, a midpoint between medium and high security could entail subdivisions beyond the subsets required for medium security while not isolating every field on a separate node. In considering PII as an illustration, medium-high security might involve separating the SSN from criminal history (both currently stored on node 2) and doing the same with the DOB and state ID, using a total of four nodes to store the SSN, criminal history, DOB, and state ID (and associated information) instead of two.
Finally, in the equations above, it is important to note that overhead does not multiply by a constant factor with each node addition. There is some increase in I/O and connection costs, and there may be some increase in time cost and CPU utilization (depending on the types of queries and node structure), and so on. However, although the total cost does equal the system expenses plus the cost of total nodes (and their associated overhead), each node adds a much smaller percentage of overhead, by comparison, than that involved in running any system.
Chapter 7  Summary, Limitations, and Future Work

7.1 Summary

In this paper, we have presented the beginnings of a design for increased PII security through the well-established principle of data/server isolation. We divided the personal data fields used for our records into subsets, distributing them across one-, two-, and three-node designs, using one additional node to store all routines and procedures. In our experiments, we demonstrated the cost increases that might be expected with several types of statements and queries, showing some increase in overhead, overall, with each node addition. We also noted, however, the extensive security benefits that server isolation provides, including the ability to implement every new security technology invented, present or future, to protect each server individually.

In addition, we provided an overview of security levels, from low to high, demonstrating the steps one might take in determining what level of security to use and logical ways to compartmentalize and distribute subsets of data fields across nodes. This overview is merely a draft that might be greatly improved upon and further developed through more extensive analysis of the specific costs versus benefits involved in node additions.

Although our contribution focused on security through strategic distribution of data subsets across individually secured nodes, this measure alone would not be sufficient to secure PII. Our goal was to demonstrate one aspect of a viable solution to the
overwhelming problem of PII breaches within a multifaceted security system. It is important that the reader also understand and implement other important Big Data principles, such as sound database selection, good encoding, and strategic data encryption. Thus, we also provided a brief overview of these foundational principles and directed the reader to others' work, where the underlying concepts can be reviewed in greater depth.

7.2 Limitations and Future Work

Our experiments did not address the additional security issues and overhead that would be encountered in an expanded implementation, such as replication across sites. For this limited study, all nodes were located together both logically and physically, minimizing the latency and communication costs that could otherwise occur. Future research might experiment with various structures both within a subnet and across different networks.

Another limitation is that in our approach to safeguarding PII data, we do not address the issue of insider threats, which—intentional or otherwise—account for a significant percentage of data breaches (Hatchimonji, 2013; PTAC, 2011). Although strategic planning with regard to data distribution across nodes, along with the server/client authority structure defined earlier in this paper, greatly aids in protecting against massive data breaches, there is still the concern that an insider with the appropriate permissions could misuse legitimately accessed data.

An interesting direction for future research might be the development of policies facilitating joins/aggregations at different permission levels for query processing. In our paper, we primarily focused on a limited set of data typically accessed in routine queries,
such as for an employment background check, but there may be legitimate reasons for regularly accessing more extensive sets of PII data as well. In such cases, the client should acquire the appropriate permissions before being granted a higher level of access, perhaps through a capabilities system such as that suggested in the Naval Postgraduate School's "A Survey of Distributed Capability File Systems and Their Application to Cloud Environments" (2014).

Along this vein, future studies might include statistics on the exact data and workloads most often required for routine business queries (and other frequent queries on PII). Such research might also involve varying the subsets of data stored on individual nodes while using an expanded implementation of replicated data across various sites, as mentioned above.

While these are only a few of the directions future research might take, our primary hope is that this paper can serve as inspiration for actual execution of these strategies. PII security is both important enough to merit action and achievable here and now. Notably, not one of the concepts mentioned in this paper is new; we merely apply well-known best practices from other areas to a new field. Thus, we maintain that the task of securing PII is not too difficult, too complex, or too advanced in technology. The simple problem is that it remains undone.
References


Chen, Z., Gehrke, J., & Korn, F. (2001). Query optimization in compressed database systems (pp. 271–282). In ACM SIGMOD.


Untrusted Data Repository (SUNDR). Proceedings of the 6th Symposium on
Operating Systems Design and Implementation. Retrieved from
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.4.8515

Madden, S. (2009, June 30). MapReduce vs. parallel DBMSs: A comparison of
approaches to large-scale data analysis. Brown University, Department of
Computer Science (video).

force attacks in Office Communications Server. Microsoft TechNet. Retrieved

McCallister, E., Grance, T., & Scarfone, K. (2010). Guide to protecting the
confidentiality of personally identifiable information (PII). National Institute of
Standards and Technology (NIST). Retrieved from
csrc.nist.gov/publications/nistpubs/800-122/sp800-122.pdf

Microsoft. (2005, April 5). Server and domain isolation using IPsec and group policy
(overview). WindowsSecurity.com Retrieved from
http://www.windowsecurity.com/whitepapers/Network_Security/Server-Domain-
Isolation-IPsec-GP-Overview.html

cc725770(v=ws.10).aspx


*Northern Rockies Coordinating Group.* Retrieved from  
http://www.fs.fed.us/r1/fire/nrcg/  


