Privacy-preserving OLTP database systems with OLAP support

Sidorov, Vasily

2016

Sidorov, V. (2016). Privacy-preserving OLTP database systems with OLAP support. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/66931>

<https://doi.org/10.32657/10356/66931>

NANYANG
TECHNOLOGICAL
UNIVERSITY

**PRIVACY-PRESERVING
OLTP DATABASE SYSTEMS
WITH OLAP SUPPORT**

VASILY SIDOROV

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Doctor of Philosophy

2016

Acknowledgements

To my family.

Paper can hardly convey my deepest gratitude to all those who made this work possible. First and foremost, I want to thank my supervisor and my Teacher Dr. Ng Wee Keong. He reached out to me when I was a young and inexperienced master's student in Moscow State University, and believed that I have the potential. His wise guidance brought me to finishing this work, his constructive criticism was constantly motivating me to improve, his careful approach to mentoring me allowed me to grow as a researching scientist. He has shown a limitless patience when working with me and has taught me countless invaluable life lessons.

I want to thank Dr. Zhu Huafei, who set an example of a hard-working scientist for me, who was always full of ideas and was always ready to give a hand, and whose precious advice were always on the spot and insightful. I want to thank my mom and dad, whose support I always felt, even through thousands of kilometers between us.

Doing this research and pursuing a doctoral degree was truly a life-changing experience for me. Mere four calendar years appeared to be what seemed like a lifelong journey full of doubts and determination, failures and accomplishments, defeats and victories. I have embarked on this journey as an immature student, but thousands of hours spent in the silence of libraries, in early morning lectures and seminars, or in the lab during the late hours, all this resulted in the work that is now before you, and in me becoming, hopefully, a developed researcher.

I would also like to thank Nanyang Technological University, the School of Computer Engineering, and A*STAR for giving me the opportunity to do this; the Republic of Singapore where I always felt welcome and where I found so many great friends; Ms Len Ah Chan, Ms Chiam Poh Ling, and Mr Thomas Loo Kian Hock for their constant support; Singapore Ministry of Education for the research grant that partially supported this work.

Contents

Acknowledgements	i
Contents	iii
List of Figures	vii
List of Tables	ix
List of Definitions	xi
Abstract	xiii
1 Introduction	1
1.1 Motivation	1
1.2 Research Objective and Approach	3
1.3 Research Issues	5
1.4 Thesis Organization	7
2 Related Work	9
2.1 Definitions	9
2.2 Encrypted Search and Querying	10
2.2.1 Types of Encrypted Search and Queries	10
2.3 Authentication, Authorization, Access, and Key Management	26
2.4 Auditability	31
2.5 Existing Systems	35
2.5.1 CryptDB: Structure, Capabilities, Workflow	36
2.5.2 Limitations of CryptDB	38
2.6 Summary	40
3 System Overview	43
3.1 Approach Overview	43

3.2	Model Design	44
3.2.1	Notion of “Data Security”	44
3.2.2	Threat Model	45
3.2.3	Model Topology	46
3.2.4	Terminology	48
3.3	Summary	49
4	Query Execution Model	51
4.1	Querying Capabilities	52
4.1.1	DDL	52
4.1.2	DML	53
4.1.3	TCL	54
4.2	Abstract Query Execution Model	54
4.2.1	DDL	55
4.2.2	DML	57
4.3	Summary	68
5	Transaction Management	71
5.1	Introduction to OLTP	71
5.2	Atomicity under the QEM	73
5.2.1	Proxy Standpoint	74
5.2.2	User Standpoint	77
5.3	Isolation under the QEM	79
5.4	Durability under the QEM	81
5.5	Transactions under the QEM	82
5.6	Summary	82
6	Model Implementation	85
6.1	Storage Model	85
6.1.1	Viable Approaches	85
6.1.2	Proposed Approach	86
6.1.3	Analysis and Improvements	88
6.1.4	Cryptoset Security Analysis	88
6.2	Operations Modules	92
6.2.1	Arithmetic Operations	92
6.2.2	Equality Matching and Search	94
6.3	Summary	101

7	Empirical Results	103
7.1	Arithmetic Operations	104
7.1.1	Multiplication	104
7.1.2	Addition	106
7.2	Equality Matching and Search	108
7.3	Full System	111
7.3.1	Correctness of Implemented SQL Queries	112
7.3.2	Correctness of OLTP	114
7.4	Summary	116
8	Conclusions	119
8.1	Theoretical Model	121
8.2	Implementation and Empirical Results	122
8.3	Other Results	123
8.4	Limitations	124
8.5	Future Research	124
	References	127

List of Figures

2.1	Taxonomy of notable works on encrypted search	12
2.2	RSA-based proxy re-encryption scheme	15
2.3	Forward index	22
2.4	Inverted index	23
2.5	Defense layers	27
2.6	Cloud key management infrastructure	29
2.7	The architecture of cloud data storage service	32
2.8	CryptDB onion encryption scheme	35
2.9	CryptDB architecture	36
3.1	Main system components	47
4.1	How the QEM affects the interaction with the DBMS	55
4.2	Example of a <code>CREATE TABLE</code> DDL query transformation	56
4.3	Predicate in a Conjunctive Normal Form	61
4.4	Example of an <code>INSERT INTO</code> DML query transformation	64
4.5	Example of a <code>SELECT</code> DML query transformation	67
6.1	Illustration of the multiple tables approach to storage model	86
6.2	Illustration of the single table approach to storage model	86
6.3	System architecture for equality matching scheme	94
6.4	Encryption workflow	97
7.1	ElGamal homomorphic multiplication performance	105
7.2	Paillier homomorphic addition performance	107
7.3	Performance testing results	111
7.4	Queries schedule for testing the isolation of transactions	116

List of Tables

1.1	Categories of threats	5
2.1	Categorization of works by usage scenario vs. encryption scheme type	18
2.2	Example plaintext data	20
2.3	Example encrypted data	20
2.4	Categorization of works by type of data vs. type of search	23
5.1	Failure points and locations for atomicity analysis	78
7.1	System configuration sets used for search benchmarking	110

List of Definitions

2.1	Definition (Domain)	9
2.2	Definition (Attribute)	9
2.3	Definition (Relation schema)	9
2.4	Definition (Tuple/Record)	9
2.5	Definition (Relation)	10
2.6	Definition (Database)	10
2.7	Definition (Trapdoor function)	16
2.8	Definition (Trapdoor for encrypted search)	16
2.9	Definition (Authentication)	26
2.10	Definition (Authorization)	26
3.1	Definition (Encryption)	48
3.2	Definition (Hash)	48
3.3	Definition (Cryptographic transformation)	48
3.4	Definition (Cryptoset)	48
3.5	Definition (Decryptoset)	49
3.6	Definition (Virtual data schema)	49
4.1	Definition (Single-pass query)	57
4.2	Definition (Multiple-pass query)	57
5.1	Definition (Serializability)	71
6.1	Definition (Negligible function)	88
6.2	Definition (Indistinguishability/IND-CPA)	89
6.3	Definition (Semantic security)	89

Abstract

Migration of data storage and processing appliances to the cloud is a stable trend in recent years. As many confirm, enterprises could gain various managerial and financial advantages from such change. However, at the same time, new security risks arise. In particular, certain risks of confidential data leaks. Using cloud platforms typically means losing control over the hardware, which might be considered advantageous from many points of view, but aggravates the security risks and blocks some approaches to their mitigation. Additionally, the cloud platform provider itself could be considered a security threat. Encryption of the data could alleviate the problem for the storage, but being done in a directly, it makes processing of the data in the cloud impossible.

Specifically, cloud-hosted database systems are very affected. Typically, database systems store large amounts of sensitive, confidential information; and typically, they are expected to be able to carry out complex data processing tasks—either transactional or analytical. Thus, finding a way for a cloud-hosted database system to operate the data it stores in a privacy-preserving manner is a demanded research direction. This work is dedicated to a careful and systematic investigation of this issue.

Even though the history of relational database systems is now more than 40 years long, and many approaches had time to get standardized, there still is a notable diversity in practical database systems, many of which have a narrow purpose orientation, and many make attempts to be more or less universal. Moreover, the notion of “data security” is very diverse and volatile by itself; it depends on many factors, including the level of importance of specific data, and the set of threats it needs to be protected from. It is thus reasonable to explore abstract database models and abstract security models, and investigate how they interact, how they behave when combined, and how the database has to adapt its protocols in order to function under the security model.

This work theoretically considers an abstract relational database system that is able to execute data processing primitives over encrypted data, and combine these primitives into more or less arbitrary sequences, which gives way to supporting significant subsets of SQL over encrypted data. The work also considers practical implementations of sample encrypted processing primitives and demonstrates a proof-of-concept encrypted database system, which shows that the theoretical model that is developed and discussed in this work is feasible in practice.

Chapter 1

Introduction

Secrets are things we give to others to keep for us.

— Elbert Hubbard

1.1 Motivation

In recent years, more and more businesses' operating activities have started to involve an exponentially growing amounts of information and its security became a precious asset [2, 49]. However, managing the fast-growing amounts of data is a non-trivial and costly task. In the same time frame, the Software-as-a-Service (SaaS) paradigm was becoming more and more popular [22]. Hence, certain vendors started to provide such services as *Database-as-a-Service* (DaaS). Others provide just an infrastructure (IaaS) or a platform (PaaS) for the customers to setup their own data managing solutions on top of.

PaaS solutions at their current level of development do not find a strong interest from enterprise customers whereas SaaS and IaaS solutions combined maintain up to 90% of the market [21, 30]. The analysis also shows that, while NoSQL¹ data solutions are having a steady growth, the market is strongly dominated by traditional relational database systems, and with current trend lines the status quo is projected to stay for years ahead [6]. Only recently have major cloud providers started providing managed relational database solutions² in a form of DaaS to their customers. However, current DaaS solutions are typically not much more than managed installations of

¹NoSQL is a collective term to denote non-relational database systems, such as document-oriented systems, key-value storages, and others.

²E.g., Amazon RDS, Microsoft Azure SQL Database, Google Cloud SQL, Rackspace Cloud Databases.

conventional DBMSs³ on top of IaaS in the same way the IaaS customer could do it himself, and in the same way it is done as an on-premise solution.

At the same time, the outsourced data storage and processing has its advantages—it allows organizations to reduce drastically IT maintenance costs (which is especially important to new small businesses) and to improve flexibility and scalability of business [1]. On the other hand a whole set of new problems arise: access control, data privacy, trustworthiness of IaaS provider, performance, auditability, and many more.

One of the major concerns are the constantly increasing risks of data theft—security vulnerabilities in system software, applications, or network protocols implementations, which are all not rare, could (and often do) lead to the sensitive data being leaked. In addition, in the data moving to the outsourced cloud database scenario⁴ these risks are aggravated by the fact that not only third party ill-wishers, which could try to intercept data transfer from the outsourced storage to client and back should be considered. In general, it should also be assumed that the cloud service provider itself is untrusted, as it has full and uncontrolled access to the hardware, software, and the data.

A direct encryption could guarantee data privacy along with causing a performance bottleneck: without applying additional efforts, database becomes no longer able to answer data queries without sending back the whole encrypted piece of data for further client-side decryption and query execution. This setup, obviously, renders database systems useless, and though might be acceptable in certain scenarios, could not be considered a universal solution.

High demands of corporate and individual customers in outsourced data storage motivate researchers to work in these directions and implement their visions of how it should be done. Exempli gratia, R. Popa *et al.* in their project “CryptDB” relied on a homomorphic encryption, which made the database server able to perform certain computations on encrypted data without decryption [59]. Other researchers could evaluate homomorphic encryption as too weak and encrypt sensitive data with stronger non-homomorphic probabilistic encryption while performing all computations on a client side. This second approach is evidently much less efficient, especially if the client-side application is hosted on a thin client or a mobile device, but the sensitive data is safer.

³Database Management System.

⁴From now on in this work the terms *outsourced database*, *outsourced data storage*, *database-as-a-service* and *cloud database* will be used as synonyms.

As the current trends in this area and current state of the art appears to be, there is a significant difference between building a regular and a secure database. It has already been more than 40 years since E.F. Codd first introduced a relational model of data in 1970 [20], and many generations of relational database systems adapting iteratively to current demands by commercial sector had elaborated more or less universal approaches in database system design, which meet almost all customers' requirements. Most of the modern popular relational database systems follow these approaches and mostly differ only in their targeting—OLTP⁵ (MySQL, MSSQL, Oracle, ...) or OLAP⁶ (Vertica, Teradata, MonetDB, ...).

When researchers discuss introducing security into a database system, the new angles of view arise. One of them is the security–performance trade-off, mentioned earlier. The security itself is a complex notion, which could vary in its level and target (e.g., secure against certain attack types, secure against certain adversary behavior models, etc.) and could be achieved in a number of approaches. Therefore, in the case of a *secure* database it is much harder to design a universal solution. On the other hand, it is still possible to make individual solutions for specific requirements for performance and security levels. However, it is unclear how exactly can one build a secure database system given the requirements, which brings us to the next section.

1.2 Research Objective and Approach

Summarizing what was said up to this moment, while privacy-preserving data processing is in demand, every specific application requires its own set of data processing primitives, has its own security requirements, and has its own limits of what can be sacrificed for security: e.g., time complexity, space complexity, etc. With that in mind, let us state a research objective for this work:

To create an abstract, universal way to translate a given set of relational, OLTP, and OLAP tasks into a semantically identical set of privacy-preserving tasks with security and performance properties variability.

Analyzing current state of the art in secure cloud databases and data storage systems, a researcher could notice that the set of tools, which are usually used by developers is finite. For every aspect in designing a database system there are only a few approaches—each with its benefits and limitations. Combination of these approaches finally form a developed database system and defines its performance,

⁵Online Transaction Processing

⁶Online Analytical Processing

security level and other features. A solution to a certain aspect made using a specific approach and tailored in a way that it can be used as a part of a whole system without interfering with other parts will be called a “building block”. Now, the approach to achieving the research objective could be formulated in a following way:

To develop a basic set of *building blocks*, to design *approaches* to create new ones, and to produce a *methodology* allowing enterprise system architects, administrators, or hired third parties to construct practical and secure relational cloud OLTP/OLAP database systems using the building blocks as close as possible to their very own needs (regarding the security–performance trade-off) and with known security properties.

There is a wide range of possible usage scenarios where *secure* relational cloud databases are essential.

As an example, let us consider an imaginary retail company Acme Corp. It first needed to keep records on customers’ orders from their visits to the stores in a database—marketing and planning department wanted to calculate total amount of visitors for any subset of stores and total income (i.e., sum of all bills) of any subset of stores—both during various time periods. Company wants to outsource the database to the cloud to reduce IT maintenance spending but information about customers should be kept undisclosed.

Based on this demands a formal requirements were produced: company needs a multiuser cloud database, which stores data in an encrypted form (to keep data private), supports temporal (`DATETIME`), numeric, (`INTEGER`⁷) and textual (`CHAR`) data, and is capable of:

- performing equality/inequality checks for all data types—so that it is possible to filter for specific store, customer or date;
- greater/less checks for the numeric and temporal data—so that it is possible to specify time periods;
- `COUNT()` aggregation function—to get the amount of customers;
- `SUM()` aggregation function—to calculate the income;
- process concurrent queries from different users.

After that the *methodology* should be used to determine which system architecture is the most suitable for the situation, which encryption schemes should be used, how required operations should be implemented, what access control techniques should be

⁷Assuming that prices in the stores are integral.

	Internal threat	External threat
Passive adversary	Curious DBA, ...	Industrial espionage, ...
Active adversary	Malicious DBA, ...	Industrial sabotage ⁸ , ...

Table 1.1: Categories of threats

used, how to assess the overall system security, etc.

After the secure DBMS was created for the Acme Corp., the marketing and planning dept. realized that they also need to see what the overall average bill, average bills of individual customers, and average bill at each store are and how are they changing through the time. Moreover, greater/less checks for numeric data appeared to be completely unused and the decision was to cut this functionality out in order to reduce the system complexity.

So the support of an **AVG()** aggregation function was included into the formal requirements and greater/less checks for numeric data were removed. Since the developers of the secure DBMS for Acme Corp. were using the *methodology*, which is intended to provide a good flexibility and modularity, the changes in the requirements did not entail significant changes in the existing system and the required changes were done relatively fast.

1.3 Research Issues

In order to achieve the stated objective using the proposed approach a set of research issues should be explored and, if possible, solved.

Security Threats

Discussing security without identifying the exact types and classes of threats against which the system is supposed to be secure is too abstract and thus pointless from both practical and research perspectives.

Typical threats considered in relation to secure DBMSs could be roughly divided into 4 categories as shown in Table 1.1. Here the database administrators (DBAs) are classified as *internal threats* since they are involved into the infrastructure and are a part of it. Another wide-spread type of an internal threat is a malicious authorized personnel but this is out of the scope of this work since it is an issue that is supposed to be tackled using primarily non-technical approaches, and is more of a human resources competency. Cells of the table show some examples but the list of real threats is not limited to that.

⁸Malicious modification or destruction of rivals' data

First, the threat model should be established; second, the system and all its parts should be designed and analyzed with regard to the range of considered possible threats as per the threat model.

Processing of Encrypted Structured Data

It is required to determine what general approaches are there to process encrypted data, and specifically structured data, and analyze if these approaches or their fusions are suitable regarding the constraints that we have considering the research objective.

System Topology

By the nature of the research problem that is considered in this work, there always are several agents in the system: with varying purposes, capabilities, and levels of trust. At the very least, there is always a trusted data owner and an untrusted database server; there might or might not be other agents. It is needed to develop a topology for our model that suits the requirements and is efficient and economical (i.e., does not require unfeasible resources).

Abstract Query Execution Model

As the proposed approach states, the aim is to first develop a general abstract methodology of building privacy-preserving DBMSs, which is to a certain extent independent of specific data processing primitives. A prevalent part of such DBMS is a query execution model. Having established general rules (e.g., that users interact with the DBMS using a SQL-like language) and general properties of data processing primitives (e.g., data inputs and outputs, constraints on the results of processing, etc.), it is required to construct a concise model of execution of queries that satisfy the assumptions and rules and combine the primitives in arbitrary ways as long as it falls under the model constraints.

ACID Properties and Transactions

As soon as a database model leaves an ideal environment and is considered from a point of view of practical application it faces many well-known issues, among which are concurrent users, network, hardware, or software failures, and many others. Typically, a non-distributed DBMS is considered reliable for general practical application if it maintains ACID properties:

Atomicity If a transaction or a query fails mid-execution due to whatever reason, then all changes are rolled back and the database state is left unchanged.

Transactions and queries could only be executed in full or not executed at all.

Consistency Any transaction or query brings the database from one valid state to

another according to all defined rules including, but not limited to, constraints, cascades, triggers, and any combination thereof.

Isolation Concurrent execution of queries and/or transactions results in a system state that would be obtained if they were executed serially, i.e., one after another in the order they were received by the DBMS.

Durability Once a transaction has been committed or a query has been executed, all the modifications to the data will persist even in the event of power loss, crashes, or any other errors (except a data storage failure).

It is required to investigate how the model behaves in a non-ideal environment (i.e., with possibility of concurrent users and failures), and ensure that the query execution model is adapted to maintain ACID properties for both individual queries and transactions.

Practical Modules for the Query Execution Model

As the abstract model is defined and analyzed, next step needed is to develop approaches to creating and then create some of the *building blocks*: modules that implement certain data processing primitives. These blocks might be such essential and fundamental data processing primitives as equality checking, which is required for a majority of higher level data processing operations; addition/multiplication of numeric data; or some other fundamental, basic operations.

Empirical Analysis

Lastly, it is required to create an implementation of the theoretical model and plug it into a practical system. This would require investigation of “bridges” that connect the model implementation with real agents: an interface for user interaction that is capable of taking queries from the user and feeding him back the results; an interface to the database that implements storage and processing of the data upon a command from the model implementation; etc. The resulting system should be subjected to a thorough analysis.

1.4 Thesis Organization

This work is organized as follows. Chapter 2 considers evolution and current state of the art in the area of encrypted data storage and processing, and related areas. Chapter 3 considers high-level questions of a system intended for encrypted data processing in the cloud. Chapter 4 is dedicated to an abstract, universal theoretical model of encrypted query execution. Model behavior in non-ideal and multi-user

scenarios is analyzed in Chapter 5. Binding of the abstract model to a practical database system is considered in Chapter 6. Experimental results are presented in Chapter 7. And lastly, Chapter 8 is providing conclusions to this work.

Chapter 2

Related Work

This chapter covers most relevant and influential works related to storing, searching, and processing encrypted data in remote databases and data storages as a whole, and different individual aspects such as key management, data consistency audit, access control, search and querying, multi-user access, etc.

2.1 Definitions

This and following chapters contain problem statements, definitions, and discussions relying on basic concepts widely used by researchers of the encrypted structured data processing. Before we proceed, these basic concepts should be properly defined, including a definition of such foundational for this work concept as a *relational database*. Generally speaking, a relational database is a set of *relations*, each of which is a set of *tuples*, which, in their turn, are sets of *attributes*' values. Attributes of each relation are defined by a *relation scheme*.

Definition 2.1 (Domain). Domain of a certain variable¹ v is a set D of the values it could possibly contain. ◁

Definition 2.2 (Attribute). Attribute is a pair $\langle \text{Identifier}, \text{Domain} \rangle$. ◁

Definition 2.3 (Relation schema). Relation schema is a finite and non-empty set of attributes. ◁

Definition 2.4 (Tuple/Record). A tuple or a record of a certain relation schema S_R is a tuple of pairs $\langle \text{Identifier}, \text{Value} \rangle$ such that **1)** for each attribute $A \in S_R$ there is exactly one pair with an identifier that is equal to the identifier of A in the tuple; and **2)** in the tuple there are no pairs with an identifier such that there is no attribute with the same identifier in S_R ; in other words, there is a one to one correspondence

¹Any variable. In our case the variable is a cell in a database table.

between pairs in a tuple and attributes in S_R . **Value** of an attribute must belong to the **Domain** of this attribute. $\triangleleft$

In this work they are mostly referred to as “records” rather than “tuples”, since the term “tuple” is often used in the following sections in its mathematical meaning—an ordered list of elements.

Definition 2.5 (Relation). Relation is a set of records with same relation schema. $\triangleleft$

We also define a function $\text{schema}(\cdot)$ that returns a schema for a given relation.

Definition 2.6 (Database). A database is a collection of relations. $\triangleleft$

Relations are often referred to as “tables”, tuples—as “records” or “rows”, and attributes—as “columns”. It is worth noting that according to this set of definitions there could not be two relations with same relation scheme, though practical DBMSs usually allow users to have several relations with identical schemes and are still able to distinguish them since each relation is given a unique identifier (name) when created. Manipulation with relation schemes, relations, and tuples they consist of is usually done using a query language, the most wide spread of which is the SQL² language and its dialects (e.g., Transact-SQL by Microsoft, PL/SQL by Oracle). Those components of a query language that are interesting in the scope of this work are *Data Definition Language* (DDL), *Data Manipulation Language* (DML) and *Transaction Control Language* (TCL). DDL is used to create, modify, or destroy relation schemes; DML is used to create, modify, destroy, or retrieve tuples; TCL is used to process a set of DML queries as a single transaction.

2.2 Encrypted Search and Querying

One of the most demanded tasks for encrypted databases is data querying, especially searching. Without special efforts in that direction querying encrypted data is only possible if the whole encrypted database is transferred to the local machine and decrypted—then the user can query the decrypted database. This obviously causes a severe performance bottleneck, which could be addressed in many ways, which are systematically considered in this chapter.

2.2.1 Types of Encrypted Search and Queries

For this study on encrypted search the approaches, scenarios and types of encrypted search and querying are classified into groups. Solutions for encrypted search could be

²SQL is often referred to as an acronym for Structured Query Language, although “SQL is not an acronym for anything” (from O’Reilly “Learning SQL”).

aimed at a *single user* scenario, where there is only one concurrent database session, or *multi-user* scenarios, where there are several concurrent users eligible to execute queries, who potentially have different access rights. The queried data itself could be either *structured* (e.g., a relational database) or *unstructured* (encrypted file system, non-relational database, etc.). Different approaches to querying encrypted data rely on different underlying technologies: some are applicable to the data encrypted with *symmetric encryption*, some—to the data encrypted with *asymmetric encryption*. In certain cases it is possible to use *indices* for large amounts of encrypted data, and sometimes not so the search procedure needs to go through all data values one by one, i.e., perform *full-domain search*. Finally, searching and querying techniques differ by the subset of supported features from the very *basic searching* for constant value to *complex compound queries* with range checks, joins and other useful capabilities.

Some of the works selected for the overview in this chapter are those, which made the biggest influence in this area of research [11, 14, 19, 34, 38, 43, 67]. Others were selected since they illustrate well the evolution of the topic and current state of the art.

Works that are presented in this section could be organized in a taxonomy showed in Figure 2.1. In this section we examine what directions were researched and which approaches were selected by researches to solve the problems of encrypted search. Evidently, not all combinations of parameters have been paid attention to by the researchers.

Song *et al.* states 4 most important properties a query on encrypted data should have [67]:

Provable secrecy Untrusted server should not be able to learn anything about the plaintext when only given a ciphertext.

Query isolation Server should not be able to learn anything about plaintext values from the search query results.

Controlled searching Untrusted server should not be able to search for an arbitrary word without user’s authorization.

Hidden queries User may ask the untrusted server to search for a secret word without revealing the word to the server.

Multi-User vs. Single User Scenario

Generally, single user scenarios are much simpler, whereas multi-user scenarios may introduce additional issues. For example, there could be individual access rights

Figure 2.1: Taxonomy of notable works on encrypted search

for every user and, since we do not trust the perimeter defense (e.g., access control lists), we want our data to be encrypted in a way that even if somebody had accessed the restricted data, he was unable to decrypt it. Another concern is revocation of access from particular users. From the performance prospective, ways to invalidate the user's keys when access is revoked without re-encrypting database are needed, as users with revoked access (who, probably, have left organization, and are no longer bound by information security policies) can not be expected to keep their keys in secret forever.

Single User Scenario Single user scenarios do not necessarily mean there will always be the only user. It more likely means that users are indistinguishable from each other in the sense that they all have same access rights, same keys, same privileges in the system. In certain scenarios this is good enough and there is no need in more complex approaches, e.g., a remote file storage for a small work group.

Every member of the group can access and add, remove, or edit files there with same privileges, but they do not want the hosting provider or a malicious intruder to be able to peek the files. It is often assumed that no more than one user is using the system at any given moment.

In other cases the system itself is *meant* to be used by the only one user, like a personal email service. There is usually only one person who has access to the email and has all the privileges, so there is no need in maintaining multi user setting.

Single user solutions, in general, rely on a symmetric encryption as there is no need in a more complex asymmetric approaches. They are usually pretty straightforward—data is encrypted with certain granularity and stored at a remote host (be it a database or anything else). The simplest implementation of a single user database would just encrypt every value of data with deterministic symmetric encryption and store them in the database as it was plaintext data. Whenever user wants to find records with certain value, he could just encrypt the value with the same key and then perform a lookup for all records containing this value’s ciphertext. This solution could even utilize database’s indexing capabilities. Of course, this solution is very basic and is vulnerable to statistical attacks, for instance. On the other hand it is simple and fast and is used with just little modifications and additions in some works [59].

Multi-User Scenario Multi-user scenarios are generally more complex than single user ones. This is caused by a necessity to provide several more features.

- Amount of data accessible by each user could be different. Some data could be accessible by several users, some—by only one specific user. A user should not be able to decrypt data to which he has no access. At the same time, he should be able to search and query freely through all the data he has access to.
- Generally, adding or removing a user to/from the system should be simple and fast. Ideally, it should take $O(1)$ to do any of these operations.
- After a user is removed from the system his keys should no longer be valid, i.e., even if he (or anybody who appeared to know his keys) somehow gains access to the encrypted data, he should not be able to decrypt it.
- System should allow concurrent reads/writes and process them correctly.

There are two main groups of solutions providing proper multi-user access to encrypted data—based on a symmetric encryption and based on an asymmetric (public key) encryption. Symmetric solutions tend to have more flexibility and

potential for systems where users have relatively same privileges. Asymmetric ones are better suitable for situations when among users there are *super*-users with highly different tasks and privileges. In more details these two approaches are considered below.

To be able to give differentiated access to the data some special efforts need to be taken. Dong *et al.* proposed to use an RSA-based *proxy re-encryption* to do that [27] (see Figure 2.2). Proxy re-encryption makes it possible to keep the data encrypted with a master key and give users only a share of the master key (each user has unique share), and the complimentary share is kept by a DBMS. Shares of the master key, on their own, has no ability to encrypt or decrypt data, but once they are combined it becomes possible to fully operate the data. Typical workflow with proxy re-encryptions does not reveal the master key neither to user nor to DBMS. This approach also gives an opportunity to easily revoke/grant access from/to users. Procedure becomes as simple as create two shares (one for a new user and one for the proxy) of a master key to grant access or to instruct proxy to destroy a share of master key corresponding to certain user to revoke his access.

Later in 2006 Yang *et al.* proposed a specific way to store redundant data for searching purposes along with the ciphertexts of the data values themselves [78]. The sole idea of storing additional data for searching purposes was first proposed by Hacigümüş *et al.* in one of the most important papers in this field [38]. Evolution of the idea of proxy re-encryption from the works by Dong in combination with ideas of Yang was later explored by Hung *et al.*, which is one of the more recent works on the topic [44]. Approach proposed by Hung provides a very high level of security by introducing several proxies instead of one as it was in the done by Dong, so that even if a malicious user colludes with $(n - 1)$ proxies out of total n , the data is still safe. Solution is transparently scalable to any amount of proxies and gives some flexibility, making it possible for trusted users to use less proxies in order to improve performance.

Considerations of the approach to performing encrypted search that is used in this work are provided in Chapter 6, and propose a further evolution of this branch of research, making it possible to perform more complex relational queries than just basic search queries, and providing better and more flexible security.

Symmetric vs. Asymmetric Setting

There are two major types of encrypted systems—those based on symmetric encryption schemes, and those based on asymmetric schemes. The choice between the two of

Figure 2.2: RSA-based proxy re-encryption scheme³

them depends on the intended workflow of the system, as the two types of encryption schemes have different traits, including the performance and security. Such properties as typical workload profile, structure of the database clients, and many others may influence the choice between the two.

Symmetric Setting The first significant work on encrypted search in a symmetric setting was done by Song *et al.* [67]. In a symmetric setting, as it was shown in this seminal paper, only an authorized user can produce meaningful queries to the database, add/modify data in the database, and decrypt query results. Scheme proposed by Song only supported full-domain search and relied upon a word by word encryption, but further works suggested more options.

Two researchers had later examined the ways to build a secure index to search through encrypted data in a symmetric setting. First, two important directions were investigated by E.-J. Goh. He has formally introduced a property of indistinguishability against the chosen keyword attack, the so-called IND-CKA property—semantic security against adaptive chosen keyword attack⁴; and a stronger form of the IND-CKA property—the IND2-CKA property [34]. Also, Goh proposed two secure indexing schemes that use Bloom filters, which are both compact and efficient, and provide probabilistic search. Then, unlike previous researchers, Curtmola *et al.* proposed using an inverted index instead of a forward index [23]. They studied *trapdoor* privacy, which was out of the scope of previous works, and eventually improved the security definitions given previously by E.-J. Goh.

Strictly speaking, a *trapdoor* or a *trapdoor function* is defined as follows:

³Figure is taken from a work by C. Dong *et al.* [27].

⁴IND-CKA property is considered in more details later in sections related to indexed search.

Definition 2.7 (Trapdoor function). If f is a trapdoor function, then there exists some secret information y , such that given $f(x)$ and y it is easy to compute x . $\triangleleft$

On the other hand, when one speaks of an encrypted search by *trapdoor*, he means a similar yet different thing. A trapdoor for a value w is some information based on a value w , which can only be constructed by authorized parties (e.g., a user with search privileges), which can then be transferred to the untrusted cloud database server where it will be used to find encrypted records, containing value w . In order to be sure that only authorized parties can construct a trapdoor it needs to be also based on some secret information, which is only known to those authorized parties. This could be formalized as follows:

Definition 2.8 (Trapdoor for encrypted search). A trapdoor for encrypted search $t = f(w, K_u)$, where w is the value the trapdoor is being constructed for; K_u is a secret known to data owner and authorized user u . Function f must be a one-way function. Trapdoor t must not reveal the value w unless K_u is known. $\triangleleft$

In the end of this definition it is said that trapdoor should not reveal the original value if secret is not known. This is a minimum requirement. Usually, trapdoor should not also reveal the value w if secret is *known*. Moreover, it should leak as little information about the original value as possible.

Let us assume there is a simple database consisting of one table with only one column containing different words. Database is hosted at a cloud service provider and thus is untrusted, so the data owner encrypts words with probabilistic encryption (same word w could result in different valid ciphertexts). In order to be able to search words a trapdoor is generated for each word and is stored along with the ciphertext in the database, e.g., a hash $h = \text{hash}(w||k)$ of the word w concatenated with a secret key k is computed. Now, when a querier wants to test if the database contains a specific word w_0 , he computes $\text{hash}(w_0||k)$ of w_0 concatenated with the key k , and tests if any of the words in the database have the same trapdoor.

Song *et al.* had only considered single user usage scenario. Further works have made attempts to extend the solutions to work in a multi-user setting. E.g., Curtmola *et al.* presented a solution where multiple parties can issue search queries [23]. The party who produces the data—which is then queried—is able to dynamically manage who is capable of issuing meaningful search queries. As it was proposed by Bao *et al.*, the system gets a new entity—*user manager* [8]. User manager’s purpose is to manage users’ rights to access search capabilities. In the suggested scheme all users are capable of creating searchable data and searching through it. While having

some obvious benefits, this approach also has significant limitations. First of all, the newly introduced user manager can issue the search queries and, moreover, decrypt the received data, which leads to the necessity to have a *fully trusted* user manager. Apart from that, the user manager is needed to produce index, which may become a performance bottleneck.

Golle *et al.* developed two similar schemes for encrypted search [36]. Suggested schemes give the users an ability to bind several keywords to any data item (document) in the document-oriented database (such as an email server), and then do a conjunctive keyword search. While the capabilities of the scheme proposed by Golle are not very extensive, a wide and comprehensive security analysis is done for the solution.

A rarely studied problem of querying multidimensional data in encrypted form was considered by Hore *et al.* [42]. His work appeared to be an extension of his own ideas and ideas proposed by Hacigümüş, in order to support multidimensional data: the secure indexing tag is computed based on bucketization. Another research issue, which appeared to be rarely examined is fuzzy search, i.e., matching *similar* keywords so that the system is able to return correct search results even if a user has misspelled the search term. This issue was partially mentioned in works of Goh [34], and was extensively investigated by Li *et al.* [48], Raykova *et al.* [61], and C. Bösch [16].

Asymmetric Setting According to the key work on encrypted search in an asymmetric setting done by D. Boneh *et al.*, in such a setting every entity is eligible to produce searchable encrypted data without any explicit user authorization, but only the user can issue meaningful search queries and decrypt the data [11]. The scheme proposed by Boneh is often referred to as PEKS (Public Encryption with Keyword Search) and has lots of severe limitations. The PEKS scheme only covers a single keyword encryption and is only able to work with exact matching. Later work by Boneh extends the system to support conjunctive keyword search along with introducing a support of range and subset queries [14].

An interesting approach to the problem was selected by Bösch *et al.* While their basic solution is built on top of a symmetric encryption scheme, when they extend it to the multi-user setting, they switch to using asymmetric public key encryption schemes [17].

Categorization Intended usage scenario—in a multi-user environment or with only a single user—and choice between symmetric and asymmetric encryption schemes have a strong impact on each other. In Table 2.1 some of the key works on encrypted search are categorized based on these two system properties.

	Multi-user	Single user
Symmetric	[38, 42, 43, 53, 67]	[8, 17, 23, 27, 36, 44, 76–78]
Asymmetric	[17]	[11, 14]

Table 2.1: Categorization of works by usage scenario vs. encryption scheme type

Structured vs. Unstructured Data

All works related to the secure cloud data storage and/or processing belong to one of the two well-separated clusters: structured data and unstructured data. A very different sets of research issues are faced by the works from these two groups. Solutions concerning unstructured data are mostly interested in storing, retrieving (i.e., searching), and sharing documents. Approaches to query encrypted structured data mostly examine ways to perform OLAP (Online Analytical Processing) and OLTP (Online Transaction Processing) tasks.

Search task for the unstructured data is often aggravated by a necessity to perform a full-text search through remotely stored encrypted documents. Additional efforts are needed to provide wild-card search. On the other hand, querying structured data often implies performing relational operations (e.g., joins), range queries (i.e., test whether certain encrypted value is less or greater than some constant value or another encrypted value), and set operations (e.g., union, intersection, subtraction).

Unstructured Data One of the fundamental works in querying unstructured data was done by Song *et al.* [67]. It introduces a solution applicable to secure remote file storages or email services. Authors start with a very basic solution and iteratively improve it in order to make it compliant with 4 properties of secure search that were considered above. The proposed solution is able to perform a full-text search through documents (i.e., emails in the email server). Building a simple index for the data in order to make search faster is also addressed.

One of the most important ideas in Song’s work is *probabilistic search*. This means that results of a query that is searching for documents containing a word W contain all the documents with the word W as well as some (adjustable) amount of false-positive documents. This idea was then widely incorporated by later researches: probabilistic search makes it much harder for a curious adversary to perform a sensible analysis of what was returned to certain queries, and execute statistical attacks.

Another fundamental work in querying unstructured data by E.-J. Goh has presented a comprehensive analysis on building secure indices for unstructured data

storages [34]. According to Goh, a *secure index* is a data structure that allows only a querier with a trapdoor for a word x to test in $O(1)$ time if the index contains x . The index should not reveal any information about the data without valid trapdoors, and the trapdoor, in its turn, could not be generated without a secret key.

The author suggests building an efficient and secure (in aforementioned sense) index using pseudo-random functions and Bloom filters. Bloom filter is a probabilistic data structure with a purpose of testing whether an element belongs to a set or not. Its probabilistic property implies that false positive results are possible; however, false negatives never occur. I.e., a query returns either “probably in the set” or “definitely not in the set”. This is advantageous in certain scenarios. The structure is efficient in terms of space but has certain limitations: elements could only be added to the set, not removed (there are modifications that mitigate this limitation); the more elements are added, the higher becomes the probability of a false positive response.

A comprehensive analysis in the paper shows the full procedure of construction a secure index using Bloom filters. The author also proves that such index could be used to test the membership without revealing the set elements. Moreover, a general way to calculate optimal parameters for Bloom filter is described and proved. He also examines in details the procedures of initial setup and altering the stored data, and demonstrates how a simple search could be performed, how to compose several simple searches with a Boolean operations (AND, OR, NOT), and even how to run queries with wild-cards and regular expressions.

Solutions suggested by E.-J. Goh are relatively universal, independent, and modular, which makes them easy to integrate into existing systems working with unstructured data, and with some adjustments and modification they could also be incorporated into systems aimed at working with structured data to facilitate searching and querying.

Structured Data Encrypted storages for structured data are more aligned with the topic of the current research. The most fundamental work on querying an encrypted storage of structured data was done by Hacigümüş *et al.* His work has fully covered an implementation of an encrypted relational OLTP database [38].

His work introduces a very important for future explorations in this direction idea of storing an additional data that facilitates search along with the encrypted data. The paper suggests an encryption of data at a tuple level and makes it possible to use a predefined set of attributes in queries, e.g., *all* attributes. We can see how the plaintext data from Table 2.2 is converted into encrypted data with supportive

<i>eid</i>	<i>ename</i>	<i>salary</i>	<i>addr</i>	<i>did</i>
23	Tom	70 000	Maple	40
860	Mary	60 000	Main	80
320	John	50 000	River	50
875	Jerry	55 000	Hopewell	110

Table 2.2: Example plaintext data (source: [38])

<i>etuple</i>	<i>eid^S</i>	<i>ename^S</i>	<i>salary^S</i>	<i>addr^S</i>	<i>did^S</i>
1001001001000010001...	2	19	81	18	2
1000101001001001101...	4	31	59	41	4
1111100100101001000...	7	7	7	22	2
1010101010111111001...	4	71	49	22	4

Table 2.3: Example encrypted data (source: [38])

values in Table 2.3.

Another key idea proposed by Hacıgümüş *et al.* is probabilistic search, which was first introduced for unstructured data by Song. In both works probabilistic search supposes final filtering of query results on a client side after decryption, when it is easy to determine items, which do not actually satisfy the query predicate. Probabilistic search in the scheme of Hacıgümüş is achieved using another fundamental concept—*bucketing*, which was then widely adopted by many researchers [26, 37, 42, 43, 53, 69, 72]. What can be seen in Table 2.3 in columns *eid^S*, *ename^S*, *salary^S*, *addr^S*, and *did^S* are identifiers of buckets corresponding to the original plaintext values. Several plaintext values could be in the same bucket but no value could be in several buckets. Knowing which bucket the value that the querie is interested in belongs to, he can instruct the DBMS to return all tuples with certain attribute in that bucket. After decryption he would need to filter out other values, which also appeared to belong to that bucket.

Authors suggest different mapping functions for bucketing of plaintext values for different purposes. For example, to make it possible to perform range queries, authors suggest using order-preserving bucketing. In this work the most simple *equi-width* partitioning of domain into buckets was used.

Equi-width bucketing was later revisited by Hore *et al.* [43]. This work presents a comprehensive analysis on how to perform a full set of relational operations on encrypted data, which makes it possible to implement any dialect of SQL on top of that system. On the other hand, some relational operations could experience a severe

performance drop caused by the buckets being too large—e.g., join operations are very sensitive to that. An analysis on this matter that was conducted in the scope of this project showed that encrypted join operation on big data sets could be executed slower by a factor of hundreds as compared to plaintext join operation if buckets are big, and the more data there is, the bigger this factor becomes.

Not all solutions use bucketing to implement structured search. Popa *et al.* used a deterministic encryption wrapped into a probabilistic encryption layer in their project CryptDB [59]. As soon as the database receives instructions to search through values, it decrypts the outer layer and exposes the deterministic encryption which is easily searchable. Benefits and limitations of such approach are discussable, and are considered later in this chapter. Another approach proposed by Hung uses deterministic encryption with manual adding of probabilistic properties to obfuscate data [44]: each data value v is concatenated with a unique random string r and is then encrypted with a deterministic encryption $E_k(\cdot)$. Database then stores $\langle E_k(v||r), r \rangle$. To search through the database for a certain value v_0 , the system goes through each record, takes the corresponding random string r_i , and tests if $E_k(v_0||r_i) = E_k(v||r_i)$. If yes, then $v = v_0$.

Indexed vs. Full-Domain Search

A direct approach to the search is to do a full-domain lookup (sometimes referred to as *full scan search*). The limitation of this approach is speed—full-domain lookup works in time $O(n)$. On the other hand, index-based search provides *much* better performance, e.g., a lookup using a widely used B-tree or B+-tree, on average, works in time $O(\log n)$. For example, for a database containing 1 000 000 records a full-domain lookup will take 1 000 000 comparisons and an indexed lookup will take only about $\ln(1\,000\,000) \approx 14$ comparisons.

Full-Domain Search Full-domain search goes sequentially, item by item, through all the data and does the needed comparisons. While it has obvious performance limitations, it has much more flexibility than indexed search. Unlike the indexed search where the index restricts the variety of possible queries, in the full-domain search the query predicate can contain any set of checks and operations, which could even change on the fly [68].

In some cases index for encrypted data could contradict with the security requirements for the encrypted data storage. For example, in the model suggested by Hung the requirement is that two ciphertexts were different even if the plaintext values are equal (i.e., *probabilistic* encryption), so that the adversary is unable to tell which

data items in the database are equal, and thus is unable to perform, e.g., a statistical attack [44]. Clearly, an index (in its usual form) will destroy the efforts to make the encryption probabilistic. A possible compromise solution is to use bucketing, that proposed and developed by many researchers [38, 42, 43]. It would aid a much faster search (slower than indexed search, but much faster than full-domain search) but still prevent the adversary from learning equal items in the database. Moreover, bucketing technique allows us to rely on a database built-in indexing mechanisms, which are usually faster than any external implementations of indexing mechanism.

In many cases (this mostly affects encrypted storages of structured data) full-domain encrypted search could be aided with some additional data structures, which are not “indices” in the strict formal meaning of this word, i.e., they do not allow to locate all records satisfying the search predicate rapidly. As long as this is not caused by a lack of knowledge on how to build an index, but by the necessity to conceal as much information about encrypted data as possible, these additional data structures could also be called “indices”. Indices in this extended meaning were suggested in works by Yang [77, 78].

Indexed Search There is a huge difference between indices for structured and unstructured data. Indices for structured data are intended to help to find all records *equal* to some value, whereas indices for unstructured data are intended to help locate all records *containing* some value. There are two basic ways to build an index for unstructured data—*forward index* and *inverted index*, which are illustrated in Figures 2.3 and 2.4 (source: [68]).

<i>Item identifiers</i>	<i>Keywords</i>
Item 1	keyword1, keyword3, keyword5
Item 2	keyword5
Item 3	keyword2, keyword4
Item 4	keyword2, keyword3
Item 5	keyword1, keyword2, keyword4
Item 6	keyword4, keyword5, keyword6
Item 7	keyword2, keyword3, keyword4, keyword5

Figure 2.3: Forward index

A full and comprehensive study on indices for unstructured data was performed by E.-J. Goh [34]. In his work Goh defined a secure index and formulated a security model for indices known as semantic security against adaptive chosen keyword attack (IND-CKA). On the practical side he developed a construction of an efficient IND-CKA-

<i>Keywords</i>	<i>Item identifiers</i>
keyword1	Item 1, Item 5
keyword2	Item 3, Item 4, Item 5, Item 7
keyword3	Item 1, Item 4, Item 7
keyword4	Item 3, Item 5, Item 6, Item 7
keyword5	Item 1, Item 2, Item 6, Item 7
keyword6	Item 6

Figure 2.4: Inverted index

secure index called Z-IDX, which uses pseudo-random functions and Bloom filters. The work demonstrates the performance of constructed index, shows how searches over encrypted data are performed using it, and proves that constructed index is capable of handling arbitrary updates, and does not depend on encryption or compression applied to the source documents.

Categorization Two more aspects of encrypted querying—whether the search is indexed or full-domain, and whether the data to be queried is structured or unstructured—have been considered. Significant works in this field could be divided into these groups in a way shown in Table 2.4.

	Indexed search	Full-domain search
Structured	[8, 17, 38, 42, 43, 53, 77, 78]	[14, 27, 44, 76, 78]
Unstructured	[11, 23, 34, 36, 67]	[67]

Table 2.4: Categorization of works by type of data vs. type of search

Basic Search vs. Complex Queries

The most basic type of a query to the database is a simple search query involving only equality matching. Even this basic functionality could suffice for many real applications. Some of more sophisticated queries could be built on top of equality matching only, i.e., checking whether a value is a member of a set (IN operator in SQL) could be unfolded into series of equality matching. Many researches only considered equality matching [17, 44, 67, 77]. It is worth noting that there are two types of equality matching, which is well illustrated by these two queries: **1) SELECT * FROM t WHERE $t.a = 5$; 2) SELECT * FROM t WHERE $t.a = t.b$.** When we are dealing with encrypted data these are two very different tasks; some works [44] only support equality matching of type 1 and do not support equality matching of type 2.

A considerably more complex problem is to provide a *fuzzy* or *wild-card* search.

Fuzzy search retrieves not only records containing exact match but also records with similar words, i.e., query `SELECT * FROM t WHERE name = "Jean-Claude Van Damme"` would also have returned records where `name` is `Jean Claude Van Damm` or `Jean Claud Vandamme`, if the equality matching was fuzzy. Wild-card search gives users an ability to retrieve records to the whole set of possible search queries at once. E.g., if a user is not sure how the name of Jean-Claude Van Damme is spelled correctly, he could make a query, in which all pieces he is unsure about are replaced with a wild-card symbol “%”: `SELECT * FROM t WHERE name LIKE "Jean%Clau%Dam%"`. Wild-card and fuzzy queries over encrypted data are a non-trivial issue. The most wide-spread approach to make it possible is to encrypt all closest (e.g., in terms of Levenshtein distance) variations of the word along with the original one: for a word “cat” it would be “aat”, “bat”, “cat”, “dat”, ..., “cbt”, “cct”, ..., “caa”, “cab”, ..., “act”, “cta”, etc. To support wild-card search one would need to encrypt all possible wild-card queries, which the current word will satisfy. For a word “cat” that would be “cat”, “ca*”, “c*t”, “*at”, “c*”, “*t”, “*” (we assume that wild-card symbol “*” means any character repeated 1 or more times). Several researchers have studied fuzzy and wild-card encrypted search [16, 34, 48, 61].

Fuzzy and wild-card search is suitable for textual values. When it comes to numeric data, the next step after equality search are range queries. Range queries involve greater-less comparisons between numeric values and are usually more demanded in querying structured data. Two main approaches used for range queries are **1)** order-preserving encryption; **2)** order-preserving bucketing. Order-preserving encryption (OPE) makes it possible to perform direct comparisons between the two encrypted values since for the OPE function $E_K(\cdot)$ if $E_K(x) < E_K(y)$ then $x < y$. This approach was used, for example, in works by Popa [57–59]. OPE, in general, gives an adversary an ability to perform statistical attacks and learn additional information from ordering. If this is not acceptable, order-preserving bucketing could be more appropriate. It allows users to have a combination of probabilistic search and range queries. This approach was used in works by Hacıgümüş *et al.* [38] and Hore *et al.* [42, 43].

One of the crucial operations on relational databases are joins; namely, equi-joins and range-joins. If data storage is capable of equality checks of type 2, then equality joins could be built on top of that. In the same way, range-joins could be built on top of range queries. Typical issue with joins is that different tables might be encrypted with different encryption keys in order to improve security. That makes it difficult to perform type 2 equality checks and range queries on the data from different tables,

which is in the road of executing join operations. Researchers usually need to invent complex workarounds to preserve ability to do arbitrary joins or allow only to perform joins on *predefined* sets of tables. Both approaches were investigated by Popa [59].

The most complex research issue is to give users an ability to perform *arithmetic operations* on encrypted data. This would give a green light for the encrypted OLAP, which is surely a highly demanded feature nowadays when data leaks cost more and more to the enterprises. Sending data back from the database server to the client's computer so that he decrypts the data and performs needed computations is extremely inefficient and could even be impossible, e.g., if a client uses a thin client or a mobile device to issue the query, and the amounts of data are big. Having a trusted powerful proxy between the user and the DBMS, which would perform all computations is also inefficient and diminishes the benefits of moving the database to the cloud. But in certain scenarios this approach could be viable. The more general approach is to use a *homomorphic* encryption scheme, which allows the database to perform arithmetic operations directly with encrypted data. For example, the Paillier encryption scheme $P_K(\cdot)$ makes it possible to perform an operation of numeric addition over encrypted data: $P_K(x) \times P_K(y) = P_K(x + y)$ [54]. This is used by the CryptDB project [57–59] to perform addition of encrypted data in the database. There are also encryption schemes for multiplication of encrypted data, such as ElGamal. Encryption schemes, allowing for both multiplication and addition of encrypted data are called *fully homomorphic encryption schemes*. Though they exist, they are extremely inefficient to the point of being impractical, which makes it difficult to provide the ability to perform queries that combine operations (like `SELECT a + b * c FROM t`), which could be essential for many OLAP tasks.

Aggregation functions and additional query clauses are often used in querying databases. Most simple of them—like `COUNT`—do not need any capabilities from database. The `GROUP BY` statement and `DISTINCT` query modifier need equality checks of type 2. The `LIKE` operator needs a wild-card search. Such aggregation functions as `MIN` and `MAX`, as well as `ORDER BY` clause need the ability to perform range queries. Aggregation function `SUM` obviously needs a homomorphic encryption if we want it to be executed by the database. Such function as `AVG`, ideally, would need a fully homomorphic encryption because for this function to execute it is required to calculate a sum of all values and divide it by their amount n (or multiply by $1/n$), but in fact could be implemented using only `SUM` and `COUNT` functions, and the actual division would be performed locally, which does not create any noticeable

overhead neither in data transfer nor in calculations. One example of implementing aggregation functions was given by Mykletun and Tsudik [53], who demonstrated how to execute aggregators `SUM` and `COUNT` in the encrypted DaaS model proposed by Hacıgümüş [38].

2.3 Authentication, Authorization, Access, and Key Management

This chapter considers research issues related to the key management, and user authentication and authorization. First of all, the difference between authentication and authorization should be clarified.

Definition 2.9 (Authentication). Authentication is an act of confirming the claimed identity of an entity. ◁

Usually, the entity is a person, and authentication is performed by forcing the person claiming for a certain identity to provide some information that could only be known or created by the true holder of this identity. This could be a secret password or a one-time password generated by a hardware token.

Definition 2.10 (Authorization). Authorization is an act of granting certain privileges to an authenticated entity. ◁

It is clear that the process of authentication precedes the process of authorization. Each user or a group of users could be authorized to perform different sets of actions. Exempli gratia, some users could be only allowed to search and query data, others could also be granted rights to create/modify data, and DBAs could additionally be authorized to modify relation schemes but lack rights to modify or create data.

These tasks are simple or even unnecessary while encrypted database system is used by the only one user (or a group of indistinguishable users). But as soon as there are several users and they want to share data with each other, these problems arise. The simplest way to share the encrypted data is for one user to share the secret key with another user so now they both could encrypt and decrypt data. Such an approach increases the risk of unauthorized data access since it is assumed that no one except data owner could be fully trusted and user with whom the master key is shared could (possibly unintentionally) reveal it to an adversary.

Access control systems are often referred to as “perimeter defense”, which is an important part of any practical system, but can not be the only defensive technique used, since as soon as the perimeter defense is breached, all confidential information

Figure 2.5: Defense layers

is fully exposed. The next level of defense is encryption, which brings issues related to key management (see Figure 2.5). Authentication and authorization may be implemented as a part of the perimeter defense but additionally the encryption model may be developed in such a way that there would be no need in authentication and only authorized users would be able to perform manipulations with data. That way even if perimeter defense is compromised, breached, or downed, the confidential data is still secure unless the adversary has got the encryption keys. It is even possible to develop a model in such a way that in case the outer layers of defense are downed, the system could still perform some tasks.

Authentication and authorization subsystems are not considered in this work, which rather focuses on the problem that is going to be formulated now. For each value v_i from a set of plaintext values $V \equiv \{v_1, v_2, \dots, v_n\}$ there is its encrypted representation $E_{v_i} \in E$ in the database. E_{v_i} could contain a set of ciphertexts for the original value v_i encrypted with different schemes and also could contain some additional information that is used for certain operations. Every $E_{v_i} \in E$ supports a certain subset of operations $O_{E_{v_i}} \subseteq O$, where $O \equiv \{o_1, o_2, \dots, o_p\}$ is a set of all supported by the DBMS operations on encrypted data. Each user $u_i \in U$ is authorized to perform a set of operations $O_{E_{v_i}}^{u_i} \subseteq O_{E_{v_i}}$. The problem is to develop a model of encryption $v_i \rightarrow E_{v_i}$, and propagation of encryption and decryption keys to the users so that every user u_i would be able to perform all operations $O_{E_{v_i}}^{u_i}$ and only those.

Main difficulties in key management appear when a shift happens from a single

user system ($|U| = 1$) to a multi-user system ($|U| > 1$), and the system shifts from a static system to a dynamic system, i.e., the data could be not only read but also written [25].

Damiani *et al.* present an approach for the implementation of access control and key management for multi-user encrypted databases through selective encryption along with some empirical results [25]. Four entities are involved in the modeled situation:

- Data owner (person)
- User (person)
- Client (application)
- Server (application)

The research is based on some assumptions. First of all, the server is trusted to preserve data integrity. At the same time, the server is not trusted with the data confidentiality. The database is assumed to be encrypted on the tuple level, and the database itself is *static*, i.e., data could only be *read* from the database.

Proposed key management scheme is based on the idea that if two tuples in the database could be accessed by the same group of users with the same access rights (i.e., same *access groups*), then these tuples could be encrypted with the same shared key. On the other hand, the number of keys a client should know increases with the growth of the amount of such access groups, which leads to a large amount of information a user needs to know and, more importantly, keep secret. That is why the authors suggest an alternative solution—to use *key derivation* mechanisms along with a user hierarchy that is built based on the access lists.

Knowing access lists, it is possible to construct a directed acyclic graph (DAG) of access groups, which are then transformed into trees, since DAGs are a difficult data structure to work with. Keys are then derived from a master key using one-way functions. Experiments have shown suggested scheme to be a viable and easy to use solution (taking into account restrictions mentioned earlier).

In their research on the key management infrastructure in the cloud computing environment Lei *et al.* propose a general *cloud key management infrastructure* (CKMI) architecture, which is a collection of policies, procedures, techniques, and technology for managing all kinds of cryptographic keys in the cloud environments, which are used for various cloud applications' encryptions [46]. Authors claim that CKMI results in less costs for developing and supporting. CKMI consists of a *cloud key*

Figure 2.6: Cloud key management infrastructure⁵

management client (CKMC) and a *cloud key management server* (CKMS). CKMC is a part of one of three fundamental cloud service models: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), or Infrastructure-as-a-Service (IaaS) (see Figure 2.6). CKMS and CKMCs communicate to each other using a *cloud key management interoperability protocol* (CKMIP), which consists of three primary elements:

1. *Objects*: symmetric keys, asymmetric keys, certificates, etc.
2. *Operations*: actions taken with respect to an Object.
3. *Attributes*: properties of an Object.

Authors note that confidential data hosted in the cloud must be protected using a combination of access control, contractual liability, and encryption, which offers the benefits of minimum reliance on the cloud service provider and lack of dependence on detection of operational failures. According to authors, main issues and challenges in CKMI are as follows:

- Keys must themselves be protected in storage, transit and backup;
- Access to the keys must be limited to those who specifically need the individual keys. There should be also policies governing the key stores, which use the separation of roles to help control access;
- Secure backup and restore procedures should be implemented.

CKMI proposed by Lei is a general concept of a universal protocol of key management between different entities in heterogeneous cloud context, which is a typical situation for enterprise systems. Implementation of such protocol and infrastructure should guarantee some level of security in the key management security regardless of the specific encryption schemes and other details.

Dong *et al.* propose a scheme for a multi-user and dynamic searchable data

⁵Figure is taken from the paper by Lei [46].

encryption without involving shared keys. In the suggested model each authorized user has his own keys to encrypt and decrypt data [27].

Developed system exploits a *proxy re-encryption* scheme (see Figure 2.2). Proxy re-encryption schemes are flexible and could be built on top of various cryptosystems, e.g., ElGamal or RSA. The system investigated in the paper is based on the RSA encryption.

Still, designed system requires a trusted *key management server* (KMS) to operate, which seems to be at odds with using an untrusted data storage service, but the KMS requires much less resources and management efforts.

Except for developing a KMS architecture, adapting a proxy re-encryption scheme, and introducing internal communication protocols, authors also provide mechanism allowing a simple keyword search through encrypted data without revealing plaintext of the query to the untrusted storage server. In addition, such considerations as user revocation, secure channels to transfer RSA keys, and risks of collusion attacks are present in the paper. Performance measures performed by authors prove the right of proposed scheme to be considered for use in production along with other schemes.

This paper presents a searchable data encryption scheme, which does not require shared keys unlike all previously proposed schemes. Two constructions were proposed and both were formally defined and formally proved for security. Suggested proxy re-encryption scheme or similar constructions were later used in several researches [44,59].

Hacıgümüş *et al.* have studied general implications of 4 main stages of key management on a typical DaaS [39]. Examples use results of previous works by the same authors [38,42,43]. The mentioned four stages of key management are:

Key generation. This stage involves the creation of the encryption keys meeting the specifications of the underlying encryption techniques. Authors distinguish two types of key generation: *pre-computation*—keys are pre-computed and stored, they can be used directly in an on-demand fashion; *re-computation*—keys are not stored and generated on the fly from initial seed values (a key generation algorithm should be deterministic).

Key installation. After the keys are generated, the authorized users need to access and use them. This stage defines where and how the generated keys are stored. Authors propose a special data structure aimed at storing the keys—a *key registry*.

Key distribution. After a key is generated and stored in the key registry, it needs

to be transferred to the authorized user on his demand. This process is called *key distribution*. Depending on the situation and requirements, key distribution can be the responsibility of a client application, a trusted third party, or a server application.

Key update. A typical security requirement is a finite lifetime of every key used in the system, which leads to periodical key updates. Additionally, if a key compromise is suspected (or detected) an emergency re-keying is needed to be performed. Updating keys for the whole database usually takes a long time and could interfere with other transactions, and thus slow down the whole system. The work considers various ways to overcome this along with their benefits and limitations.

Key management usually highly depends on specific requirements, encryption schemes, and other details of a specific solution, thus only very practical works like the CryptDB project [59] consider specific implementations of the key management systems. Other works mainly focus on general issues, interaction between a key management subsystem and other parts of the system, performance issues, etc. These works form a set of recommendations for developing a key management system and highlight typical problems in real key management systems.

2.4 Auditability

Verifiability of data that is stored at untrusted servers is one of the important issues regarding cloud data stores. The storage service providers that are occasionally experiencing Byzantine failures may decide that hiding the fact that data was corrupted is more beneficial for them. Moreover, as some researchers fear, in an attempt to save their expenditure on storage space, the service provider might neglect to keep and *intentionally* delete rarely accessed data of non-key customers. On the other hand, cloud storage service provider could ignore a user's request to delete his data and while showing the data being erased still keep it. These concerns motivate the research in the area of techniques of data integrity audit and ensuring data deletion.

Figure 2.7 shows the model architecture. The architecture consists of four different main parts: a data owner, a user, a cloud server (CS), and a third-party auditor (TPA). Here the TPA is assumed to be semi-honest (in other words, it is *only* guaranteed that his main task, which is to audit data, is executed correctly and in full; he can still be trying to steal the data), and having the necessary resources and knowledge to assess

⁶Illustration is taken from the paper by Wang [70].

Figure 2.7: The architecture of cloud data storage service⁶

the security of a cloud storage on behalf of the data owner, when requested. In this scenario, the data owner may be either an individual or an enterprise customer [70].

The owner of the data pursues to verify that his data is stored securely. In order to do so, he delegates the TPA to perform data audit; however, he intends to keep data private from both the cloud storage provider and the TPA. As it was said earlier, the TPA is assumed to be a professional, reliable, and unbiased auditor, and thus having no incentive to collude with either the CSP or the owners during the auditing process. At the same time, the data should not leak towards the TPA through the auditing protocol. TPA's methods should be efficient enough for him to be able to process large amounts of data in a reasonable time.

Wang *et al.* present a scheme that is able to ensure storage correctness and localize data errors by utilizing a homomorphic token with distributed verification of erasure-coded data [71]. Such scheme is able to work with dynamic data storages allowing block update, delete, and append operations (but not the block insertion operation).

Model developed by the authors involves three network entities:

Client. An entity that has large data files to be stored in the cloud. Can be either an individual customer or an organization.

Cloud storage provider (CSP). Has significant resources and expertise in building and managing distributed cloud storage servers, owns and operates live cloud computing systems.

TPA. An optional entity, which has expertise and capabilities that clients do not have, is *trusted* to assess and expose risk of cloud storage services on behalf of the clients upon request.

Two types of adversary model is considered in the work—a *weak adversary* and a *strong adversary*:

Weak adversary. The adversary is interested in corrupting the user’s data files stored on individual servers.

Strong adversary. Strong adversary is the worst case scenario. It is assumed that the adversary is able to compromise *all* data storage servers and to modify data in the way that it is still consistent.

For the security and dependability of the cloud data storage under the aforementioned adversary models, authors aim to achieve following goals in the scheme design:

Storage correctness. The model is able to verify if the data was stored correctly and kept intact;

Fast localization of data errors. The model is able to efficiently locate a malfunctioning server if data corruption has been detected.

Dynamic data support. Storage could be verified for correctness when users actively modify, delete, or append the data files in the cloud

Dependability. Model is able to increase data availability in the presence of *Byzantine failures*, malicious data modifications, and server colluding attacks, i.e., to minimize the effect of data errors or server failures.

Lightweightness. Storage correctness checks require a reasonable and feasible overhead.

Work contains a thoughtful, detailed, and sound description of the full life cycle (file distribution preparation $\rightarrow$ verification tokens precomputation $\rightarrow$ correctness verification $\rightarrow$ error localization $\rightarrow$ file retrieval $\rightarrow$ error recovery) of working with data under the developed scheme [71, sect. III]. Further, issues related to providing dynamic data operations support are considered for the update, delete, and append operations [71, sect. IV]. In this work authors suggest to construct an insert operation on top of these three operations, which is possibly an inefficient approach.

In the later work the same authors (Wang *et al.*) have achieved to enable both public auditability and data dynamics including block insertion operation for secure cloud data storage [73]. To be more specific, the presented work

- Proposes an auditing protocol supporting for fully dynamic data operations (including block insertion, which is missing in most existing schemes);
- Enables batch auditing where multiple delegated auditing tasks from different users can be performed simultaneously by a *third party auditor* (TPA);
- Proves the security and justifies the performance of the proposed scheme.

Similarly to the previous work, the three different network entities involved in the system model developed by the authors are:

Client. Like previous work—an entity (individual customer/organization), having large data files to be stored in the cloud.

Cloud storage server (CSS). An entity managed by a cloud service provider, has significant storage space and other resources needed to maintain clients’ data.

TPA. Unlike previous work, the TPA is not optional.

According to Wang, a checking (auditing) scheme is called secure if **1)** there exists no polynomial-time algorithm that can cheat the verifier with non-negligible probability; **2)** there exists a polynomial-time extractor that can recover the original data files by carrying out multiple challenges–responses.

Except being able to perform block insertion it is also important for the scheme to be capable of auditing without actual retrieving the data blocks themselves. In order to achieve this the authors resort to previously published homomorphic authenticator technique⁷. Homomorphic authenticators are an unforgeable metadata generated from individual data blocks, which can be securely aggregated in such a way to assure a verifier that a linear combination of data blocks is correctly computed by verifying only the aggregated authenticator.

The scheme proposed by authors is built on top of the Merkle hash tree and bilinear mappings. A procedure for default integrity verification is shown in details in the paper along with an ability to handle fully dynamic data operations (modification, deletion, insertion) with integrity assurance. Security of the scheme (in accordance to the security definition given by authors) is proved and performance analysis results are present.

Mithun Paul and Ashutosh Saxena in their work focused on issues of being sure that cloud data storage provider actually deletes data upon user request, i.e., it is not

⁷G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, “Provable Data Possession at Untrusted Stores”, in *Proc. of CCS’07*. New York, NY, USA: ACM, 2007, pp. 598–609;

H. Shacham and B. Waters, “Compact Proofs of Retrievability”, in *Proc. of ASIACRYPT’08*. Melbourne, Australia: Springer-Verlag, 2008, pp. 90–107.

Figure 2.8: CryptDB onion encryption scheme⁸

stored anymore [55]. Authors propose a scheme for a client to get *proof of erasability* (POE), i.e., proof that the target data is completely destroyed or is irreversibly rendered useless.

The idea of the comprehensive data destruction algorithm is to modify systematically the most significant bits (MSB) of every data chunk, which, as authors claim, makes data destroyed beyond recoverability, and refutes any concerns on privacy and security.

A scenario with cloud data storage provider having several servers storing the same data is investigated superficially. In this case data owner who is destructing his data in the cloud should take into account a so-called by authors *exposure window*—time span between the moment when destructive query has been executed on the master server and the moment when it was replicated to the slave servers.

2.5 Existing Systems

Up to the current moment the only research describing an almost fully-operational encrypted relational database is the one done by R. Popa *et al.* describing the CryptDB [57–59]. Authors use a *modular* approach to construct CryptDB: to support each feature like keyword search, equality selection, range checks, joins, etc. a different secure primitive is used. They are all combined into a system by the means of a so-called “*onion*” *encryption*, i.e., a more rarely used secure primitive is wrapped into another, more often used one (see Figure 2.8). Another one is the work by Hacıgümüş *et al.* This work covers performing a full set of relational operations on encrypted data and became a basis for many future researches.

⁸Illustration is taken from the paper by Popa [58].

Figure 2.9: CryptDB architecture⁹

2.5.1 CryptDB: Structure, Capabilities, Workflow

As it was mentioned before, CryptDB is an *almost* fully operational encrypted relational database. As authors claim, they have collected a trace of 126 million SQL queries from MySQL servers in production, and as analysis showed, CryptDB is capable of executing operations over encrypted data for 99.5% of the 128 840 columns seen in trace. The solution shows a considerably good performance and low overhead, the throughput is reduced only by 14.5% for a web-forum application phpBB, and by 26% for queries from the database performance benchmarking tool TPC-C as compared to plaintext MySQL database. Adjusting existing applications to use CryptDB needs almost negligible amount of changes to the source code.

As it is reflected in the title of one of the papers, CryptDB is a *practical* secure database [59], which means that authors were balancing between providing full feature set, good performance, and high security level. CryptDB’s architecture consists of two interconnected parts: a database proxy and an unmodified DBMS (e.g., MySQL). CryptDB uses user-defined functions (UDFs) in order to be able to perform operations over encrypted data in the unmodified DBMS. Figure 2.9 illustrates the architecture of CryptDB. Rectangular and rounded boxes represent processes and data, respectively. Shaded boxes indicate components, which were added by authors of CryptDB. Dashed lines show the separation between components of the system—users’ computers, an application server, a proxy server running a CryptDB proxy program (this could actually be the same application server), and the DBMS server.

CryptDB considers only two kinds of threats, which are shown in the figure along with their influence areas as dotted lines. Threat 1 considers a curious database administrator (DBA) with complete access to the DBMS server, i.e., who can see all the data stored in the DBMS. CryptDB prevents him from learning anything from what he sees. Threat 2 considers an adversary, who gained a full access and complete control over both the software and hardware of the application, proxy and DBMS

servers. In this case CryptDB guarantees that the adversary is unable to obtain confidential information belonging to the users, *who are not logged in*.

CryptDB relies on several cryptographic schemes in order to be able to execute certain operations on data.

Deterministic encryption (DET). For deterministic encryption authors use block ciphers—Blowfish and AES for 64-bit and 128-bit values respectively. Smaller values are padded to 64 bit. Deterministic encryption is used to aid operations using equality checks like **1)** selects with equality predicates; **2)** equality joins; **3)** GROUP BY; **4)** COUNT; **5)** DISTINCT; etc.

Order-preserving encryption (OPE). For order-preserving encryption CryptDB uses the scheme proposed in [10] by Boldyreva *et al.*, which is the first provably secure symmetric order-preserving encryption. Encryption $\text{OPE}_K(\cdot)$ is called an order-preserving encryption if $(x < y) \implies (\text{OPE}_K(x) < \text{OPE}_K(y))$ for any secret key K . OPE is used for the server to be able to perform **1)** range queries, i.e., check if value is in the interval $[c_1, c_2]$; **2)** ORDER BY; **3)** MIN; **4)** MAX; **5)** SORT; etc.

Homomorphic encryption (HOM). Homomorphic encryption allows server to perform arithmetic operations on the encrypted data without decrypting it. For CryptDB the authors have chosen the Paillier probabilistic cryptosystem (see [54]), which allows to perform summation of encrypted values, i.e., $\text{HOM}_K(x) \times \text{HOM}_K(y) = \text{HOM}_K(x + y)$. Built-in functions like SUM or AVG, which rely on summation need to be rewritten as a UDF, which takes into account specific way of calculating a sum in an encrypted form.

Joins (JOIN and OPE-JOIN). A separate encryption scheme is used by authors to aid execution of equality- and range-joins. This is needed because deterministic and order-preserving encryptions use individual encryption keys for each column, so they can not be matched directly. The encryption scheme for equality joins—JOIN—is constructed as $\text{JOIN}(v) = \text{JOIN-ADJ}(v) \parallel \text{DET}(v)$, where $\text{DET}(\cdot)$ is a deterministic encryption, and $\text{JOIN-ADJ}(\cdot)$ is a new cryptographic primitive introduced by the authors, which allows for performing equality checks between any two different columns. As for order-preserving joins, CryptDB requires to know ahead of time the set of tables, which are to be involved into range joins.

Word search (SEARCH). SEARCH scheme is used to perform searches on en-

⁹Illustration is taken from the paper by Popa [58].

encrypted text and to support such SQL operations as LIKE. SEARCH is implemented using a protocol described by Song *et al.* [67] with some additions aimed at improving the security. DBMS is able to execute queries like `SELECT * FROM  $t$  WHERE  $t.m$  LIKE "% something %"` but is unable to retrieve results using LIKE operator to make a wild-card search: `SELECT * FROM  $t$  WHERE  $t.m$  LIKE "somet%"`.

2.5.2 Limitations of CryptDB

CryptDB is one of the first systems of that type and has many new and elegant solutions and notable improvements on existing techniques and approaches. At the same time CryptDB has many limitations, which require a closer investigation and improvement where possible.

Capabilities

In order to test CryptDB, Popa *et al.* used a trace of 126 million SQL queries collected from some real applications working with a production MySQL server. The percentage of fully supported actions and data types is about 99.5%. The remaining 0.5% of operations could be crucial for certain applications and they simply would not work without full support.

Range joins (i.e., joins, which involve order comparisons) are among the queries, which are not *fully* supported by CryptDB. It is still possible to perform range joins in CryptDB, but it requires that columns involved in such joins were declared by the DBA ahead of time. While this allows a user to use range joins it reduces the flexibility and takes away the ability to perform arbitrary queries.

Apart from that, while CryptDB allows to perform arithmetic operations on encrypted data in the DB, there are also certain computations CryptDB does not support. As the authors state, their system is not capable of evaluating comparison and arithmetic computation in the same query like `SELECT * FROM  $t$  WHERE  $age * 10 + 2 > salary$` . In order to do that the CryptDB system should be modified in a way, which enables the proxy to do some computations. After that to fulfill query like the one above proxy will first form a temporary column *tmp* filled with values returned by a query `SELECT  $age * 10 + 2$  FROM  $t$` , which then will be compared against values from column *salary*.

Moreover, CryptDB is unable to perform a multiplication of two encrypted values. Based on a description of how arithmetic operations are done with constant values (like `SELECT  $age + 7$  FROM  $t$`) given in the paper, it seems that CryptDB is also

unable even to multiply encrypted value and constant value, i.e., it is unable to perform any multiplication operations.

In addition, the approach authors used to make comparisons of encrypted data items possible (using order preserving encryption schemes) makes CryptDB unable to directly compare values from two arbitrary different columns. In theory, this could be done using the OPE-JOIN encryption, but this will only work on a predefined set of columns.

Another limitation relates to a multi-user scenario: when working in a multi-user context CryptDB is unable to perform server-side computations on values belonging to (and thus encrypted for) different users.

Stored procedures are semi-supported: it is possible to rewrite existing stored procedures, which were initially designed to work with plain-text in an encryption-aware manner. But still, these procedures could face architectural limitations like the ones described above.

Security Analysis

Research on CryptDB addresses two types of possible threats:

1. Curious database administrator;
2. Adversary gains full control of the application, proxy, and DBMS servers.

In this second threat scenario CryptDB does not provide any data safety guarantees for currently logged in users. In other words data safety is guaranteed only for the users who are currently logged off. It is worth noticing that secure databases are highly demanded by the commercial sector, which implies a certain usage model—almost all users are simultaneously logged in during office hours. This means that in fact, for the threat type 2 CryptDB almost does not guarantee anything in this wide-spread usage model.

Also, while still providing certain guarantees of data confidentiality, CryptDB does not ensure the integrity, freshness, or completeness of results in the scenario of adversary compromising application, proxy, and/or DBMS server, or of a malicious DBA—data could be deleted or modified and this could remain unnoticed.

Apart from that, CryptDB does not hide the overall table structure, number of rows, types of columns, and approximate data size in bytes. CryptDB also reveals some information on relationships between data items—what operations are performed with the data. On the other hand, CryptDB does conceal table names and order, which makes table structure analysis harder for an adversary, but we can not consider

this as a serious security improvement, since this is a typical example of “security through obscurity”.

The minimum granularity, which CryptDB is able to work with is column. This could lead to a situation when a weaker onion layer is revealed for the whole column even though only several rows were needed. Revealing a weaker layer of an onion in the CryptDB onion encryption layers structure (see Figure 2.8) always means exposing deterministic encryption, which gives an adversary a chance to perform statistical attacks and, which is considered by some researchers unacceptable for a production-ready secure DBMS.

Efficiency and convenience

CryptDB is created upon a concept that is called *a wrapper approach*—it is built on top of an existing DBMS (MySQL in this case).

Since CryptDB is by design supposed to be transparent for the user, this approach allows a customer to migrate existing applications from the plaintext database, which is currently in use to the CryptDB with few or no changes at all, which is especially important when the customer is working with closed-source applications that he has no ability to amend. Still, as far as one can conclude from the related publications, CryptDB requires some small changes to be made to the applications’ code to get it to work with the CryptDB, so it seems that closed-source applications should be initially designed to support CryptDB, which is a non-negligible limitation.

CryptDB is also hugely relying on a proxy which is supposed to be *fully* trusted. In certain situations this could be hardly achievable and thus could become a single point of failure and compromise the security of a whole system. There are solutions by other researchers, which also use the proxy approach but solve this problem [44].

Benchmarks conducted by authors show that throughput is reduced by 14.5% on real applications and by 26% on synthetic tests. These are very good results but it seems that these benchmarks were mostly concerned on OLTP and simple queries rather than on OLAP which is a highly demanded by commercial sector feature, and is likely to show a much bigger performance drop.

2.6 Summary

This chapter introduced a system of basic definitions describing a relational database and its parts. In the rest of the chapter the current state of the art in several areas that are related to the research objective have been discussed in details. First of all, a problem of the encrypted search was examined, it was highlighted what major types

and approaches are there, what issues usually arise, and how do researchers solve them. Another important issue is encryption keys management and related issues of authentication, authorization, and access management. It was highlighted and discussed what are the approaches proposed by various researchers. Additionally it has been discussed what are currently known solutions for the issue of auditability of encrypted data in a cloud database or data storage.

Second half of the chapter is dedicated to the thorough analysis of one of the most complete projects on creating an encrypted database—the CryptDB project, developed in MIT. This solution’s capabilities were discussed, as well as its approaches and limitations, and how do they correspond to the demands and expectations of potential users.

Chapter 3

System Overview

This chapter gives a general overview of the work presented in this thesis, which is elaborated in more details in the following chapters.

There is a multitude of possible ways of achieving the research objective that was stated in Section 1.2. However, along with this objective there was also an additional intention behind this research to create a practical solution; and this intention was in part a reason for the research approach to be formulated the way it was. Practicality may be measured in different ways and from different standpoints. The two main factors that were considered to determine practicality of our approaches were losses in performance, and feasibility of deployment of such solution in real production scenarios. Some of the foundational approaches in this project were chosen over the others with the intention of practicality in mind.

3.1 Approach Overview

There are two options in creating an encrypted DBMS, which are as follows:

1. Create a DBMS from scratch with internal designs crafted to cater to the needs of security and privacy;
2. Augment existing DBMSs (like MySQL, PostgreSQL, etc.) with external and internal modules to provide an encryption layer.

While developing an encrypted DBMS from ground up is a more flexible approach, which gives much more opportunities to embed complex algorithms and protocols in the core of the system, it requires an unjustified amount of additional labor to re-implement all the traditional components of a modern DBMS including, but not limited to, query scheduler, query parser, storage layer, indexing mechanisms, and many more. The result of choosing such an approach is bound to be no more than a

proof-of-concept and completely unusable in practice. Moreover, due to components providing data security being a fraction of the components that are required for a DBMS to operate, any empirical measures on such system would not provide accurate data of how the data security components perform in the overall system as these data would be contaminated by very probable performance issues of other components.

The “wrapper” approach has both benefits and limitations to it. Here is how both a researcher and a potential user of a secure DBMS could benefit from this approach:

- It allows the researcher to focus on providing security and not to implement all the other (often complex and hard to implement) database subsystems such as transaction processing unit, storage manager, indices, query compiler, etc.
- It makes it possible to use different underlying DBMSs (e.g., MySQL, PostgreSQL, MS SQL Server, etc.) with little or no changes to the system.

The wrapper approach has its limitations too:

- Adding encryption awareness layer to the DBMS without interfering with its source code may be significantly less efficient as it would have been if the DBMS source was rewritten in a manner to achieve same goals.
- It might appear to be more complicated to tweak or change inner mechanisms of the DBMS in an encryption-aware way (e.g., transaction processing or indexing).

Considering these observations and keeping in mind the practicality intention, the augmentation of existing systems (the wrapper approach) was chosen as the course of action for this research.

3.2 Model Design

3.2.1 Notion of “Data Security”

The notion of *data security* is extremely broad and covers a vast amount of specific issues. Let us take a moment to analyze the term and state what is understood by “data security” in the scope of this work.

“Security” of Data Security

Data security is often considered by both practitioners and researchers as a tri-fold notion consisting of following components: *confidentiality*, *integrity*, and *availability* [3].

Confidentiality This refers to ensuring that data may only be read by the authorized parties;

Integrity This notion is two-fold:

- Data can only be written by the authorized parties,
- Data is not corrupted;

Availability Data is not lost or destroyed, and could be retrieved in a timely manner.

As it could be seen, security includes measures against media degradation, accidents, natural disasters, etc.; it covers network connectivity issues and hardware or software failures; of course, a part of security is also actions against malicious activity.

“Data” of Data Security

Generally, data might be considered to be in one of the *three states of digital data* [7]. Namely, *data at rest*, *data in use*, and *data in motion*.

Data at rest Data is considered to be in this state when it was written to the media for the storage purposes.

Data in use In this state data is being transformed, aggregated, analyzed, and/or processed in any other form.

Data in motion Data is being transferred; usually, by the means of a computer network.

Similar to how it was with the notion of “security”, the notion of “data” appears to be very broad.

3.2.2 Threat Model

In the course of this work when we say “data security”, we, of course, mean a much narrower set of issues. But first and foremost, for the discussion not to be completely abstract it is required that a threat the data needs to be secured against is determined. This section establishes and discusses the *threat model*—a description of an adversary’s possible behavior.

As it was briefly mentioned in the introductory chapter, in the scenario of migrating the data processing infrastructure to the cloud, the cloud platform provider itself becomes an uncontrollable threat to the security of the data in all of the senses discussed above. However, from a common sense point of view, corrupting or stealing the data hardly would be an internal company policy in the case of any more or less established cloud platform provider as this has a high chance of surfacing and demolishing the company’s reputation. On the other hand, somebody in the IT staff of such provider might find it profitable to exploit his position. At the same time, such person or a group would prefer not to reveal their actions and thus are likely to

refrain from interfering with the system too much. Based on these observations and assumptions, the threat model in the scope of this work is formulated as follows:

A passive observer with full unrestricted access to the DBMS and the hardware it runs on aims to maximize his knowledge of the plaintext data that is being stored and processed in the system.

An obvious observation would be that among the three components of security this work focuses on the confidentiality. It is also easy to observe that there are proven effective techniques to provide confidentiality of the data at rest and data in motion under the chosen threat model; some examples would be: for the data at rest—full disk or file-level encryption, or Transparent Data Encryption (TDE) mechanisms that are included with many modern DBMSs or are available as standalone products; for the data in motion—SSL/TLS or other secure encrypted network protocols. However, *confidentiality of data in use* in the context of a relational cloud DBMS is a topic that leaves a lot of room for investigation and is a chosen direction for this research.

There are several important corollaries of the threat model that influence how the system is designed. First, as the DBMS can clearly never be supplied with the decryption keys, it is required that there was a part of the system with a higher level of trust than the DBMS. Second, as the adversary is modeled to be passive, the model may rely on all protocols and algorithms being executed as intended. And third, the adversary is allowed to execute data retrieval queries as they do not change the data and thus are assumed to be untraceable; that is, of course, given that an adversary is able to construct a meaningful data retrieval query, which might not necessarily be the case.

The term *untrusted cloud* that is being in use throughout this work refers to this threat model unless stated otherwise.

3.2.3 Model Topology

To summarize the previous discussion, here are the main assumptions about the environment of the system under consideration:

1. The DBMS is running in a completely untrusted environment but follows all protocols and instructions;
2. Database users may not possess vast computational resources at hand (say, a thin client);
3. The system design should follow the wrapper approach;
4. Existing applications should be able to switch to using the encrypted database with no or little changes.

Figure 3.1: Main system components

The last assumption is derived from the practicality intention and brings us to the requirement that the system is transparent for the authorized user; i.e., ideally, the user continues to issue plaintext queries the same way he did before but now they are not going directly to the DBMS but are routed through the system, which takes care of everything related to securing the data.

Based on these assumptions it is possible to draft a system that constitutes of four components (See also Figure 3.1):

User “User” is generally understood to be an entity that interacts with the data in the database. Most commonly it is an application rather than a person. The user is assumed to have limited computational resources meaning that he is generally unable to perform en-/decryption or other operations with data in high volumes; the user may be considered as running on a thin client or a mobile device.

Database Management System (DBMS) This is a regular relational DBMS (e.g., MySQL, Oracle, MS SQL Server, PostgreSQL, etc.) running in an untrusted cloud. The DBMS might have installed additional stored procedures, user defined functions, etc. in order to facilitate execution of the security protocols.

Proxy The proxy is one or several intermediate agents between the user and the DBMS. Instead of connecting to the DBMS directly, the user connects to one of the proxies. The proxy is responsible for relaying the semantics of the user’s plaintext queries to the now encrypted DBMS; decrypting and post-processing of the DBMS’s response and relaying it back to the user.

Key Management System (KMS) An entity that generates, stores, delivers, destroys, and renews encryption keys that are in use in the system.

The user is assumed to be authorized to access the database and thus is considered fully trusted. Although there might be additional layers of security such as access

control mechanisms for the user to get authorized, most of the protocols and algorithms designed in the course of this work require the user to possess a secret generated by the KMS in order to be able to produce meaningful queries and comprehend the server's responses.

It is easy to conclude from the description of the proxy that it has to possess certain parts of the secret from the KMS as well, and is thus required to be of a higher trust level than the DBMS. Generally, the proxy is assumed to be fully trusted; however, as it will be shown in the next chapters, certain algorithms do not require that.

The KMS is of course assumed to be fully trusted. Even though the KMS is the most dangerous part of the system as it is a single point of compromise, it is only required to be online from time to time, when it is needed to grant or revoke access rights to or from the user, or renew the keys. Other than that, security of the KMS is out of the scope of this research.

3.2.4 Terminology

This section introduces the non-standard terminology that is used in the following chapters and provides a short discussion of the suggested notions.

Definition 3.1 (Encryption). Encryption is a pair of functions $e(x, k_e)$ and $d(x, k_d)$ where $d(e(x, k_e), k_d) = x$. Parameters k_e and k_d are called encryption and decryption keys, respectively, and might or might not be equal. $\triangleleft$

Encryption is a general abstraction of a reversible cryptographic transformation. Case of $k_e = k_d$ resembles symmetric encryption schemes, case of $k_e \neq k_d$ resembles asymmetric encryption schemes.

Definition 3.2 (Hash). Hash is a function $h(x, k)$ such that $\nexists g : g(h(x, k), k) = x$ for $\forall x$. $\triangleleft$

Hash is a general abstraction of a non-reversible cryptographic transformation. Argument k is not necessarily significant as there are many hash functions that do not take a key (e.g., MD5); and many of them do (e.g., HMAC).

Definition 3.3 (Cryptographic transformation). Cryptographic transformation is a function $f(x, k)$, which is either an encryption (i.e., a counterpart decryption function exists) or a hash (decryption function does not exist), as per Definitions 3.1 and 3.2. $\triangleleft$

Definition 3.4 (Cryptoset). Let $f_1, \dots, f_n$ be cryptographic transformations with a set of respective keys $k_1, k_2, \dots$; for a plaintext domain V , a cryptoset of a plaintext

value $v \in V$ is a function $\psi(v) = \{f_1(v, k_1), \dots, f_n(v, k_n)\}$. $\triangleleft$

A cryptoset is one of the fundamental concepts in the proposed model. A cryptoset function transforms a plaintext value into a set of its cryptographic representations using several cryptographic transformations. The word “cryptoset” will be used to refer to both a function, and the resulting vector of encrypted or hashed values, depending on the context. A specific set of cryptographic transformations is decided based on what types of operations the database is required to be able to perform against its data.

Definition 3.5 (Decryptoset). Let ψ be a cryptoset $\psi : V \rightarrow \tilde{V}$. Let $\tilde{V}' = \{x \mid \exists a_i \in x : a_i \text{ is an encryption} \wedge \exists y \in \tilde{V} : x \subseteq y\}$ be a set of specific subsets of the set $\tilde{V}$. Then, decryptoset δ_ψ is a function $\delta_\psi : \tilde{V}' \rightarrow V$. Elements of the set $\tilde{V}'$ are called *subcryptosets*. $\triangleleft$

Decryptoset function is a reverse of the cryptoset function—it decrypts a cryptoset back into a plaintext value. Decryptoset function δ_ψ corresponding to a given cryptoset function ψ is also defined on a set of ψ -subcryptosets, a set of partial ψ -cryptosets, which contain at least one encryption (a reversible cryptographic transformation). Decryptoset function only exists for reversible cryptosets, i.e., containing at least one encryption.

As long as the model described in this work offers a *transparent* database encryption, the user interacts with the database as if it is a regular plaintext database. However, as it is shown in next chapters, the actual schema of the data in the database is different, so as to accommodate cryptosets and some additional data that is required to execute encrypted operations. The original data schema does exist only virtually and is thus called a virtual data schema¹.

Definition 3.6 (Virtual data schema). The original schema of the plaintext database that only exists as an abstraction for the user. $\triangleleft$

3.3 Summary

This chapter gives an overview of the proposed model of an encrypted database. Following chapters talk in details about the components of the model and how they interact; one of the purposes of this chapter is to give the reader an understanding, what components there are, and why, what are their general responsibilities, and what channels of interaction do they have. The chapter also provides definitions for notions that the next chapters are built upon, as well as a brief discussion of the

¹See Figure 4.1 on p. 55

defined notions.

First part of this chapter takes a deeper look at the notion of data security, discusses the threat model, and based on that establishes what exactly is understood by the words “secure database” in this work. After taking into account certain assumptions about the profile of a potential user of the secure database, the threat model, and other considerations, the topology of the system is defined and justified. And lastly, crucial for understanding the following chapters terminology is defined.

Chapter 4

Query Execution Model

If the course of this work was suggesting developing a privacy-preserving DBMS from ground up, the reasonable way would have been to develop encryption-aware counterparts of the basic relational operations, which then could have been combined into more complex queries.

In the course of work that has actually been chosen, we are bound to work on a higher level: develop encryption-aware equivalents for the more high-level operations and exploit the existing implementations of the relational algebra. Examples of such high level operations would be numeric and textual equality checks, numeric greater/less checks, numeric addition, numeric multiplication, etc.

The model of query execution is based on the notion of a cryptoset that was introduced in the previous chapter (see Definition 3.4), and two assumptions about the capabilities of the cryptographic primitives the cryptoset comprises of, which are discussed below.

There are two general types of operations over data values that could be met in a workflow of a DBMS: operations with the resulting value being in the same domain¹ as the operands (e.g., addition, multiplication, etc.: the operands are numbers, and the result is a number); and operations with the result being boolean (e.g., equality check: the operands are numbers or strings, and the result is a **true/false** boolean value).

Let O_v be a set of the operations that result in a same-domain value; let O_b be a set of the operations resulting in a boolean value. Both sets contain only the plaintext operations that the encrypted DBMS is required to be able to execute; their encryption-aware counterparts form sets $\tilde{O}_v$ and $\tilde{O}_b$, respectively. With that let us

¹See Definition 2.1

formulate the assumptions:

Assumption 1 (Domain-valued). For every plaintext operation $f \in O_v$, where f is a function $f(x_1, \dots, x_n), n \geq 1$ there exists a cryptoset $\psi(\cdot)$, a decryptoset function $\delta_\psi(\cdot)$, and a function $g_f \in \tilde{O}_v$ such that $\delta_\psi(g_f(\psi(x_1), \dots, \psi(x_n))) = f(x_1, \dots, x_n)$.

Assumption 2 (Boolean-valued). For every plaintext operation $f \in O_b$, where f is a function $f(x_1, \dots, x_n), n \geq 1$ there exists a cryptoset $\psi(\cdot)$, a decryptoset function $\delta_\psi(\cdot)$, and a function $g_f \in \tilde{O}_b$ such that $f(x_1, \dots, x_n) \implies \delta_\psi(g_f(\psi(x_1), \dots, \psi(x_n)))$.

Relaying the assumptions in an intuitive way, we assume that for any basic operation that we want to execute against encrypted data there exists a cryptosystem that possesses a property of homomorphism for that operation. It is easy to notice that Assumption 2 is weaker than Assumption 1: encrypted version of operation has to yield **true** if the plaintext version yields **true**, but if the plaintext one yields **false** the encrypted one may yield **true** or **false** at random. It is also easy to observe that both assumptions hold for such basic operations as addition, multiplication, greater/less/equality checks.

Assumption 2 is defined in a weaker form than Assumption 1 for a reason. The way Assumption 2 is defined, operations from O_b , when transformed into their encryption-aware equivalents, might return false positives. Even though this chapter is considering an abstract model, i.e., not concerning specific operations in O_v and O_b , for the vast majority of applications one of the most demanded and useful operations in O_b is equality check; a big fraction of other, more complex operations require the equality check, e.g., equi-joins, search, and many more. In certain cases, making it possible to do precise equality checks and other boolean-valued operations makes the system vulnerable to a plethora of statistical attacks. However, Assumption 2 does not *prohibit* precise boolean operations, so they could be implemented and used, if necessary.

4.1 Querying Capabilities

A typical query language such as SQL comprises of a data definition language (DDL), data manipulation language (DML), and transaction control language (TCL).

4.1.1 DDL

In the scope of this research model, certain simplifications are incorporated to make it feasible to describe it. One of such simplifications is that the DBMS is assumed to only host one database, and all the schemas are permanent, and are created during

the database setup.

Thus, it is only required to support the `CREATE TABLE` command of the DDL. To be more specific, here is what the model supports:

```
<create table query> ::=
    CREATE TABLE <table name> (
        <column definition>
        [, <column definition> , ...]
    )

<column definition> ::=
    <column name> <column domain>
```

These are bare essentials of the DDL that are required. Throughout this work the column domain is assumed to be unsigned integer. `CREATE TABLE` queries usually also define indexes on certain columns, however, using the DBMS's indexing mechanism heavily relies on how exactly encrypted search is implemented. This is considered in more details in Chapter 6.

4.1.2 DML

Data manipulation language is the biggest and most powerful component of the SQL. Moreover, it varies to a certain extent from DBMS to DBMS. In the scope of this work only an essential core of the DML is covered, which is mostly identical in all implementations of SQL, and is representing a majority of query structures in real world applications. In most cases, extending the model to incorporate larger subsets of SQL is a matter of technicality, and does not require major additional research.

The subset of the DML supported by the described model is a set $\mathcal{Q}_{\text{DML}}$ of all the queries that satisfy the following:

```
<query> ::= <insert query> | <update query>
          | <select query> | <delete query>

<insert query> ::=
    INSERT INTO <table name>
    VALUES (<const>, ...)[, (...), ...]

<update query> ::=
    UPDATE <table name>
    SET    <column name> = <expression>
    [    , <column name> = <expression>, ...]
    [<FROM clause>]
```

```

[<WHERE clause>]

<select query> ::=
    SELECT <expression>[, <expression>, ...]
    [<FROM clause>]
    [<WHERE clause>]

<delete query> ::=
    DELETE FROM <table name>
    [<WHERE clause>]

<WHERE clause> ::=
    WHERE <WHERE predicate>

<FROM clause> ::=
    FROM <table name>[, <table name>, ...]

<expression> ::= <constant>
                | <column name>
                | <operation>

```

<constant> is any value from the applicable domain. <WHERE predicate> is defined in sections below. Concepts of <table name> and <column name> are self-explanatory. <operation> is also considered in details in sections below.

4.1.3 TCL

The supported extent of the transaction control language under the scope of this research is as follows:

```

<start txn query> ::=
    BEGIN TRANSACTION

<commit txn query> ::=
    COMMIT

<cancel txn query> ::=
    ROLLBACK

```

Execution of TCL queries in an encrypted context is considered in Chapter 5.

4.2 Abstract Query Execution Model

The main focus of the research on the query execution model is on the DML component, as the queries of this family are the ones that actually bring value, whereas DDL and TCL queries are more of a service nature.

Figure 4.1: How the QEM affects the interaction with the DBMS

The general idea behind the query execution model is to transform plaintext queries that are issued against the *virtual* data schema into their encryption-aware equivalents that are intended to be run against the *actual* data schema that is stored in the encrypted database (see Figure 4.1). For every column in the virtual data schema, which only exists in the imagination of the user, the database stores several columns that contain cryptographic representations of a plaintext value—its corresponding cryptoset. The queries are transformed so that **1)** all references to columns in the virtual data schema are substituted by references to correct columns; **2)** all plaintext constants are encrypted; **3)** all expressions are changed to their encrypted equivalents as per Assumptions 1 and 2; **4)** the query is split into several interconnected queries, if necessary.

Let us recall the definitions from Section 2.1. Let $\mathcal{D}$ be a database $\mathcal{D} = \{r_1, \dots, r_n\}$; let $\mathcal{A} = \bigcup_i s_i$ be a set of all attributes (columns) of the database $\mathcal{D}$, where s_i is a relation scheme for relation $r_i \in \mathcal{D}$, $i \in [1, n]$. These definitions will be used throughout this and following chapters.

The rest of this chapter covers processing of queries of the DDL and DML components of SQL, and the TCL component will be considered in details in Chapter 5.

4.2.1 DDL

Essentially, transformation of a DDL query is equivalent to a transformation of a set of plaintext virtual data schemas into real encryption-aware versions of the data schemas, which are able to accommodate cryptosets instead of plaintext values.

Let $\mathcal{D}$ be the plaintext database; $\tilde{\mathcal{D}}$ be the encryption-aware database. The way the interaction of the user with the DBMS is modeled, the user is issuing queries


```

CREATE TABLE [Customers] (
  [Name] VARCHAR(128) NULL,
  [Age] INT NULL,
  [Purchases] INT NULL
)

CREATE TABLE [Customers] (
  [rowId] UNIQUEIDENTIFIER NOT NULL DEFAULT NEWID(),
  [Name.Text.Enc] VARBINARY(1024) NULL,
  [Name.Fingerprint] INT NULL,
  [Age.Paillier.Enc] VARBINARY(4096) NULL,
  [Age.ElGamal.Enc] VARBINARY(4096) NULL,
  [Age.Fingerprint] INT NULL,
  [Purchases.Paillier.Enc] VARBINARY(4096) NULL,
  [Purchases.ElGamal.Enc] VARBINARY(4096) NULL,
  [Purchases.Fingerprint] INT NULL,
  [Common.Paillier.N] VARBINARY(4096) NOT NULL,
  [Common.ElGamal.P] VARBINARY(4096) NOT NULL
)

```

Figure 4.2: Example of a CREATE TABLE DDL query transformation

against $\mathcal{D}$, which is a virtual data schema (see Definition 3.6), and they are on-the-fly converted into encryption-aware equivalents that are executed against the existing in reality encrypted database $\tilde{\mathcal{D}}$.

Assuming that $\mathcal{D}$ consists entirely of relations with no tuples in them, i.e., empty relations, what a transformation of DDL queries does is a conversion of such database $\mathcal{D}$ into its equivalent $\tilde{\mathcal{D}}$: $\mathfrak{M}(\mathcal{D}) = \tilde{\mathcal{D}}$.

Relations in the model are mapped one-to-one, so the transformation $\mathfrak{M}$ just changes the schema of each relation in a way that every original attribute is substituted by a set of attributes that can accommodate a cryptoset. Let $\psi(\cdot)$ be a cryptoset, and for any vector x let $x[i]$ denote i^{th} element of the vector. Let $\text{domain}(\cdot)$ be a function that returns a domain of a given element of a cryptoset. As the elements of cryptosets are encryptions and hashes, which usually are represented by very large integers, the most compact and handy way to store them in the database would be using such SQL types as `BINARY(...)` or `VARBINARY(...)`, although, some short hashes might fit in an integer type.

Let $\mathcal{C}$ be a set of constants of the original plaintext data domain. As assumed, database $\mathcal{D}$ is a set of empty relations $\{r_1, \dots, r_n\}$, and a schema of a relation r_i is $\text{schema}(r_i) = \{a_{i,1}, \dots, a_{i,m_i}\}$, where $a_{i,j} = \langle \text{Id}_{i,j}, \mathcal{C} \rangle$ is an attribute. Transformation operation $\mathfrak{M}$ transforms every attribute $a_{i,j}$ into a series $\tilde{a}_{i,j,1}, \dots, \tilde{a}_{i,j,k}$, where $k = |\psi(\cdot)|$. The model only requires that Id's are transformed in a unique and reversible way, without specifying how exactly they are transformed; this is left to the implementation. For any applicable integer indices p, q , the domain of $\tilde{a}_{p,q,t}$ is set as $\text{domain}(\psi(\cdot)[t])$.

The transformation $\mathfrak{M} : \mathcal{Q}_{\text{DDL}} \rightarrow \tilde{\mathcal{Q}}_{\text{DDL}}$ is defined.

Figure 4.2 demonstrates an example of a DDL query transformation performed by the implementation of the model that is considered in details in Chapters 6 and 7.

Additional, implementation-specific attributes that are not relevant to this illustration are greyed out in the figure. It is easy to notice that the textual attribute `[Name]` has a different cryptoset than the numeric attributes such as `[Age]`: it does not have Paillier and ElGamal encryptions but instead has a Rijndael² encryption. The cryptoset element marked as `Fingerprint` is a special hash that facilitates search, which is short enough to fit in an integer data type, whereas other cryptographic transformations produce longer ciphertexts, which are stored as byte arrays in a `VARBINARY` type.

4.2.2 DML

All data manipulation queries from the subset of DML defined above fall into one of the following two groups: single-pass queries and multiple-pass queries.

Definition 4.1 (Single-pass query). A query that upon transformation into its encryption-aware equivalent becomes a single query as opposed to becoming a series of queries. $\triangleleft$

Definition 4.2 (Multiple-pass query). A query that is not a single-pass query. $\triangleleft$

As it is shown in sections to follow, certain parts of the query, when transformed into their encryption-aware counterparts, are relying on operations that can not be executed by the DBMS alone, and require cooperation of the proxy (such operations as ξ, Δ, Ω , which are introduced below). Queries that do not have such operations are single-pass queries; queries that do have them are multiple-pass queries.

As stated earlier, we distinguish three classes of expressions:

1. A plaintext constant, referred to as a *constant expression*;
2. A reference to another column, referred to as a *column reference expression*;
3. An operation, referred to as an *operation expression*.

An operation expression is an n -ary function $f(e_1, \dots, e_n)$, where $f \in O_v \cup O_b$, and $e_i, i \in [1, n]$ is an expression. Any expression could or could not be computed by the DBMS in a single pass; based on that, expressions are called single- and multiple-pass expressions accordingly. If at least one expression in the query is a multiple-pass expression then the query is a multiple-pass query. However, the reverse is not true. It is easy to notice that constant and column reference expressions are always single-pass expressions.

A set of all expressions $\mathcal{E}$ is defined through this recursive definition:

²Rijndael cryptosystem is better known as AES, however they are not exactly same. While the encryption and decryption processes are identical, AES has constraints on key and block length: it has 128-bit blocks and 128-, 192-, or 256-bit keys.

$$\mathcal{E} \equiv \mathcal{C} \cup \mathcal{A} \cup \bigcup_i \{O_v^i \times \underbrace{\mathcal{E} \times \dots \times \mathcal{E}}_i \mid O_v^i \subseteq O_v \text{ is a set of } i\text{-ary operations}\},$$

and $O_v = \bigcup_i O_v^i$.

WHERE predicate is generally a logical expression. As it is well known, any logical proposition can be expressed in a Conjunctive Normal Form (CNF). When represented as a tree, a CNF of a WHERE predicate has 4 levels (see Figure 4.3, left):

- Level 1:** root, conjunction;
- Level 2:** disjunctions;
- Level 3:** negations;
- Level 4:** leaves, actual operations.

Since any WHERE predicate has an equivalent CNF, it is assumed without a loss of generality that a predicate is always in a form of CNF.

If there are no negations in the predicate, the leaves are still referred to as positioned on level 4. Even in a degenerate case of a predicate consisting of just one operation, we still consider the CNF tree as having conjunction on L1 and disjunction on L2 with just one operand each. Every leaf is a boolean-valued expression from the set $\mathcal{B}$ defined as:

$$\mathcal{B} \equiv \bigcup_i \{O_b^i \times \underbrace{\mathcal{E} \times \dots \times \mathcal{E}}_i \mid O_b^i \subseteq O_b \text{ is a set of } i\text{-ary operations}\},$$

and $O_b = \bigcup_i O_b^i$.

A set of all possible predicates is denoted $\mathcal{W}$.

The set $\mathcal{Q}_{\text{DML}}$ comprises four subsets $\mathcal{Q}_{\text{DML}} \equiv \mathcal{Q}_{\text{INSERT}} \cup \mathcal{Q}_{\text{UPDATE}} \cup \mathcal{Q}_{\text{DELETE}} \cup \mathcal{Q}_{\text{SELECT}}$. For any given database $\mathcal{D}$ this sets are defined as follows:

$$\begin{aligned} \mathcal{Q}_{\text{INSERT}} &\equiv \mathcal{D} \times \bigcup_i \underbrace{\mathcal{C} \times \dots \times \mathcal{C}}_i \\ \mathcal{Q}_{\text{UPDATE}} &\equiv \mathcal{D} \times \bigcup_i \underbrace{\mathcal{D} \times \dots \times \mathcal{D}}_i \times \left(\bigcup_j \left(\underbrace{\mathcal{A} \times \dots \times \mathcal{A}}_j \right) \times \left(\underbrace{\mathcal{E} \times \dots \times \mathcal{E}}_j \right) \right) \times \mathcal{W} \\ \mathcal{Q}_{\text{DELETE}} &\equiv \mathcal{D} \times \mathcal{W} \\ \mathcal{Q}_{\text{SELECT}} &\equiv \bigcup_i \underbrace{\mathcal{D} \times \dots \times \mathcal{D}}_i \times \bigcup_j \underbrace{\mathcal{E} \times \dots \times \mathcal{E}}_j \times \mathcal{W} \end{aligned}$$

Let $\mathfrak{D}_{\mathcal{D}}$ be a set of all databases with same set of relation schemas as the database $\mathcal{D}$: $\mathfrak{D}_{\mathcal{D}} = \{\mathcal{D}_0 \mid \mathcal{D}_0 \text{ is a database, s.t. } \forall r_0 \in \mathcal{D}_0 \exists r \in \mathcal{D} : \text{schema}(r_0) = \text{schema}(r); \forall r \in \mathcal{D} \exists r_0 \in \mathcal{D}_0 : \text{schema}(r) = \text{schema}(r_0)\}$. Any query $q \in \mathcal{Q}_{\text{INSERT}} \cup \mathcal{Q}_{\text{UPDATE}} \cup \mathcal{Q}_{\text{DELETE}}$ is a mapping $q : \mathfrak{D}_{\mathcal{D}} \rightarrow \mathfrak{D}_{\mathcal{D}}$; any query $q \in \mathcal{Q}_{\text{SELECT}}$ is a mapping $q : \mathfrak{D}_{\mathcal{D}} \rightarrow R$,

where R is a set of all relations with schemas from $\mathfrak{P}(\times_i \text{schema}(r_i))$, $r_i \in \mathcal{D}$, where $\mathfrak{P}(\cdot)$ is a powerset operator.

Let $\tilde{\mathcal{D}}$ be a database obtained from $\mathcal{D}$ using a transformation described in the DDL transformation section. Let $\tilde{\mathcal{C}}$ be a set of cryptosets corresponding to constants from $\mathcal{C}$; a set of encryption-aware equivalents of plaintext expressions $\tilde{\mathcal{E}}$ is defined as:

$$\tilde{\mathcal{E}} \equiv \tilde{\mathcal{C}} \cup \tilde{\mathcal{A}} \cup \bigcup_i \{ \tilde{O}_v^i \times \underbrace{\tilde{\mathcal{E}} \times \dots \times \tilde{\mathcal{E}}}_i \mid \tilde{O}_v^i \subseteq \tilde{O}_v \text{ is a set of } i\text{-ary operations} \},$$

and $\tilde{O}_v = \bigcup_i \tilde{O}_v^i$.

And similarly for boolean-valued expressions, a set $\tilde{\mathcal{B}}$ is defined as:

$$\tilde{\mathcal{B}} \equiv \bigcup_i \{ \tilde{O}_b^i \times \underbrace{\tilde{\mathcal{E}} \times \dots \times \tilde{\mathcal{E}}}_i \mid \tilde{O}_b^i \subseteq \tilde{O}_b \text{ is a set of } i\text{-ary operations} \},$$

and $\tilde{O}_b = \bigcup_i \tilde{O}_b^i$.

A set $\tilde{\mathcal{W}}$ is defined in the same way as the set $\mathcal{W}$ but with leaves from the set $\tilde{\mathcal{B}}$ instead of $\mathcal{B}$.

Encryption-aware equivalents of DML queries are:

$$\begin{aligned} \tilde{\mathcal{Q}}_{\text{INSERT}} &\equiv \tilde{\mathcal{D}} \times \bigcup_i \underbrace{\tilde{\mathcal{C}} \times \dots \times \tilde{\mathcal{C}}}_i \\ \tilde{\mathcal{Q}}_{\text{UPDATE}} &\equiv \tilde{\mathcal{D}} \times \bigcup_i \underbrace{\tilde{\mathcal{D}} \times \dots \times \tilde{\mathcal{D}}}_i \times \left(\bigcup_j \left(\underbrace{\tilde{\mathcal{A}} \times \dots \times \tilde{\mathcal{A}}}_j \right) \times \left(\underbrace{\tilde{\mathcal{E}} \times \dots \times \tilde{\mathcal{E}}}_j \right) \right) \times \tilde{\mathcal{W}} \\ \tilde{\mathcal{Q}}_{\text{DELETE}} &\equiv \tilde{\mathcal{D}} \times \tilde{\mathcal{W}} \\ \tilde{\mathcal{Q}}_{\text{SELECT}} &\equiv \bigcup_i \underbrace{\tilde{\mathcal{D}} \times \dots \times \tilde{\mathcal{D}}}_i \times \bigcup_j \underbrace{\tilde{\mathcal{E}} \times \dots \times \tilde{\mathcal{E}}}_j \times \tilde{\mathcal{W}} \\ \tilde{\mathcal{Q}}_{\text{DML}} &\equiv \tilde{\mathcal{Q}}_{\text{INSERT}} \cup \tilde{\mathcal{Q}}_{\text{UPDATE}} \cup \tilde{\mathcal{Q}}_{\text{DELETE}} \cup \tilde{\mathcal{Q}}_{\text{SELECT}} \end{aligned}$$

Let us introduce an operator $\mathring{\mathfrak{P}}$. It is similar to a powerset operator, but produces a set of all *ordered* subsets of the argument set. If A is a set of functions, then $\mathring{\mathfrak{P}}(A)$ produces a set of all possible superpositions of these functions.

This section considers a mapping $\mathfrak{M} : \mathcal{Q}_{\text{DML}} \rightarrow \mathring{\mathfrak{P}}(\tilde{\mathcal{Q}}_{\text{DML}})$ such that $\forall q_1, \dots, q_n \in \mathcal{Q}_{\text{INSERT}} \cup \mathcal{Q}_{\text{UPDATE}} \cup \mathcal{Q}_{\text{DELETE}}, \forall q_S \in \mathcal{Q}_{\text{SELECT}} : \mathfrak{F}(\tilde{q}_S((\tilde{q}_1 \circ \dots \circ \tilde{q}_n)(\tilde{\mathcal{D}})), w) = q_S((q_1 \circ \dots \circ q_n)(\mathcal{D}))$, where $\tilde{q}_* = \mathfrak{M}(q_*)$, $w \in q_S$ is a predicate, $\mathfrak{F}$ is a decryption and filtering function, which is defined later in this chapter.

Expressions

As per Assumption 1 and Assumption 2, there is a direct one-to-one projection from sets O_b and O_v into sets $\tilde{O}_b$ and $\tilde{O}_v$, respectively. This section builds on top of that fact and considers mappings $\mathfrak{M} : \mathcal{E} \rightarrow \tilde{\mathcal{E}}$ and $\mathfrak{M} : \mathcal{B} \rightarrow \tilde{\mathcal{B}}$.

An expression $e \in \mathcal{E}$ is either a constant ($e \in \mathcal{C}$), an attribute ($e \in \mathcal{A}$), or an operation with its operands being expressions as well ($e \in O_v \times \mathcal{E} \times \dots \times \mathcal{E}$). For a given database $\mathcal{D}$, an expression $e \in \mathcal{E}$ is a function $e : R \rightarrow \mathcal{C}$, where $R = \bigcup_i r_i, r_i \in \mathcal{D}$. Similarly, for a given encrypted database $\tilde{\mathcal{D}}$, an expression $e \in \tilde{\mathcal{E}}$ is a function $e : \tilde{R} \rightarrow S_{\tilde{\mathcal{C}}}$, where $\tilde{R} = \bigcup_i r_i, r_i \in \tilde{\mathcal{D}}$; $S_{\tilde{\mathcal{C}}}$ is a set of subcryptosets (see Definition 3.5) of cryptosets from $\tilde{\mathcal{C}}$.

Different elements of a cryptoset are required by operations from $\tilde{O}_v$ for them to be executed. These operations produce subcryptosets. If the original plaintext expression contained nested operations, which were transformed into nested encryption-aware operations, a resulting subcryptoset of an inner operation is generally assumed to lack elements required by an outer operation³. Thus, a function $\xi : S_{\tilde{\mathcal{C}}} \rightarrow \tilde{\mathcal{C}}$ is introduced, which restores a subcryptoset into a full cryptoset. To maintain strictness, let us add ξ as an unary operation into the set $\tilde{O}_v$.

In the case of $e \in \mathcal{C}$, a transformation of e would be $\mathfrak{M}(e) \equiv c'$, where $c' \subseteq \psi(e)$ is a subcryptoset. Elements of the subcryptoset are decided depending on where expression e was encountered in a query: if it was an operand for an operation, subcryptoset should contain at least the elements required by the said operation. If the result of an expression was supposed to be returned as a result of a query, then any valid subcryptoset is acceptable since it contains at least one reversible encryption by definition, and thus might be decrypted back into the domain $\mathcal{C}$.

As it was described in the DDL transformation, an attribute $a \in \mathcal{A}$ is mapped into multiple attributes $a_1, \dots \in \tilde{\mathcal{A}}$ in order to accommodate a cryptoset, which is a vector. In the case of $e \in \mathcal{A}$, similarly to previous case, it is transformed into a subset of the attributes $a_1, \dots \in \tilde{\mathcal{A}}$, such that they contain subcryptosets appropriate to where the expression e was encountered.

In the case of $e \in O_v \times \mathcal{E} \times \dots \times \mathcal{E}$, expression $e = \{o, e_1, \dots, e_n\}$, where $o \in O_v, e_* \in \mathcal{E}$. As per Assumption 1, o transforms into a respective $\tilde{o} \in \tilde{O}_v$. As for the operands, generally, either $e_* \in \mathcal{C} \cup \mathcal{A}$ or e_* is an operation as well. Let us notice that any finite expression that has operations as its operands on the inner-most

³However, in specific cases this is not true, it is safe to assume so. Implementations might optimize cases there this assumption does not hold.

Figure 4.3: Predicate CNF. Multiple-pass leaves are gray; single-pass leaves are white.

levels has operations with all operands from $\mathcal{C} \cup \mathcal{A}$. With that in mind, an inductive (recursive) transformation of operands of e could be applied:

- If $e_i \in \mathcal{C} \cup \mathcal{A}$, e_i is transformed as described above, when cases $e \in \mathcal{C}$ and $e \in \mathcal{A}$ were considered.
- If e_i is an operation with operands $e_i = \{o_0, t_1, \dots\}$, this recursive transformation is applied to it, and then its result is “wrapped” into an operation $\xi \in \tilde{O}_v$ in order to accommodate it for the operation $\tilde{o}$: $\mathfrak{M}(e_i) \equiv \{\xi, \{\tilde{o}_0, \mathfrak{M}(t_1), \dots\}\}$.

This concludes consideration of transformation $\mathcal{E} \rightarrow \tilde{\mathcal{E}}$.

An expression $e \in \mathcal{B}$ adds a layer on top of the expressions that were just considered. However, the fact that operations from O_b result in a boolean value does not change much. For a given database $\mathcal{D}$, an expression $e \in \mathcal{B}$ is a function $e : R \rightarrow \{0, 1\}$, where $R = \bigcup_i r_i$, $r_i \in \mathcal{D}$. Similarly, for a given encrypted database $\tilde{\mathcal{D}}$, an expression $e \in \tilde{\mathcal{B}}$ is a function $e : \tilde{R} \rightarrow \{0, 1\}$, where $\tilde{R} = \bigcup_i r_i$, $r_i \in \tilde{\mathcal{D}}$.

Expression $e \in \mathcal{B}$ is transformed in the same exact way as the third case of transformation of expression $e \in \mathcal{E}$.

Mappings $\mathfrak{M} : \mathcal{E} \rightarrow \tilde{\mathcal{E}}$ and $\mathfrak{M} : \mathcal{B} \rightarrow \tilde{\mathcal{B}}$ are completely defined.

WHERE Predicates

As established earlier, a **WHERE** predicate of a query, either plaintext or encryption-aware, is a logical expression in a conjunctive normal form. A plaintext **WHERE** CNF could be seen as a tree-like structure (see Figure 4.3, left) with expressions from $\mathcal{B}$ as leaves. This section considers a mapping $\mathfrak{M} : \mathcal{W} \rightarrow \tilde{\mathcal{W}}$.

First step of transforming a **WHERE** predicate is to map its leaves from $\mathcal{B}$ to $\tilde{\mathcal{B}}$ as described in the previous section. However, if at least one leaf requires an execution of operation $\xi \in \tilde{O}_v$, the DBMS is unable to compute a value of the predicate for a specific record and thus is unable to do the filtering at all. However, at least partial filtering could potentially significantly reduce the amount of computations overall; the more complex computations are needed to be done with the tuples that satisfy

the predicate, and the more tuples there is, the bigger is the impact of filtering out as many tuples as possible. The leaves that use operation ξ are marked as multiple-pass leaves (see Figure 4.3).

Let us consider a following transformation of the CNF tree. For every disjunction d (L2) every path is traversed down from d to the leaf l ; in case the leaf l is marked as multiple-pass, this path $d-l$ (L3 and L4) is completely removed from the tree. Childless disjunctions are then removed from the tree as a clean-up, if any. As a result a different predicate is obtained that is computable in a single pass (see Figure 4.3, right).

The following theorem states that the transformation of a multiple-pass CNF tree into a single-pass CNF tree that was described above does not induce false negatives during the filtering, i.e., it does not filter out records that should be selected/updated/deleted. It could, however, induce additional false positives—this issue is considered below.

Theorem 1. Let T be a set of tuples that satisfy the original non-empty predicate $w \in \widetilde{\mathcal{W}} \ (\forall t \in T : w(t) = 1)$. Let T' be a set of tuples that satisfy the transformed predicate $w' \in \widetilde{\mathcal{W}} \ (\forall t \in T' : w'(t) = 1)$. Then, $T \subseteq T'$. $\triangleleft$

Proof. The transformation is done in two steps: **1)** removing multiple-pass leaf nodes; **2)** removing childless disjunctions. If we prove that no tuple $t_0 \in T$ is eliminated in any of these two steps, the theorem is proven.

1) Removing leaves. First, let us note that to prove this part all that is needed is to show that for any given tuple, such transformation can never change the boolean result of the predicate from 1 to 0 (changing from 0 to 1 is acceptable: we will end up with more tuples than necessary but it will affect performance, not correctness). Second, let us note that removing an operand from a disjunction is equivalent to replacing the said operand with 1.

It is reasonable to only consider those disjunctions that have children removed, as others have an unchanged resulting value. If at least one operand of a disjunction is equal to 1, then the result of the disjunction is 1. Hence, if the original disjunction results in 0, after replacing one of the operands with 1, it will result in 1; if the original result is 1, the transformed result is 1. We have shown that the discussed transformation may only result in certain nodes on L2 changing their compute value from 0 to 1.

Let us consider how it will affect the root conjunction. If at least one of the

unaffected by the transformation disjunctions results in 0, then the conjunction will result in 0 regardless of whether any of the disjunctions may have changed their result to 1, i.e., the predicate result will *not* change with the transformation. In case all the unaffected disjunctions result in 1, there are three options:

1. Some of the affected disjunctions resulted in 0 before the transformation, and *not* all of them changed to 1; the predicate does not change the result and stays 0;
2. Some of the affected disjunctions resulted in 0 before the transformation, and *all* of them changed to 1; the predicate changes the result from 0 to 1;
3. All of the affected disjunctions initially resulted in 1, and stay 1 after the transformation; the predicate does not change the result and stays 1.

We have shown that for any given tuple the discussed transformation can probably change the predicate result from 0 to 1 but never from 1 to 0.

2) Removing childless nodes. A childless disjunction node is in fact a disjunction operation with no operands, which results in a 1. It is easy to prove that adding or removing a 1-valued operand to a conjunction does not change the result. Thus, removing a childless disjunction from the tree is resulting in an equivalent logical proposition, which obviously does not change the outcome for any given tuple.

The theorem is proven. □

Let us point out that the theorem only considers non-empty predicates, i.e., the ones that have at least one leaf. An empty predicate is transformed into an empty predicate.

INSERT Queries

A query $q \in \mathcal{Q}_{\text{INSERT}}$ is $q = \{r, c_1, \dots, c_n\}$, where $r \in \mathcal{D}$ is a relation, and $c_1, \dots, c_n \in \mathcal{C}$ are constants. This section considers a mapping $\mathfrak{M} : \mathcal{Q}_{\text{INSERT}} \rightarrow \mathring{\mathfrak{P}}(\tilde{\mathcal{Q}}_{\text{DML}})$. The result of a plaintext query $\mathcal{D}' = q(\mathcal{D})$ is a database with the relation r changed to the same relation having a new tuple $(c_1, \dots)$: $r' = r \cup (c_1, \dots)$. So, the result of a plaintext INSERT query is defined as:

$$q(\mathcal{D}) \equiv (\mathcal{D} \setminus \{r\}) \cup \{r \cup (c_1, \dots)\},$$

Recalling that, as stated in the DDL section, upon transformation the relations of the database are mapped in a one-to-one fashion, an encryption-aware INSERT


```

INSERT INTO [Customers] (
  [Customers].[Name.Paillier.Enc], [Customers].[Name.ElGamal.Enc],
  [Customers].[Name.Fingerprint],
  [Customers].[Age.Paillier.Enc], [Customers].[Age.ElGamal.Enc],
  [Customers].[Age.Fingerprint],
  [Customers].[Purchases.Text.Enc],
  [Customers].[Purchases.Fingerprint],
  [Common.Paillier.N], [Common.ElGamal.P]
)
VALUES (
  0x2BD9ACAC...943B4E6F, 0x8728B180...3649234E,
  8697,
  0x05674992...A3D24742, 0xE0703A92...36BE5EC5,
  16320,
  0x577A34D0...6FB5DA86,
  383,
  0x8519AC7A...88F0FB2B, 0x0FA16067...5D29253E
), (
  0x3279A8B0...75D6C49E, 0xBE50786D...EF193BFA,
  4020,
  0x2E8C898A...98968BE9, 0x74380392...C018B0FA,
  1018,
  0x38573BD3...B06AE9C8,
  40707,
  0x8519AC7A...88F0FB2B, 0x234EC522...433576CF
)

```

`INSERT INTO [Customers] (`
`[Customers].[Name],`
`[Customers].[Age],`
`[Customers].[Purchases]`
`)`
`VALUES (`
`'Kiera Nader', 24, 63`
`), (`
`'June Kuphal', 32, 86`
`)`

Figure 4.4: Example of an INSERT INTO DML query transformation

query $\tilde{q} \in \tilde{\mathcal{Q}}_{\text{INSERT}}$ obtained as $\tilde{q} = \mathfrak{M}(q)$ is defined as follows:

$$\tilde{q} = \mathfrak{M}(q) \equiv \{\mathfrak{M}(r), \psi(c_1), \dots, \psi(c_n)\},$$

$$\tilde{q}(\tilde{\mathcal{D}}) \equiv (\tilde{\mathcal{D}} \setminus \{\tilde{r}\}) \cup \{\tilde{r} \cup (\psi(c_1), \dots, \psi(c_n))\},$$

where $\tilde{\mathcal{D}} = \mathfrak{M}(\mathcal{D})$, $\tilde{r} = \mathfrak{M}(r)$.

The transformation $\mathfrak{M} : \mathcal{Q}_{\text{INSERT}} \rightarrow \mathring{\mathfrak{P}}(\tilde{\mathcal{Q}}_{\text{DML}})$ is completely defined. It is worth noting that an INSERT query is always transformed into a single INSERT query.

Figure 4.4 demonstrates transformation of a sample INSERT query performed by the implementation of the model. Implementation-specific attributes that are not relevant for this illustration were greyed out.

SELECT Queries

This section considers a mapping $\mathfrak{M} : \mathcal{Q}_{\text{SELECT}} \rightarrow \mathring{\mathfrak{P}}(\tilde{\mathcal{Q}}_{\text{DML}})$. Let $q \in \mathcal{Q}_{\text{SELECT}}$ be a SELECT query, $q = \{d_1, \dots, d_n, e_1, \dots, e_m, w\}$, where $d_* \in \mathcal{D}$ are relations, $e_* \in \mathcal{E}$ are expressions, $w \in \mathcal{W}$ is a WHERE predicate. We will show how to construct the result of transformation $\mathfrak{M}(q)$.

As mentioned in the discussion of DDL transformation, relations are mapped one-to-one, so transformation of the d_* part of the query q into $\tilde{d}_*$ is straightforward. Expressions e_* are transformed into $\tilde{e}_*$ as described earlier. The WHERE predicate w is transformed into $\tilde{w}$ according to what was described earlier.

Algorithm 1 Extraction of single-pass expressions for intermediate computations

```

function ISSINGLEPASS( $e$ )  $\triangleright e \in \tilde{\mathcal{E}}$ 
  if OPERATION( $e$ ) is  $\xi$  then  $\triangleright$  extracts an operation
    return 0  $\triangleright$  not a single-pass expression
   $s \leftarrow 1$ 
  for each  $o \in \text{OPERANDS}(e)$  do  $\triangleright$  extracts a list of expressions-operands
    if  $o \notin \tilde{\mathcal{C}} \cup \tilde{\mathcal{A}}$  then  $\triangleright$  expr. from  $\tilde{\mathcal{C}} \cup \tilde{\mathcal{A}}$  is always single-pass
       $s \leftarrow s \wedge \text{ISSINGLEPASS}(o)$ 
  return  $s$ 

function GETSINGLEPASSEXPRS( $e$ )  $\triangleright e \in \tilde{\mathcal{E}}$ 
  if ISSINGLEPASS( $e$ ) then
    return  $\{e\}$ 
   $r \leftarrow \{\emptyset\}$   $\triangleright$  An empty set
  for each  $o \in \text{OPERANDS}(e)$  do
    if ISSINGLEPASS( $o$ ) then
       $r \leftarrow r \cup \{o\}$ 
    else
       $r \leftarrow r \cup \text{GETSINGLEPASSEXPRS}(o)$ 
  return  $r$ 

function GETALLSINGLEPASSEXPRS( $E$ )  $\triangleright E \equiv \{\tilde{e}_1, \dots, \tilde{e}_m\}$ 
   $r \leftarrow \{\emptyset\}$ 
  for each  $e \in E$  do
     $r \leftarrow r \cup \text{GETSINGLEPASSEXPRS}(e)$ 
  return  $r$ 

```

Now let us consider expressions $\tilde{e}_*$. Each expression $\tilde{e}_i$ potentially contains operands that require execution of operation ξ —restoring a subcryptoset into a full cryptoset. Such operations as ξ and operations $\Delta, \Omega, \mathfrak{F}$ that are defined below are executed by the proxy, due to the DBMS having no access to the encryption/decryption keys, our inability to transfer decrypted tuples or plaintext predicates to the untrusted DBMS, and our assumption of the user being computationally weak. This involves transfer of required arguments from the DBMS to the proxy, and transfer of results back to the DBMS (if required for sequential computations). Thus, an expression that relies on the operation ξ makes the query a multiple-pass query.

At this point, if q appeared to be a single-pass query, the query $\tilde{q} \equiv \{\tilde{d}_1, \dots, \tilde{d}_n, \tilde{e}_1, \dots, \tilde{e}_m, \tilde{w}\}$ is the corresponding query from $\tilde{\mathcal{Q}}_{\text{SELECT}}$.

If it is a multiple-pass **SELECT** query, we take the structure $\{\tilde{d}_1, \dots, \tilde{d}_n, \tilde{e}_1, \dots, \tilde{e}_m, \tilde{w}\}$ and extract a set of expressions $E = \{\tilde{e}_1, \dots, \tilde{e}_m\}$ from it. Now let us form a set $E' = \text{GETALLSINGLEPASSEXPRS}(E)$ (see Algorithm 1). Let $\tilde{q}_1 = \{\tilde{d}_1, \dots, \tilde{d}_n, E', \tilde{w}\}$ be a first query of the transformation of the original multiple-pass query q .

It is easy to notice that the relation that such a query produces contains tuples with all the values being subcryptosets instead of belonging to $\mathcal{C}$. Second, according to Theorem 1 and Assumption 2, even after decryption the relation possibly contains tuples that do not satisfy the original predicate w . Let us define a filtering function $\mathfrak{F}$.

Let ψ and δ_ψ be the cryptoset and decryptoset functions. Let us define a function $\Delta(t)$ that takes as an argument a tuple $t = \{c_1, c_2, \dots\}$, where $c_i \in S_{\tilde{\mathcal{C}}}$, and produces a tuple $t' = \{c'_1, c'_2, \dots\}$, where $c'_i \in \mathcal{C}$ and $c'_i \equiv \delta_\psi(\xi(c_i))$, or, in other words, decrypts a tuple of subcryptosets. Let $\Omega(t, w)$ be an operation that takes a tuple t consisting of elements from $\mathcal{C}$ and a plaintext predicate $w \in \mathcal{W}$, and returns the tuple only if it satisfies the predicate w :

$$\Omega(t, w) \equiv \begin{cases} t & \text{if } w(t) = 1, \\ \emptyset & \text{if } w(t) = 0. \end{cases}$$

The filtering function $\mathfrak{F}$ is defined as $\mathfrak{F}(\tilde{q}(\tilde{\mathcal{D}}), w) \equiv \{\Omega(\Delta(t), w) \mid \forall t \in \tilde{q}(\tilde{\mathcal{D}})\}$, where $w \in \mathcal{W}$ is a plaintext predicate. $\mathfrak{F}$ results in a plaintext unencrypted relation.

Let $R = \mathfrak{F}(\tilde{q}_1(\tilde{D}))$ be the decrypted result of $\tilde{q}_1$. We now form a DDL query $\tilde{q}_d \in \tilde{\mathcal{Q}}_{\text{DDL}}$ that creates a temporary relation $\tilde{R}_{\text{tmp}}$ based on data schema obtained as $\text{schema}(R)$. We also create an **INSERT** query $\tilde{q}_{\text{INS}}$ that inserts R into $\tilde{R}_{\text{tmp}}$.

Now a relation $\tilde{R}_{\text{tmp}}$ is obtained that has all attributes required for the original query as well as has all the innermost operations of the original query pre-computed and stored in respective columns. Now we traverse the operations from E and substitute operations that are precomputed with references to corresponding cryptosets in $\tilde{R}_{\text{tmp}}$. We then traverse operations from E again and squash newly-formed structures of type $\{\xi, a\}$ into just a , where a is a reference to a cryptoset (i.e., a set of attributes) in $\tilde{R}_{\text{tmp}}$. Lastly, we squash references to cryptosets into a reference to a limited subset of required attributes in the cryptoset. Let $\hat{E}$ be the results of these actions over E . Now, if all expressions in E are single-pass, we form a single-pass query $\tilde{q}_{\text{final}} = \{\tilde{R}_{\text{tmp}}, \hat{E}, \emptyset\}$. Otherwise, we take a structure $\{\tilde{R}_{\text{tmp}}, \hat{E}, \emptyset\}$ and pass it through the procedure for multiple-pass query again, forming a new temporary table.

The transformation $\mathfrak{M} : \mathcal{Q}_{\text{SELECT}} \rightarrow \mathfrak{P}(\tilde{\mathcal{Q}}_{\text{DML}})$ is completely defined⁴.

⁴Some notes on the algorithm of transformation of **SELECT** queries.

1) In second and next stages of multiple-pass queries we do not supply a predicate. This is done so because temporary tables have tuples already filtered by the previous stage, so additional filtering would not give any improvement.

2) After the query is executed all temporary relations should be deleted.

```

SELECT (
  [Orders].[Amount] *
  [Orders].[Price]
) AS [Revenue]
FROM [Orders]
WHERE [Orders].[Customer] = 'June Kuphal'

SELECT [dbo].[ElGamalMultiplication] (
  [Orders].[Amount.ElGamal.Enc],
  [Orders].[Price.ElGamal.Enc],
  [Orders].[Common.ElGamal.P]
) AS [Revenue.ElGamal.Enc],
[Orders].[Customer.Text.Enc]
FROM [Orders]
WHERE [Orders].[Customer.Fingerprint] = 40707

```

Figure 4.5: Example of a SELECT DML query transformation

Figure 4.5 shows how a sample single-pass **SELECT** query with a select expression and a **WHERE** clause is transformed by the model implementation (see Chapters 6 and 7). **ElGamalMultiplication** is an operation from $\tilde{O}_v$ that corresponds to the plaintext operation of integer multiplication from O_v . It is easy to notice that the encryption-aware query requests column **[Customer]**, whereas the original plaintext query does not. That is done in order for the filtering function $\mathfrak{F}$ to have the data required to check whether the decrypted tuple satisfies the plaintext **WHERE** condition.

DELETE Queries

This section considers a transformation $\mathfrak{M} : \mathcal{Q}_{\text{DELETE}} \rightarrow \mathring{\mathfrak{P}}(\tilde{\mathcal{Q}}_{\text{DML}})$. Let $q \in \mathcal{Q}_{\text{DELETE}}$ be a **DELETE** query $q = \{d, w\}$.

Let us start with doing the first steps of transformation and converting d and w into $\tilde{d}$ and $\tilde{w}$, respectively, according to procedures described above. As it was discussed in the section dedicated to the transformation of **WHERE** predicates, a transformed predicate yields false positive tuples, which is inappropriate for such query as **DELETE**, as it will lead to deleting tuples that were not supposed to be deleted.

Let us create a **SELECT** query $\tilde{s} \in \tilde{\mathcal{Q}}_{\text{SELECT}}$, $\tilde{s} = \{\tilde{d}, *, \tilde{w}\}$, where $*$ denotes all attributes of $\text{schema}(\tilde{d})$. Let us also construct a new predicate

$$\hat{w}(t) \equiv \bigvee_i (t = t_i), \quad \forall t_i : t_i \in \{\Omega(\Delta(\tau)) \mid \tau \in \tilde{s}(\tilde{\mathcal{D}})\}.$$

Finally, the **DELETE** query $\tilde{q} \in \tilde{\mathcal{Q}}_{\text{DELETE}}$, $\tilde{q} = \{\tilde{d}, \hat{w}\}$ is formed. So, the original query q is transformed into a sequence of queries $(\tilde{s} \circ \tilde{q})$.

The transformation $\mathfrak{M} : \mathcal{Q}_{\text{DELETE}} \rightarrow \mathring{\mathfrak{P}}(\tilde{\mathcal{Q}}_{\text{DML}})$ is defined.

3) Results of both single-pass query and of query $\tilde{q}_{\text{final}}$ are then put through a filtering $\mathfrak{F}$ with the predicate $w \in q$ supplied, as stated in the section, where $\mathcal{Q}_{\text{DML}}$ and $\tilde{\mathcal{Q}}_{\text{DML}}$ are defined.

UPDATE Queries

Transformation of UPDATE queries $\mathfrak{M} : \mathcal{Q}_{\text{UPDATE}} \rightarrow \mathring{\mathfrak{P}}(\tilde{\mathcal{Q}}_{\text{DML}})$ combines approaches of all other types of queries. Let $q = \{d, d_1, \dots, d_m, a_1, \dots, a_n, e_1, \dots, e_n, w\}$ be an UPDATE query, where d is the relation that is supposed to be updated.

First, let us do the usual straightforward transformation $\{d, d_1, \dots, d_m\} \rightarrow \{\tilde{d}, \tilde{d}_1, \dots, \tilde{d}_m\}$.

We then construct a plaintext SELECT query $q_S = \{d, d_1, \dots, d_m, e_1, \dots, e_n, w\}$ and transform it into a multiple-pass encryption-aware equivalent $\{\tilde{q}_{\text{SEL}}^1, \tilde{q}_{\text{DDL}}^1, \tilde{q}_{\text{INS}}^1, \tilde{q}_{\text{SEL}}^2, \dots, \tilde{q}^{\text{final}}\}$ or a single-pass encryption-aware equivalent $\{\tilde{q}^{\text{final}}\}$, depending on the properties of q_S . We then append a $\tilde{q}_{\text{DDL}}^{\text{final}}$ and $\tilde{q}_{\text{INS}}^{\text{final}}$ queries that create a temporary relation $\tilde{r}_{\text{tmp}}$ with the results of the SELECT query $\tilde{q}^{\text{final}}$.

The system keeps track⁵, which tuple in the relation $\tilde{r}_{\text{tmp}}$ corresponds to which tuple in the relation $\tilde{d}$. Based on this information, a predicate $\hat{w}$ for the final UPDATE query is constructed aiming to do a natural join of these two relations.

Similarly to an INSERT query, an UPDATE query is supposed to update all elements of a cryptoset in the database. Thus, every plaintext attribute a_i is converted into a tuple of attributes $\{\tilde{a}_{i,1}, \tilde{a}_{i,2}, \dots\}$ that represents all elements of a respective cryptoset in the database.

A final UPDATE query $\tilde{q} \in \tilde{\mathcal{Q}}_{\text{UPDATE}}$ is:

$$\tilde{q} \equiv \{\tilde{d}, \tilde{r}_{\text{tmp}}, \{\tilde{a}_{1,1}, \tilde{a}_{1,2}, \dots, \tilde{a}_{n,1}, \tilde{a}_{n,2}, \dots\}, \{\tilde{r}_{\text{tmp}}^{1,1}, \tilde{r}_{\text{tmp}}^{1,2}, \dots, \tilde{r}_{\text{tmp}}^{n,1}, \tilde{r}_{\text{tmp}}^{n,2}, \dots\}, \hat{w}\},$$

where $\tilde{r}_{\text{tmp}}^{i,j}$ is an attribute in $\tilde{r}_{\text{tmp}}$ storing a pre-computed by the SELECT query element of a cryptoset that is a result of an expression that an attribute $\tilde{a}_{i,j}$ in $\tilde{d}$ is supposed to be updated with.

The transformation $\mathfrak{M} : \mathcal{Q}_{\text{UPDATE}} \rightarrow \mathring{\mathfrak{P}}(\tilde{\mathcal{Q}}_{\text{DML}})$ is completely defined.

4.3 Summary

This chapter formally introduces the abstract query execution model. As established in the previous chapter, the user issues plaintext queries, which are transformed into encryption-aware queries along the way by the intermediate agents. The idea of the query execution model is two define a mapping between plaintext and encryption-aware queries, such that a plaintext query is mapped into an encryption-aware query

⁵As discussed in later chapters, it could be done by to keeping unique tuple IDs in every tuple—a simple and efficient approach.

(or a superposition of queries) with the same semantics.

First in the chapter, the two core assumptions about the existence of operations over encrypted data are stated. Then, a full definition of the subset of SQL that is supposed to be supported by the encrypted database is provided. The subset is split into components the same way the whole SQL is split into the TCL, DDL, DML, etc. In order to formalize the mapping, a space of plaintext queries and a space of encryption-aware queries are defined. Then, a sub-space by sub-space the whole space of plaintext queries is covered: a mapping for each element is formally defined.

Chapter 5

Transaction Management

It is frequent that certain data manipulations could not be done in a single query due to various reasons: limitations of the query language, intermediate data needs to be processed outside of the database, etc. The ultimate goal of a transaction mechanism is to provide ACID¹ properties for groups of DML queries. When a DML query is transformed according to the query execution model, the DBMS receives a different query, or a group of queries, which might lead to a violation of one or several of ACID properties. This chapter investigates how the query transformation affects ACID properties of individual queries and transactions.

5.1 Introduction to OLTP

One of the main concerns in preserving the semantics of transactions under the query execution model is transaction isolation.

When a database is used by a single user at a time there is no need for the isolation: a typical relational database ensures that queries from the user are executed in the order they were received and the next query does not start to be executed until the previous one is finished. However, if there are simultaneous sessions, one session might observe inconsistent data when another session is in the middle of a multi-query data manipulation.

A textbook understanding of a transaction relies on a notion of *serializability*. Consider this definition:

Definition 5.1 (Serializability). Transaction scheduling and execution strategy is called *serializable* if the result of such strategy is identical to the result of a strategy

¹Atomicity, Consistency, Isolation, and Durability. See Chapter 1, or later in this chapter for more details.

that executes transactions strictly one at a time in the order they were received by the DBMS. ◁

As long as the DBMS's transaction management strategy is guaranteed to be serializable, the DBMS may run transactions in any way it desires to; Typically, DBMSs attempt to run transactions concurrently, which drastically increases efficiency in high-load scenarios. However, many practical DBMSs by default provide a much weaker isolation property for their transactions, while maintaining the ability to promote the scheduling policy to guarantee serializability. A weaker policy is often enough but allows to improve query throughput even more. There is a multitude of isolation policies supported by modern DBMSs and they vary from one to another. This research is focused on showing correctness under the serializable isolation.

A typical example of when an isolated transaction is needed is an operation of money transfer in a bank. Alice owes Bob \$100 and issues a request to transfer the money from her account to Bob's. The bank needs to deduct \$100 from Alice's account and credit \$100 to Bob's account, which is done with two separate **UPDATE** queries. If Carol, the bank manager, happens to query an overall amount in Alice's and Bob's accounts right in between these two update queries, she will receive an incorrect amount—the money were already deducted from Alice's account and were not yet credited to Bob's. Scenarios of such nature happen frequently in many applications, and thus the transaction isolation mechanism is required.

Generally, DBMSs use *locking* or *snapshots* as a solution. The simplest way is to lock the tables that are required by the transaction, although most DBMSs support record-granularity for locking so that concurrent transactions could work with the same table, provided that they access different sets of records. Practical solutions usually have various types of locks, such as shared and exclusive and some other types of locks, and are able to lock either tables or individual tuples. Snapshots is a different strategy, which maintains several versions of the database, one for each transaction. A snapshot of data is made at the beginning of the transaction so that it works with its own stable version of data, which is inaccessible to other transactions. The DBMS guarantees correct resolution of conflicts when the snapshots are merged back into the main database after the transaction is finished.

The property of consistency in the terminology of ACID properties means that in between queries and/or transactions all data constraints such as foreign keys, triggers, etc. are valid. Data constraints are not considered in the scope of current research, and thus consistency (in terms of ACID properties) is out of the scope as well.

Another important property of transactions is atomicity. Atomicity property states that transactions are either executed in full or are not executed at all. If something goes wrong during the execution of a transaction—either an external failure such as power outage, or an error occurs during execution of one of the queries in the transaction, or a business logic decision is made that the transaction should not finish—none of the current results of the transaction are reflected in the database. Only when the transaction explicitly states that it is finished (the transaction *commits*) should the database apply all the changes the transaction has made.

A related property is the property of durability. This property states that once the transaction has committed, all the changes are persistently stored. In other words, external events that happened afterwards can not affect the results of a committed transaction—unless the data storage gets corrupted, of course.

There are several approaches for databases to ensure atomicity and durability. However, since these mechanisms are “external” to the database, and this work explores the wrapper approach, the model has no access to these mechanisms. On the other hand, the DBMS provides certain guarantees regarding atomicity and durability of transactions and individual queries. Leveraging these guarantees it is possible to investigate how to achieve same properties for the queries transformed by the query execution model (QEM).

Providing guarantees of ACID properties under the QEM requires understanding of the implementation details of a specific DBMS and how it manages transactions. We assume that the DBMS supports a serializable transaction isolation and provides means to explicitly put exclusive locks on specific tuples. We also assume that for a query or a transaction to be atomic it is enough that unless committed it does not change/add/remove any data in any of existing relations; stray temporary relations and unreleased locks are considered to not violate atomicity, as all practical DBMSs have means of automatic clean-up of such artifacts.

5.2 Atomicity under the QEM

When a user issues his queries against a virtual data schema, he expects that 1) a single query is atomic; 2) a transaction is atomic. Remembering that the model of interaction between the user and the DBMS looks like this: User $\longleftrightarrow$ Proxy $\longleftrightarrow$ DBMS, the atomicity will be considered from two perspectives. First, we will consider atomicity of transformed single queries from the proxy standpoint. Second, we will

consider atomicity of transformed queries from the user standpoint.

5.2.1 Proxy Standpoint

In this section, let us consider the atomicity of transformed single queries from the proxy standpoint.

Lemma 5.1. A transformed INSERT query is atomic from the proxy standpoint. $\triangleleft$

Proof. According to the QEM, INSERT queries are always transformed into a single query. Thus, in the case when the proxy acts as a query issuer, the transformed INSERT queries have exactly same atomicity guarantees as the original ones. $\square$

With regards to a SELECT query, a notion of atomicity is extended to also cover the results the query produces. The resulting set should be received by the querier either in full, or not received at all. In practice, it is considered enough if the querier is guaranteed to be able to tell that the result he received is incomplete.

Lemma 5.2. A transformed SELECT query is atomic from the proxy standpoint. $\triangleleft$

Proof. Single-pass SELECT queries are transformed into a single SELECT query, and thus maintain all the atomicity properties of a single SELECT query with regards to both changes of the database state and completeness of the results.

The way the transformation of multiple-pass SELECT queries is defined under the QEM, the only way they are able to modify any data in the DBMS is in temporary relations. The user does not receive any data until the final query is completely and atomically (since the final query is a single-pass query) executed.

According to the QEM, any multiple-pass SELECT query has a form $q_{\text{SEL}} \rightarrow q_{\text{INS}} \rightarrow q_{\text{SEL}} \rightarrow q_{\text{INS}} \rightarrow \dots \rightarrow q_{\text{SEL}}^{\text{final}} \iff \underbrace{q_{\text{SEL}} \rightarrow q_{\text{INS}}}_{n \geq 1 \text{ times}} \rightarrow q_{\text{SEL}}^{\text{final}}$. Let us consider possible points of failure:

$$q_{\text{SEL}} \xrightarrow{p1} q_{\text{INS}} \xrightarrow{p2} \underbrace{q_{\text{SEL}} \xrightarrow{p3} q_{\text{INS}} \xrightarrow{p4}}_{n \geq 0 \text{ times}} q_{\text{SEL}}^{\text{final}}$$

Failure point $p1$ Only a single SELECT query was executed up to that point, which does not modify the database in any way.

Failure point $p2$ A single INSERT query was executed since previous failure point.

According to the QEM, this query inserts data into a temporary relation, which, by the assumption given in the beginning of this chapter, does not violate atomicity.

Failure point $p3$ Since the last failure point only a **SELECT** query was executed, which does not affect the database.

Failure point $p4$ An **INSERT** query was executed, but it inserted data into a temporary relation, which is not a violation of atomicity.

This discussion has shown that a multiple-pass **SELECT** query is atomic with regards to the database state. Now let us consider its atomicity in terms of producing the result set.

As it is defined by the QEM and is illustrated by the diagram showing the possible failure points, none of the queries before the query $q_{\text{SEL}}^{\text{final}}$ do produce any parts of the result set: the result set is completely produced by the query $q_{\text{SEL}}^{\text{final}}$. The query $q_{\text{SEL}}^{\text{final}}$, since it is a regular **SELECT** query, maintains the atomicity property of such. Consider these possibilities:

Proxy failure Proxy fails during the execution of $q_{\text{SEL}}^{\text{final}}$. Since the proxy has failed, it is irrelevant for it whether the results it received were complete or not.

DBMS failure The database server fails during the execution of $q_{\text{SEL}}^{\text{final}}$. The proxy has possibly started receiving the results from the database. Assuming that the communication protocol allows the querier to tell whether the received data is incomplete (e.g., control sums, boundaries, etc.), the proxy would be able to dismiss the incomplete result.

Network failure The connection between the proxy and the DBMS is lost during the execution of $q_{\text{SEL}}^{\text{final}}$. From the proxy's perspective, this case is identical to the DBMS failure and has same conclusions.

We have shown that **SELECT** queries are executed atomically and at worst leave stray temporary relations in case of failure happening before the DBMS is instructed to dispose of them; we have also shown that **SELECT** queries are atomic in terms of produced results, assuming that the communication protocol provides verification data, which they typically do. $\square$

Lemma 5.3. A transformed **DELETE** query is atomic from the proxy standpoint. $\triangleleft$

Proof. **DELETE** queries are transformed into either a single **DELETE** query (a single-pass query); or a pair of queries (a multiple-pass query)—a **SELECT** query and a **DELETE** query.

Single-pass **DELETE** query preserves all the properties of the original query, and thus is atomic.

In case of a multiple-pass DELETE query, the SELECT query retrieves a superset of tuples that are supposed to be deleted, the proxy filters out tuples that are false-positives, and then the DELETE query is executed against a narrowed down set of tuples. By the definition in QEM, first SELECT query is a single-pass query. Clearly, it is required that the SELECT query acquires exclusive locks on all the tuples it returns.

Let us consider the query execution plan:

$$q_{\text{SEL}} \ \& \ \text{LOCK} \xrightarrow{p1} q_{\text{DEL}} \xrightarrow{p2} \text{RELEASE LOCKS}$$

“ $q_{\text{SEL}} \ \& \ \text{LOCK}$ ”, “ q_{DEL} ”, and “RELEASE LOCKS” are single queries and thus are atomic. Let us analyze the points of potential failure.

Failure point $p1$ Only a SELECT query is executed up to that point, and records are locked. SELECT query does not modify data, hence failure at this point will at worst result in unreleased locks, which is not a violation of atomicity by our assumption.

Failure point $p2$ Failure at this point happens after the data was atomically modified by a single, and thus atomic, DELETE query and, again, at worst results in unreleased locks.

This brings us to a conclusion that transformed DELETE queries may be considered atomic. $\square$

Lemma 5.4. A transformed UPDATE query is atomic from the proxy standpoint. $\triangleleft$

Proof. As defined by the QEM, UPDATE queries are transformed into a series of queries: a SELECT, a CREATE TABLE, an INSERT, and an UPDATE query. We have shown atomicity of SELECT and INSERT queries, and a single final UPDATE query is atomic as well. DDL queries, such as the CREATE TABLE query, are not in the scope for atomicity.

Let us consider a query execution plan:

$$q_{\text{SEL}} \ \& \ \text{LOCK} \xrightarrow{p1} q_{\text{INS}} \xrightarrow{p2} q_{\text{UPD}} \xrightarrow{p3} \text{RELEASE LOCKS}$$

Failure point $p1$ Failure at this point occurs after a SELECT query, which does not modify the data. This failure results in unreleased locks, which does not violate atomicity.

Failure point $p2$ Failure at this point leaves a stray temporary relation and unre-

leased locks, which is by our convention acceptable, but no real data has been modified in the DBMS.

Failure point p_3 Failure at this point would happen after the data was atomically modified by the **UPDATE** query and would only result in unreleased locks.

We have shown that transformed **UPDATE** queries are atomic. $\square$

Theorem 2. All transformed DML queries are atomic from the proxy standpoint. $\triangleleft$

Proof. The theorem is equivalent to saying that for $\forall q \in \mathcal{Q}_{\text{DML}}$, $\mathfrak{M}(q)$ is atomic.

By definition, $\mathcal{Q}_{\text{DML}} \equiv \mathcal{Q}_{\text{INSERT}} \cup \mathcal{Q}_{\text{UPDATE}} \cup \mathcal{Q}_{\text{DELETE}} \cup \mathcal{Q}_{\text{SELECT}}$. Thus, for $\forall q \in \mathcal{Q}_{\text{DML}}$, one and only one of the following is true: $q \in \mathcal{Q}_{\text{INSERT}}$, $q \in \mathcal{Q}_{\text{UPDATE}}$, $q \in \mathcal{Q}_{\text{DELETE}}$, or $q \in \mathcal{Q}_{\text{SELECT}}$. Lemmas 5.1, 5.4, 5.3, and 5.2, respectively, show that $\mathfrak{M}(q)$ is atomic in any of these cases.

The theorem is proven. $\square$

We have shown atomicity of single queries from the proxy standpoint.

5.2.2 User Standpoint

This section considers atomicity of transformed single queries from the user standpoint. Let us recall the interaction scheme: User $\longleftrightarrow$ Proxy $\longleftrightarrow$ DBMS. The part Proxy $\longleftrightarrow$ DBMS was considered in the previous section, and it was shown that transformed queries are executed atomically. Based on that, from the point of view of an external observer the part of communication Proxy $\longleftrightarrow$ DBMS could be considered as a single entity that executes incoming queries atomically. Thus, in this section the atomicity of this model is considered: User $\longleftrightarrow$ [Proxy & DBMS]. A group of proxy and DBMS will be called a “remote system” in this section.

Table 5.1 illustrates stages of query execution from the user’s point of view and locations, at which a failure might occur. Only a **SELECT** query has the stage of execution S_3 , as the other queries do not produce any results for the user. To ensure atomicity, queries need to be analyzed at every stage for behavior in case of a failure at any of the locations.

Lemma 5.5. Transformed **INSERT**, **DELETE**, and **UPDATE** queries are atomic from the user standpoint. $\triangleleft$

Proof. Let us consider every stage individually.

Stage S_1 The query is in transfer from the user to the remote system. Analysis of failures in every location:

		Location		
		<i>L1: User</i>	<i>L2: Network</i>	<i>L3: Proxy&DBMS</i>
Stage	<i>S1</i>		transfer→	
	<i>S2</i>			processing
	<i>S3</i>		←transfer	

Table 5.1: Failure points and locations for atomicity analysis

Location *L1* A failure of the user terminates the transfer of the query. Since the query was never fully received by the remote system, it will not start execution and thus will not affect the database.

Location *L2* A network failure will result in the same effect.

Location *L3* If the remote system fails during the transfer, it has never received the full query, and thus has not started its execution, which leaves the database state unchanged.

Stage *S2* The query was fully received and its execution has started, but is not yet finished. Analysis of failures in the locations:

Location *L1* A failure of the user at that point will not affect the execution of the query. If no other failures occur at this stage, the query will be *fully* executed, which is an acceptable outcome in terms of atomicity.

Location *L2* A failure of the network also will have no effect on the execution of the query. Similarly, it will be fully executed in case of a failure at *L2* during stage *S2*.

Location *L3* As it was already discussed, a failure of either the proxy, the DBMS, or the connection between them will result in query execution being aborted. Since, the atomicity from the proxy standpoint was already proved (Theorem 2), such termination will not change the database state.

Stage *S3* The query was fully executed and the results are being transferred back. None of the queries considered in this lemma produce the results for the user, so this stage does not occur.

The lemma is proven. □

Lemma 5.6. A transformed **SELECT** query is atomic from the user standpoint. ◁

Proof. Unlike the previously considered queries, a **SELECT** query does not modify the database, but produces a result for the user. As discussed earlier, from the user's point of view, atomicity of a **SELECT** query implies that the user either gets the full

result, or no result at all. It is easy to observe that a failure at any location during the stages $S1$ and $S2$ will result in user receiving no results at all, which is acceptable.

Let us consider stage $S3$: the results are being transferred back to the user.

Location $L1$ If the user fails while receiving the results, the atomicity property of a `SELECT` query ceases to exist.

Locations $L2$ and $L3$ If either the network fails or the remotes system fails, the user receives a partial result. Generally speaking, it is possible that the partial result seems to the user to be valid and the atomicity is violated. However, a network protocol that is used as a transport for the result set is very likely (this may be stated as a requirement of the model) to explicitly mark the boundaries of the data in transfer. If this is the case, the user is able to say definitely whether the result he has is partial or not.

The lemma is proven under the assumption that the communication protocol provides meta-data for the client. □

Theorem 3. A transformed DML query is atomic from the user standpoint. ◁

Proof. This is a direct corollary of lemmas 5.6 and 5.5. □

We have discussed the atomicity of single queries in details. Atomicity of transactions is considered in Section 5.5.

5.3 Isolation under the QEM

Similar to atomicity, the user expects that **1)** a single query is isolated; **2)** a transaction is isolated. In this section we consider isolation of single queries.

Lemma 5.7. A transformed `INSERT` query is isolated. ◁

Proof. Since, according to the QEM, an `INSERT` query is always transformed into a single query, it preserves the isolation guarantees of the original `INSERT` query, which matches the querier expectations. □

Lemma 5.8. A transformed `SELECT` query is isolated. ◁

Proof. A single-pass `SELECT` query is, by definition, transformed into a single `SELECT` query, which means it maintains all expected isolation properties.

Multiple-pass `SELECT` queries are translated into a series of queries, which brings forward the problem of isolation. Let us consider the way multiple-pass `SELECT`

queries work:

$$\underbrace{q_{\text{SEL}} \longrightarrow q_{\text{INS}}}_{n \geq 1 \text{ times}} \longrightarrow q_{\text{SEL}}^{\text{final}}$$

It is easy to notice that after the first q_{SEL} query all the consequent ones work exclusively with the temporary relation. At the same time, the DBMS guarantees that the temporary relation is only accessible by the current user. Effectively, a multiple-pass **SELECT** query implements a snapshot strategy for isolation and works with the snapshot of data as it was at the moment of first **SELECT** query, which means that it maintains the isolation property expected by the user.

Transformed **SELECT** queries are shown to be isolated. $\square$

As **DELETE** and **UPDATE** queries are commonly translated into a series of queries, isolation is a big concern: another user might issue a DML query in the middle of the pair of queries and affect the results.

Lemma 5.9. Transformed **UPDATE** and **DELETE** queries are isolated. $\triangleleft$

Proof. As discussed above, isolation is often achieved by using locks on tables and/or tuples. However, as shown in the proofs of Lemma 5.3 and Lemma 5.4, both **DELETE** and **UPDATE** queries in the first place obtain exclusive locks on a superset of tuples that are going to be affected by the actual **DELETE** or **UPDATE** query to guarantee that no other session might affect these tuples, which by definition means isolation.

All other intermediate actions that an **UPDATE** query performs between the first **SELECT** and an actual **UPDATE** are done in the temporary relation, which any DBMS typically guarantees to be only accessible by the current session.

The lemma is proven. $\square$

Theorem 4. All transformed DML queries are isolated. $\triangleleft$

Proof. An equivalent statement is that for $\forall q \in \mathcal{Q}_{\text{DML}}$, $\mathfrak{M}(q)$ has a property of isolation.

By definition, $q \in \mathcal{Q}_{\text{DML}}$ means that q belong to either $\mathcal{Q}_{\text{INSERT}}$, $\mathcal{Q}_{\text{UPDATE}}$, $\mathcal{Q}_{\text{DELETE}}$, or $\mathcal{Q}_{\text{SELECT}}$. For each of these cases, one of the lemmas 5.7, 5.8, and 5.9 shows that $\mathfrak{M}(q)$ is isolated.

The theorem is proven. $\square$

We have shown that single queries maintain isolation properties, and isolation of transactions is considered in the Section 5.5.

5.4 Durability under the QEM

Durability property states that any changes an atomic operation has made to the data in the database are persistently stored. Failure, power interruption, system restart should not affect results of operations that were finished. Obviously, in terms of single queries, durability is only applicable to **INSERT**, **UPDATE**, and **DELETE** queries, since **SELECT** queries do not modify the data. This section considers durability of single queries under the QEM.

Lemma 5.10. Transformed **INSERT** queries are durable. $\triangleleft$

Proof. In the same way as before, since **INSERT** queries are always transformed into a single **INSERT** query, the durability property is maintained. $\square$

Lemma 5.11. Transformed **DELETE** and **UPDATE** queries are durable. $\triangleleft$

Proof. Let us consider execution plans of the transformed **DELETE** and **UPDATE** queries:

$$\begin{aligned} \text{DELETE query:} \quad & q_{\text{SEL}} \longrightarrow q_{\text{DEL}} \\ \text{UPDATE query:} \quad & q_{\text{SEL}} \longrightarrow q_{\text{INS}} \longrightarrow q_{\text{UPD}} \end{aligned}$$

We can notice that after transformation, actual modification of the data in the database, not in temporary relations, is only done in the final query that actually does perform an **UPDATE** or a **DELETE**, and all preceding queries are only preparing the data for it and are already shown not to modify any data in the database, except for temporary relations; by definition of QEM, q_{INS} in the **UPDATE** query inserts records into a temporary relation as well.

Thus, the modification to the data maintains same durability properties as a single **UPDATE** or **DELETE** query, which matches the user's expectations. The lemma is proven. $\square$

Theorem 5. Transformed **INSERT**, **UPDATE**, and **DELETE** queries are durable. $\triangleleft$

Proof. This theorem is a direct corollary of lemmas 5.10, and 5.11. $\square$

We have shown that in all applicable cases, the property of durability is maintained upon query transformation.

5.5 Transactions under the QEM

Since the encrypted database model is designed under the wrapper approach, significant efforts are directed at relying on the capabilities of the DBMS as much as possible. It was shown that from the point of view of ACID properties, all single plaintext DML queries maintain their properties when transformed into their encryption-aware equivalents. In this section let us consider how ACID properties of transactions are preserved.

It is easy to notice that all plaintext SQL queries that the user issues are translated into regular SQL queries as well. Thus, if a group of queries is preceded by a **BEGIN TRANSACTION** query, from the external point of view the transaction will behave correctly and maintain atomicity of itself (all changes a transaction has made are atomically written when a **COMMIT** query is received; nothing is written in case of a failure or if a **ROLLBACK** query was issued), isolation from other transactions, and durability of changes upon receiving a **COMMIT** query.

When inside of a serializable transaction, it is not sensible to worry about the ACID properties of individual queries anymore: atomicity does not make sense, since in case of a failure the whole transaction is rolled back; isolation of individual queries is irrelevant, as the whole transaction is isolated; durability does not make sense as well, since all the changes are written to a persistent storage only by the time the transaction is committed, and they are written in a durable fashion as guaranteed by the DBMS.

As a corollary of the aforementioned, the queries from the set $\mathcal{Q}_{\text{TCL}}$ are passed directly to the DBMS without any modification. Formally speaking, $\tilde{\mathcal{Q}}_{\text{TCL}} \equiv \mathcal{Q}_{\text{TCL}}$, and $\mathfrak{M} : \mathcal{Q}_{\text{TCL}} \rightarrow \tilde{\mathcal{Q}}_{\text{TCL}}$ is defined as $\mathfrak{M}(q) = \tilde{q}$, where $q \in \mathcal{Q}_{\text{TCL}}$, $\tilde{q} \in \tilde{\mathcal{Q}}_{\text{TCL}}$, and $\tilde{q} = q$.

5.6 Summary

This chapter considered a set of issues that arise when the query execution model is being implemented as a practical solution: failures of either part of the system might result in lost, corrupt, incomplete, or inconsistent data; and multiple concurrent sessions might interfere with each other.

Satisfying the ACID criteria is equivalent to solving these issues. The chapter demonstrated that individual queries upon transformation preserve atomicity, isolation, and durability properties. Then, it was demonstrated that transactions preserve

these properties as well. Issues related to consistency (in terms of ACID criteria) are out of the scope of this work.

Lastly, the last bit of query execution model—the TCL part—was defined, hence finalizing the formal definition of the model.

Chapter 6

Model Implementation

This chapter discusses a practical implementation of an encrypted database based on the query execution model. The implementation illustrates how the model can be used in practice given the functional and other requirements. Here it is assumed that functional requirements are as follows: all data is regular 32 bit integers; required operations are equality matching, addition, and multiplication; not all attributes require all operations; threat model is same as the one assumed for this work.

6.1 Storage Model

As it has been shown in the previous chapters, the foundational element of the proposed model is the cryptoset. In the end of the day, the implementation of the system is supposed to store cryptosets in the DBMS in the form of tuples in the tables. In addition, it is needed to store some meta-data in order to make possible the execution of the operations. This section discusses approaches to that.

6.1.1 Viable Approaches

Due to the nature of the cryptoset that comprises of several representations of the same plaintext data value, it is theoretically possible that cryptoset is in an inconsistent state at some points of the time. The responsibility for maintaining it in the consistent state might be attached to various components of the system; however, in the wrapper approach the storage component is very passive; moreover, the query execution model was designed to handle the cryptosets in a consistent state by various means, including the use of the ξ operation.

Using a multiple tables approach (see Figure 6.1), every plaintext table transforms into a set of multiple tables, each holding subsets of elements of cryptosets. For example, in the illustrative figure the original plaintext table T has two columns A

Figure 6.1: Illustration of the multiple tables approach to storage model

and B. Both these attributes support searching so the table **T Search** has both columns populated with respective encryptions. Similarly, only the column A supports multiplication and only the column B supports addition. None of the columns support greater/less checks. This approach requires the database to maintain semantic links between the records in multiple different tables, which produces an unnecessary overhead; there are no clear advantages of such approach.

Another approach, the so-called “onion” approach (see Figure 2.8) is used by the CryptDB. As the analysis in the Section 2.5 shows, this approach does not give noticeable security improvements and creates a computational overhead.

Figure 6.2: Illustration of the single table approach to storage model

Under the rules of the single table approach (Figure 6.2), each plaintext table transforms into a single encrypted table, where each plaintext attribute value is translated into a set of encrypted attributes (a cryptoset), which are all stored in the same record. This approach was chosen as a more suitable for the current application.

6.1.2 Proposed Approach

The single table approach to storing cryptosets and related metadata is working well along the lines of the query execution model and the way it handles the transformation of the data schema. At the same time, it has a controllable overhead in terms of storage and little to no added complexity. It thus seems to be a suitable approach for

the purposes of this work.

As it will be discussed in more details in the following sections, the specific implementation of the model that is being created, relies on such cryptosystems as Paillier and ElGamal, and also certain hashing functions. The Paillier and ElGamal homomorphic addition and multiplication operations require the knowledge of certain parts of public key to execute; specifically, Paillier homomorphic addition requires a public parameter N , and ElGamal homomorphic multiplication requires a public parameter P .

Also, as discussed earlier, an operation of tuple equality that is required by the encryption-aware UPDATE and DELETE queries, needs every record to have a unique identifier. So, every table additionally gets a `rowId` attribute that is handled by the DBMS to be unique in every row. Popular DBMSs support such attributes: in MySQL that would be defined along the lines of “`‘rowId‘ BIGINT UNSIGNED NOT NULL AUTO_INCREMENT`”; in MS SQL Server similar to “`[rowId] UniqueIdentifier NOT NULL DEFAULT NewID()`”.

It is assumed for this implementation that the domain $\mathcal{C}$ is in practice a domain of 32-bit integers, so all plaintext values belong to the SQL type `INTEGER`. With all that in mind, the very basic and straightforward way to implement the single table approach would be for every attribute denoted in the original plaintext data definition as `<attribute name> INTEGER` to convert it to a cryptoset that stores the required encryptions and hashes, and the required public parameters. Assuming that 1024 bit keys are used, the result (for Transact-SQL) would be:

```
[rowId] UniqueIdentifier NOT NULL DEFAULT NewID(),  
  
[<attribute name 1>.Paillier.Enc] BINARY(256),  
[<attribute name 1>.ElGamal.Enc] BINARY(256),  
[<attribute name 1>.Paillier.N] BINARY(256),  
[<attribute name 1>.ElGamal.P] BINARY(256),  
[<attribute name 1>.Fingerprint] SMALLINT,  
...  
[<attribute name n>.Paillier.Enc] BINARY(256),  
[<attribute name n>.ElGamal.Enc] BINARY(256),  
[<attribute name n>.Paillier.N] BINARY(256),  
[<attribute name n>.ElGamal.P] BINARY(256),  
[<attribute name n>.Fingerprint] SMALLINT
```


Fingerprint is a hash that is used for search and is considered in details below. For this implementation we choose a 16 bit fingerprints, so it is of type `SMALLINT`, which is a 16 bit integer.

6.1.3 Analysis and Improvements

It is easily seen that data expansion factor is significant. For a table in the virtual data schema with n attributes we get an expansion from the original $4n$ bytes to $(4 \times 256 + 2) \cdot n + 16 = 1026n + 16$ bytes of encrypted data, where 16 bytes is the size of the `UniqueIdentifier` type for the `rowId`. This results in an approximate data expansion factor of ~ 256 – 257 times. However, it is possible to reduce the storage size expansion.

As the intention is to be able to execute arithmetic operations with values that are potentially in different columns, we use same encryption keys across the whole database. That allows the system not to store the public encryption parameters in every cryptoset. Moreover, it is even possible not store them in every record and store them in a separate service table instead; some DBMSs allow for persistent global variables, which might be used for that. This reduces the size requirement almost twice to $514n + 16$ bytes. An “asymptotic” estimate of data expansion factor thus would be ~ 128 – 129 times.

Also, one could notice that in every specific usage scenario certain operations will never be executed with certain columns. As an example, it makes no sense to add or multiply such values as UserIDs, so in practice it is possible reduce certain cryptosets to subcryptosets with just the elements that are actually required, which will reduce the storage expansion by a varying but often a significant factor. Knowing that no operations will ever be executed with a specific attribute, one can reduce its subcryptoset even further, to just an encryption that uses a non-deterministic cryptosystem with shorter block size and a fingerprint, which would still be a valid subcryptoset by definition.

6.1.4 Cryptoset Security Analysis

As it is discussed later in this chapter, the model implementation relies on ElGamal and Paillier cryptosystems to provide arithmetic operations building blocks. This section is dedicated to the security analysis of a cryptoset that contains these two encryptions.

A fundamental notion in security research is such of a negligible function.

Definition 6.1 (Negligible function). A function $f : \mathbb{N} \rightarrow \mathbb{R}$ is called *negligible*

if for $\forall p \in \mathbb{R}[x]$, where p is a non-zero polynomial, $\exists m \in \mathbb{N}$: $|f(n)| < \frac{1}{|p(n)|}$, for $\forall n > m$. $\triangleleft$

Based on the notion of a negligible function, ciphertext indistinguishability is defined. Informally, the property of indistinguishability states that given two messages and a ciphertext for one of them, it is impossible to tell, which message does the ciphertext correspond to.

Definition 6.2 (Indistinguishability/IND-CPA). A cryptosystem is said to be indistinguishable under chosen plaintext attack, or IND-CPA, if a probabilistic polynomially-bounded adversary can not win the following game with probability greater than $1/2 + \nu(\varepsilon)$, where ε is a predefined security parameter, and ν is a negligible function:

1. The challenger instantiates the cryptosystem with security parameter ε , and publicizes the public key.
2. The adversary obtains the public key, and then is allowed to a polynomially-bounded number of encryptions, or other computations.
3. The adversary chooses two different messages m_0, m_1 and sends them to the challenger.
4. The challenger chooses a bit $b \in \{0, 1\}$ at random, and responds to an adversary with an encrypted message $\tilde{m}_b$.
5. The adversary is allowed to a polynomially-bounded number of encryptions, or other computations, after which he responds with a bit $r \in \{0, 1\}$.

The adversary wins the game if $r = b$. $\triangleleft$

Probabilistic cryptosystems are such cryptosystems that produce different valid ciphertexts for the same plaintext value each time they are invoked. Probabilistic cryptosystems are less prone to statistical attacks and generally do not leak as much about the structure of the encrypted data as the deterministic cryptosystems. Probabilistic cryptosystems generally rely on a source of randomness, which is used in every encryption—a randomizer. Let us define a notion of semantic security for probabilistic cryptosystems.

Definition 6.3 (Semantic security). Let $\mathcal{P}$ be a finite set of possible plaintexts; $\mathcal{C}$ —a finite set of ciphertexts; $\mathcal{K}$ —a finite set of possible keys; and $\mathcal{R}$ —a set of randomizers. Let ε be a specified security parameter. Let us define for any fixed $K \in \mathcal{K}$ and for $\forall x \in \mathcal{P}$ a probability distribution $p_{K,x}$ on $\mathcal{C}$, where $p_{K,x}(y)$ is the probability that y is the ciphertext for x for a given key K and for $\forall r \in \mathcal{R}$. If for $\forall x_1, x_2 \in \mathcal{P}, \forall K \in \mathcal{K}$,

where $x_1 \neq x_2$, probability distributions p_{K,x_1} and p_{K,x_2} are ε -indistinguishable, then the cryptosystem is said to have semantic security. $\triangleleft$

To put it simpler, semantic security implies that given a ciphertext the adversary could gain as much information about the plaintext value, as if he would have without the ciphertext. Goldwasser and Micali have shown that IND-CPA is equivalent to semantic security [35].

The ElGamal cryptosystem was first introduced in 1985 and is based on the decisional Diffie-Hellman problem [28]. It is notable for having multiplication-homomorphic properties. The definition of the cryptosystem is not given in this work for the sake of brevity. The ElGamal cryptosystem was proven to be semantically secure [41].

Theorem 6 (ElGamal semantic security). ElGamal ciphertexts are semantically secure if the decisional Diffie-Hellman problem (DDH) is hard in $\mathbb{G}$. $\triangleleft$

Proof. Assume the DDH problem is hard in group $\mathbb{G}$, and the adversary has an access to an oracle able to distinguish ElGamal ciphertexts in $\mathbb{G}$. Let us show that being tasked to distinguish between (g^x, g^y, g^{xy}) and (g^x, g^y, g^r) , where $r \in \mathbb{G}$ is random (the DDH problem), the adversary can use the ElGamal oracle to solve it.

The adversary is given a challenge (g^x, g^y, g^k) . He creates an ElGamal ciphertext $c = (g^y, g^k)$. If $g^k = g^{xy}$, then c is a valid encryption of 1. If $g^k \neq g^{xy}$, then c is an encryption of some random element of the group $\mathbb{G}$.

The adversary requests the ElGamal oracle to give him two messages m_0 and m_1 . He then computes a ciphertext $c_0 = (g^y, m_0 g^k)$, which is a valid encryption of m_0 if $g^k = g^{xy}$. The adversary then gives the oracle the ciphertext c_0 . The oracle will either respond with $b = 0$ if $g^k = g^{xy}$, or will not respond otherwise, since in that case c_0 is an encryption of a random element of $\mathbb{G}$, which is not equal to m_1 with overwhelming probability.

If the oracle returns $b = 0$, the adversary concludes that the challenge was (g^x, g^y, g^{xy}) . Otherwise, it was (g^x, g^y, g^r) . Thus, the adversary solves the DDH problem, which is assumed to be hard. Hence, the assumption that ElGamal is not semantically secure does not hold, and semantic security of the ElGamal cryptosystem is proven. $\square$

Paillier cryptosystem is based on hardness of the decisional composite residuosity class problem (D-Class[n]) [54]. This cryptosystem maintains a property of additive

homomorphism, and was also shown to be semantically secure [41]. Only an outline of the proof is provided here.

Theorem 7 (Paillier semantic security). Paillier ciphertexts being semantically secure is equivalent to D-Class[n] problem being hard. $\triangleleft$

Proof. Assume m_0 and m_1 are known messages, and c is a Paillier encryption of either one of them. Assuming that c is an encryption of m_0 , $cg^{-m_0} = r^n$ is an n 'th residue. If the assumption does not hold, then it is not, with overwhelming probability.

Clearly, if an adversary is able to decide n 'th residues, then he can check whether cg^{-m} is an n 'th residue, and with that ability he can distinguish encryptions of messages.

Now, assuming that an adversary can distinguish between the encryptions of m_0 and m_1 , then, given (w, x) , the adversary can find $y : g^x y^n = w \Leftrightarrow y^n = g^{-x} w$, which means $g^{-x} w$ is an n 'th residue. Choosing a valid value g , the adversary can obtain $c = g^{m_0} g^{-x} w$, which is a valid ciphertext. Thus, the ability to distinguish between the encryptions of m_0 and m_1 can be used to determine whether c is a valid encryption for m_0 , which is equivalent to solving D-Class[n]. $\square$

Now we can proceed to analysis of security properties of a cryptoset that consists of Paillier and ElGamal encryptions. Let us consider a more general case—a cryptoset with multiple semantically secure encryptions.

Theorem 8 (Cryptoset semantic security). A cryptoset that consists of encryptions, each of which is semantically secure, is semantically secure. $\triangleleft$

Proof. Assume a cryptoset consists of encryptions $e_1(\cdot), \dots, e_n(\cdot)$, each is semantically secure. Assume for two messages and a cryptoset of either one of them, an adversary has access to an oracle that can tell, which message does the cryptoset correspond to.

Let us define a challenge for the adversary: given m_0, m_1 and a ciphertext $c = e_i(m_b), b \in \{0, 1\}, i \in \{1, \dots, n\}$, he needs to respond with a bit $r \in \{0, 1\}$. If $r = b$, he wins.

Adversary constructs a cryptoset $\psi_0 = \{e_1(m_0), \dots, e_{i-1}(m_0), c, e_{i+1}(m_0), \dots, e_n(m_0)\}$. He then challenges the oracle with cryptoset ψ_0 . If the oracle responds with 0 (meaning that the cryptoset is a valid cryptoset for m_0), then c is an encryption of m_0 . Otherwise, it is an encryption of m_1 . The adversary responds to a challenge and wins.

Winning this challenge is equivalent to encryption $e_i(\cdot)$ not being semantically secure, which contradicts the givens. Hence, a cryptoset is semantically secure. $\square$

In the context of current practical implementation of the model it means that a cryptoset with Paillier and ElGamal encryptions is semantically secure.

As soon as cryptographic transformations that facilitate search are introduced into a cryptoset, it ceases to have semantic security: it is easy to see that semantic security completely prohibits an ability to do search. On the other hand, as discussed earlier, not all cryptosets necessarily have all the components. Those that are used exclusively with semantically secure encryptions will maintain the property.

6.2 Operations Modules

6.2.1 Arithmetic Operations

Fully or Almost Fully Homomorphic Encryption Schemes

Executing arbitrary or a wide set of arithmetic operations over encrypted data would be a relatively simple task if there was a fully homomorphic encryption scheme ready to be used. There are two cryptosystems that can be considered fully or almost fully homomorphic.

First one is a scheme proposed by Craig Gentry in 2009 [32]. It is proved to be fully homomorphic, i.e., it supports *arbitrary* arithmetic expressions with multiplication and addition operations. However, it appeared to be too resource consuming to be used in practice. Later there was an attempt to make it more practical and some results were achieved, but still it was distant from being ready for practical use [33]. A simplified version of Gentry’s scheme, so-called “a somewhat-homomorphic encryption” was used in the context of an encrypted database, but since it lacks fully homomorphic properties it is unable to perform arbitrary operations [12].

Another one is a Boneh-Goh-Nissim (BGN) cryptosystem, which supports any number of encrypted additions and no more than one multiplication [13]. There are certain notable disadvantages to the BGN scheme as compared to other homomorphic encryption schemes (e.g., ElGamal, Paillier, etc.). First, the encryption process in BGN is significantly slower, which makes using this scheme a questionable approach in a DBMS, where a typical workflow would require encrypting large amounts of short messages. Second, the decryption process in BGN is done using Pollard’s lambda method (also known as Pollard’s kangaroo method), which is basically a guessing game. It takes a mean time of $O(\sqrt{T})$ for a message $1 \leq m \leq T$ but this time is

not capped with any reasonable upper bound [13, 50], which means that theoretically decryption could take any amount of time, which is hardly acceptable for a DBMS.

As of now there are no known practically usable fully homomorphic encryption schemes, which makes us to rely on a less straightforward approach.

Multiplication

There are not so many multiplication-homomorphic encryption schemes. Specifically, among the well-known there are three: RSA, ElGamal, and Boneh-Goh-Nissim (BGN). The latter was already considered above.

The RSA cryptosystem was proven to be insecure (not IND-CPA secure, to be exact) unless the plaintext messages are randomly padded prior to encryption, and the padded version of RSA loses its homomorphic properties [41, p. 33]. Thus, the RSA cryptosystem was rejected as a candidate for the implementation of the encrypted multiplication module.

This only leaves us with the ElGamal cryptosystem to use as a basis for encrypted multiplication module. However, there exists an attack on the ElGamal cryptosystem that could lead to the sensitive information being leaked if the cryptosystem is used with a 512 bits or less key [4]. Therefore, according to the widely accepted practice, a key at least twice as long should be used, which brings us to a key length of 1024 bits.

ElGamal scheme by design can not produce a ciphertext that is shorter than twice the key length, which in this case adds up to 2048 bits of the resulting ciphertext. Since the ElGamal encryption is being used for its homomorphic properties, the plaintext values that are going to be encrypted using this scheme are integers, which are typically 4 bytes long. Thus, message expansion is 64 times. It is quite a significant ratio, but at least it is linearly related to the overall size of plaintext data.

Addition

Unlike multiplication, there is a number of addition-homomorphic cryptosystems. Among the relatively well-known are these: Benaloh, Naccache-Stern, Sander-Young-Yung, Okamoto-Uchiyama and its modifications, Paillier and its modifications, Schmidt-Samoa-Takagi, Damgård-Jurik and its modifications, and Boneh-Goh-Nissim.

The Paillier cryptosystem is one of the most investigated addition-homomorphic cryptosystems, and has one of the lesser message expansion ratios. Encrypted addition module was thus based on the Paillier cryptosystem, and the key was chosen to be 1024 bit. Like the ElGamal cryptosystem, the ciphertext block size is directly related

Figure 6.3: System architecture for equality matching scheme

to the key length in the Paillier scheme and is twice as long as the key, i.e., for a 1024 bit key a message is encrypted into a 2048 bits long ciphertext. Thus, message expansion for the Paillier scheme is 64 times as well, if the message is a 4-byte integer.

6.2.2 Equality Matching and Search

To introduce the equality matching technique, let us assume that a database consists of two tables A and B with attributes A_i and B_i , $i \in \{1, 2, \dots\}$ respectively. Each cell of each table keeps a cryptoset $T_{x,y}^*$, where x is a row number, y is an attribute index, and $*$ is either A or B . The model is seamlessly extensible to an arbitrary number of tables with arbitrary number of attributes.

From a generic point of view, the idea of the proposed scheme is that one of the elements of a cryptoset is a *fingerprint* (a hash-like structure in the form of a bit array) of a plaintext value. When a querier needs to do matching with a constant value or with another attribute, he simply matches their fingerprints. If they are not equal then the underlying plaintexts are definitely not equal too. If they match, then the underlying plaintexts are *probably* equal. This means that upon retrieval the response contains some amount of false positives that are to be filtered at the client side (see below for false positive rate estimations), which is aligned with the Assumption 2 that was given in the Chapter 4. However, the design offers a flexible per-column security adjustments, which still keeps it possible to do column-column match.

Components of the System

Let us introduce a specific system layout for the purposes of defining this model. It translates well onto the model architecture we have proposed for the whole system.

Key Manager KM generates and allocates keys to users and proxies so they are able to follow the protocols needed for the system to operate. Most of the time

KM could and should be kept offline. It is needed online only during the system setup and then every time a new user is enrolled in the system.

Data owner Data owner knows which information is more sensitive and which is less sensitive; thus, it sets the data sensitivity level $m_i^* \leq m$ for every column in every table. This information is public.

Proxies There is a set of $z \geq 1$ proxies. Each proxy keeps a share of secret keys s and q ; the shares are individual for each user. For the i^{th} user the j^{th} proxy keeps $s_{i,j}^P$ and $q_{i,j}^P$.

Hashing Server This is a special type of proxy. Its purpose is to compute data fingerprints both on stages of *data encryption* and *query construction*.

Cloud Database Server Same as defined in Chapter 3.

User Same as defined in Chapter 3.

Trust and Attack Model

The Key Manager is assumed to be fully trusted. The database server, proxies, and the hashing server are modeled as semi-honest: they obey instructions and follow the protocols but are passively curious—they try to learn as much information as possible about the data stored in the database as well as user queries without initiating unauthorized actions. In addition, when users leave the system, they are not expected to either forget or keep their keys in secret, i.e., users with revoked access are considered potentially malicious.

System Construction

System Setup The KM sets up a cyclic group G of large prime order p ; w is the generator of G . KM also chooses a random master storage key s and a query key q . Additionally, KM chooses *fingerprint* length of m bits (could be any number; 16, 32 or 64 bits are good candidates as they are easier to store) and selects $m/2$ different cryptographic hashing functions $f_1(x), \dots, f_{m/2}(x)$. Every hashing function $f_i(x)$ has an image $\{1, 2, \dots, m - i + 1\}$. For example, if $m = 16$, function $f_1(x)$ has an image $\{1, 2, \dots, 16\}$, $f_2(x)$ — $\{1, 2, \dots, 15\}$, and so on. KM publicizes (G, p, w, m) . Hashing functions $f_1(x), \dots, f_{m/2}(x)$ are to be transferred in a private manner to the hashing server. Keys s and q are kept secret by the KM.

Data Model Annotation Before the first user enrolls in the database, the data owner is expected to annotate the data scheme.

In addition to the tables A and B , the DB also stores a public unencrypted service table S containing a data sensitivity level $m_i^* \in \{1, 2, \dots, m/2\}$ for every attribute.

Sensitivity levels are denoted in the following manner: attribute B_3 has a sensitivity level m_3^B , attribute A_1 —a sensitivity level m_1^A . Data sensitivity level refers to the amount of different hashing functions used to generate fingerprints (which are inspired by Bloom filters). The most sensitive data has sensitivity level 1; the least sensitive data has sensitivity level $m/2$.

User Enrollment When the system grants access to a new user U_i , KM generates shares s_i and q_i of secret keys s and q respectively, for each user and proxy. These shares are then privately transferred to respective parties. User U_i receives secret shares s_i^U and q_i^U . For every user enrolled in the system each proxy keeps corresponding shares of the keys. For every user U_i every proxy P_j keeps $s_{i,j}^P$ and $q_{i,j}^P$, and the following holds for every user U_i :

$$\begin{aligned} s &= s_i^U + s_{i,1}^P + \cdots + s_{i,z}^P = s_i^U + \sum_{j=1}^z s_{i,j}^P \\ q &= q_i^U + q_{i,1}^P + \cdots + q_{i,z}^P = q_i^U + \sum_{j=1}^z q_{i,j}^P \end{aligned}$$

Data Encryption As it was stated earlier, along with the encrypted value, the database stores a *fingerprint* of the value. So for every plaintext value $T_{x,y}^*$ the database stores a cryptoset $\langle T_{x,y}^* \langle 1 \rangle, T_{x,y}^* \langle 2 \rangle \rangle$, where $T_{x,y}^* \langle 1 \rangle$ is a ciphertext of the original value and $T_{x,y}^* \langle 2 \rangle$ is its fingerprint.

To compute $T_{x,y}^* \langle 1 \rangle$ user U_i chooses a random number $r \in Z_p^*$ and computes $c_1 = T_{x,y}^* \times w^{rs_i^U}$ and $c_2 = w^r$. He sends $\langle c_1, c_2 \rangle$ to the proxy P_1 . Upon receiving this pair of values, proxy P_j computes $c_2' = c_2^{s_{i,j}^P}$, then computes $c_1' = c_1 \times c_2'$ and finally sends $\langle c_1', c_2 \rangle$ to the next proxy in chain, e.g., P_1 sends to P_2 . The next proxy in sequence does the same. The last proxy in the chain P_z sends the resulting $\langle c_1', c_2 \rangle$ (where $c_1' = T_{x,y}^* \times w^{rs_i^U} \times c_2^{\sum_{j=1}^z s_{i,j}^P} = T_{x,y}^* \times w^{rs_i^U} \times w^{r \sum_{j=1}^z s_{i,j}^P} = T_{x,y}^* \times w^{rs}$) to the database server, which then stores this pair as $T_{x,y}^* \langle 1 \rangle = \langle c_1', c_2 \rangle = \langle T_{x,y}^* \times w^{rs}, w^r \rangle$. This is basically the ElGamal scheme modified to be computed in several steps.

The fingerprint $T_{x,y}^* \langle 2 \rangle$ of the value, which will be later used to perform value look-ups and relational joins, is constructed in a similar pipeline that starts with the user, goes through all the proxies, and ends at the Hashing Server. First, the user computes $b = T_{x,y}^* \times w^{q_i^U}$ (note the lack of the random parameter r). He then passes b to proxy P_1 that computes $b_1 = b \times w^{q_{i,1}^P} = T_{x,y}^* \times w^{q_i^U + q_{i,1}^P}$. Proxy P_1 then transfers b_1 to proxy P_2 , which computes $b_2 = b_1 \times w^{q_{i,2}^P}$, and so on until the last proxy P_z finally computes $b_z = b_{z-1} \times w^{q_{i,z}^P} = T_{x,y}^* \times w^q$. This value b_z is then transferred to

Figure 6.4: Encryption workflow

the Hashing Server.

After this point, the construction of a fingerprint is somewhat similar to the creation of a Bloom filter [34, page 5], [37, page 7]. First, the Hashing Server refers to the service table S in the database and retrieves a corresponding data sensitivity level m_y^* . Then it creates a bit array R of length m and initializes it with zeros. After that, $f_1(b_z), \dots, f_{m_y^*}(b_z)$ are computed and the set of resulting values is used as indices of values in the array R that should be replaced with 1's; i.e., the system performs $R[f_1(b_z)] := 1, \dots, R[f_{m_y^*}(b_z)] := 1$, where $R[i]$ is a reference to i^{th} element of the array R , and “ $:=$ ” is an assignment operator. Some indices could coincide, i.e., $f_i(b_z) = f_j(b_z)$, where $i < j$. This means that $R[f_j(b_z)]$ is already set to 1. If this is the case, the system increments the latter index $f_j(b_z)$ until it finds a bit in R that is still equal to 0, and sets it to 1.

Eventually a bit array R of length m with 1's on certain slots is obtained. This array is a *fingerprint* of the original value, which is then returned to the user, who sends it to the cloud database to be stored as $T_{x,y}^* \langle 2 \rangle$.

Query Construction

Column–Constant Matching The first type of queries to be supported is (e.g., for table A): `SELECT columns FROM A WHERE  $A_y = v$` , where v is a constant value. To transform the query, the system needs to generate a fingerprint R_v for the value v . The sensitivity level used in the construction should be the sensitivity level set for the queried attribute (A_y in this example). The original query is transformed into `SELECT columns FROM A WHERE  $T_y'^A \langle 2 \rangle = R_v$` .

The mechanism of fingerprints guarantees that if $T_{x,y}^* = v$ then $T_{x,y}^* \langle 2 \rangle = R_v$. On

the other hand, this does not work backwards, i.e., if $T'_{x,y} \langle 2 \rangle = R_v$ it does not mean that $T_{x,y}^* = v$. This helps to protect the information about the cells with the same original plaintext values but enlarges data transfer overhead since extra tuples are transferred as the query result to the client, who then needs to filter out the false positives. False positives rate could be adjusted with data sensitivity level m_y^* —the lower it is, the larger false positives rate is, but information about coinciding data is more secure, and vice versa.

Column–Column Matching Another type of supported queries involves matching values of two different columns (attributes) of the same table A (or B): `SELECT columns FROM A WHERE  $A_{y_1} = A_{y_2}$` . Two possible scenarios are considered: **1)** attributes A_{y_1} and A_{y_2} have same sensitivity levels $m_{y_1}^A = m_{y_2}^A$; **2)** they have different sensitivity levels $m_{y_1}^A \neq m_{y_2}^A$.

Case of $m_{y_1}^A = m_{y_2}^A$ This scenario is very straightforward and is similar to matching with constant value. It suffices to rewrite the original query in the following way: `SELECT columns FROM A WHERE  $T'_{y_1} \langle 2 \rangle = T'_{y_2} \langle 2 \rangle$` . An additional benefit of the proposed scheme is that in the case of matching two columns with equal sensitivity levels, and matching a column with a constant, the system may utilize the built-in indexing capabilities of the database system.

Case of $m_{y_1}^A \neq m_{y_2}^A$ This scenario is slightly more complicated, since fingerprints for the two columns that the querier is trying to match are built with different sets of hashing functions. To be specific, let us assume that $m_{y_1}^A > m_{y_2}^A$. To construct a fingerprint $T'_{x,y_1} \langle 2 \rangle$ for A_{y_1} hashing functions $f_1(x), \dots, f_{m_{y_1}^A}(x)$ are used; to construct a fingerprint $T'_{x,y_2} \langle 2 \rangle$ for A_{y_2} hashing functions $f_1(x), \dots, f_{m_{y_2}^A}(x)$ are used. Therefore, hashing functions $f_1(x), \dots, f_{m_{y_2}^A}(x)$ are used in the construction of both fingerprints, but $T'_{x,y_1} \langle 2 \rangle$ also used several more functions. This means that if there is a “0” in $T'_{x,y_1} \langle 2 \rangle$ and a “1” in $T'_{x,y_2} \langle 2 \rangle$ on the same position, then it is definite that the original plaintext values of T_{x,y_1}^A and T_{x,y_2}^A are not equal. Otherwise, they *could* match and the database server should return this tuple in a result set.

Hence the original query should be rewritten in the following way: `SELECT columns FROM A WHERE (  $T'_{x,y_1} \langle 2 \rangle \& T'_{x,y_2} \langle 2 \rangle$  ) =  $T'_{x,y_2} \langle 2 \rangle$` , where “&” denotes a bitwise AND operation.

The proposed scheme is rather flexible since it allows database manager to fine-tune the security–performance trade-off for every single column and still leave users the ability to perform matches between these columns. The only issue here is the

inability to rely on the built-in database indexing mechanisms in case $m_{y_1}^A \neq m_{y_2}^A$, since the **WHERE** predicate contains a bitwise operation, so the tuning should be performed with that in mind. It might appear that lowering sensitivity level for a specific column damages the performance more than improves it under the given workload profile.

Equi-Joins Equi-join is a query of the type **SELECT** *columns* **FROM** A, B **WHERE** $A_{y_1} = B_{y_2}$. Equi-join is in fact an inter-table column-column marching. It is interesting to note that the method of column-column matching does not require the columns to be in the same table, and thus makes it possible to perform equi-joins without additional efforts.

More Complex Queries The proposed model of query execution allows one to perform queries with complex predicates that combine all previously mentioned types of predicates (column-constant matching, inter-table column-column matching, and intra-table column-column matching) in arbitrary boolean expressions using **AND**, **OR**, and **NOT** operators.

Result Processing and Decryption

Let us assume that the result of user U_i 's query is a set of n records (tuples) $J = \{J_{i_1}, J_{i_2}, \dots, J_{i_n}\}$, each of them consists of c ciphertexts $T_{x,y}'^* \langle 1 \rangle = (T_{x,y}^* \times w^{r_{x,y}s}, w^{r_{x,y}})$, where $x \in \{i_1, \dots, i_n\}$, $y \in \{j_1, \dots, j_c\}$. DBMS sends this result back to user U_i through a reversed chain of proxies, from P_z to P_1 . Each proxy P_j in the chain computes $t_j = w^{r_{x,y}s_{i,j}^P}$, multiplies the encrypted value by t_j^{-1} , and sends the result of the multiplication to the next proxy in the chain, who proceeds to do the same with the received value, and so on. The last proxy in the chain P_1 sends the result to the user. The value received by the user is in fact $T_{x,y}^* \times w^{r_{x,y}s_i^U}$. The user then multiplies the received value by $(w^{r_{x,y}s_i^U})^{-1}$ and obtains a plaintext value $T_{x,y}^*$.

After that, when the user has decrypted ciphertexts and knows plaintext values he needs to do the final filtering and remove from the query results those records, which do not actually satisfy the **WHERE** predicate in the query (the false positives).

Scheme Analysis

$T \langle 2 \rangle$ is used to search certain values among data in the database or to match columns with each other without revealing to anyone (should it even be a database administrator) which exact cells of the database contain equal values. The database administrator could adjust data sensitivity level from 1 to $m/2$, where 1 corresponds to the most sensitive data, and $m/2$ —to the least sensitive data.

The fingerprint construction is equivalent to pseudo-random selection of m_i different bits to be set to 1 in the zero-filled array R . As this process suggests, for a sensitivity level $m_i \in \{1, 2, \dots, m/2\}$ the whole domain of a value is uniformly divided into $\binom{m}{m_i}$ buckets¹. This means that for a value domain containing d values (e.g., for a 32-bit integer, $d = 2^{32} = 4\,294\,967\,296$), the amount of values in each bucket is

$$v = \frac{d}{\binom{m}{m_i}}$$

For example, for a 32-bit value in a column with sensitivity $m_i = 1$ the size of a bucket is $v \approx 2.7 \times 10^7$, and in a column with sensitivity $m_i = 8$ the size of a bucket is $v \approx 3.3 \times 10^5$ (m is assumed to be 16).

As it could be easily seen, during the construction of a fingerprint (which in fact appears to be a hash of a ciphertext of an original plaintext value, which is obtained using the query key q) almost all information about the original value is destroyed and an adversary is incapable of reconstructing the original value from it. Moreover, the constructed fingerprint is generally a *good* hashing function as defined by RFC4949 [64, page 139]. This means that two values having the same fingerprint tells adversary nothing about the original values. Certainly this depends on the selected hashing functions $f_i(x)$ in every particular case, and any specific set of hashing functions $f_i(x)$ should be formally proved to deliver a good fingerprint before using.

Other Considerations

User Revocation As soon as a user is not able to query the data, correctly encrypt data for updates or inserts without a help from proxies, he might be considered as having his access revoked. Revocation of user U_i is done by the KM simply instructing all proxies P_j to remove their corresponding key shares $s_{i,j}^P$ and $q_{i,j}^P$. Even if the user were to gain access to the database, he would not be able to generate a meaningful query.

Key renewal Although compromising a key is much harder as it requires collusion of all parties, the possibility of this happening cannot be neglected. For example, the storage and query master keys s and q may be theoretically obtained directly from KM by an adversary. Hence, both these keys should be updated from time to time. The procedure for updating a storage key s may be performed in the database without the need to decrypt data using the proxy re-encryption scheme, similar to the

¹ $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

ones proposed by some previous works [27, 62]. An applicable procedure is described in details in the work by Hung *et al.* [44, Section VI.C].

The scheme is not capable of changing a query key q , when it is compromised, as efficiently as it can change the storage key s . The only way to change the query key q is to generate new fingerprints for all data; however, this can be quite time-consuming for large databases. On the other hand, a leaked query key q gives much less information to the adversary than a leaked storage key s . In fact, the query key itself would not reveal anything to the adversary. In case the adversary also knows all hashing functions $f_1(x), \dots, f_{m/2}(x)$ (or has access to the hashing server), he is able to compute sets of plaintext values corresponding to a certain fingerprint and, consequently, to all cells having this fingerprint (taking into account the sensitivity level).

Apart from the query key, the hashing functions $f_1(x), \dots, f_{m/2}(x)$ should also be changed from time to time. The process of changing the hashing functions does not differ much from the process of a query key change, and implies a re-computation of all fingerprints.

It is possible that a practical implementation could partly alleviate the process of the query key renewal by changing the query key for every column separately, one at a time (in a transaction), starting with those keeping the most sensitive data. During the procedure of changing the query key, the system should support both new and old query keys for those columns that are already updated and those which are still waiting in a queue, respectively. The same procedure may be used for the hashing functions renewal. It would save a lot of time if the query key renewal and the hashing functions renewal are combined and performed as a single task.

6.3 Summary

In this chapter an implementation of the model is discussed. The implementation is proposed under the assumption that the encrypted database is required to be able to execute such operations as equality matching, numeric addition and numeric multiplication.

First part of the chapter is dedicated to one of the issues of implementing the query execution model in practice: data storage. The section considers some of the possible approaches to storing cryptosets in a real database's relations and attributes, and justifies the choice of the single table approach to storing the cryptosets. The single table approach is then considered in more details, showing how specific traits

of the query execution model are relayed in the storage model.

The storage model is analyzed for the data expansion factor and certain optimizations are suggested in order to reduce the factor. It is also analyzed for security properties in the context of the cryptosystems that are used in the implementation—Paillier and ElGamal.

Later in the chapter use of these two homomorphic encryption schemes is discussed in the context of implementing encrypted arithmetic operations modules. Lastly, a new flexible encrypted search technique is proposed, which was created specifically for use in an encrypted DBMS. It is considered in details and theoretically analyzed.

Chapter 7

Empirical Results

This chapter discusses what was achieved in the direction of creating a practical implementation of the proposed encrypted database system.

Choosing a set of tools for the implementation required taking into account a large number of factors. First, as the presented work is investigating the wrapper approach, the underlying DBMS needs to be chosen. Typical choices for research projects are MySQL and PostgreSQL due to their open-source nature and a seemingly closer integration with the scientific community. On the other hand, this work explores issues that mainly arise in the enterprise context, where such DBMSs as Oracle and MS SQL Server have earned their dominance. Second, choosing a language to implement the system and its parts is not only a question of preference, but also might have a significant effect on the performance, development time, and also might affect or depend on the choice of DBMS. Among a vast amount of possible options, having a requirement that the language is well-performing computationally and established enough to be considered reliable, the qualified candidates were C, C++, C#, and Java.

Due to certain properties of the DBMSs, good combinations would have been MySQL or PostgreSQL and C or C++; Oracle and Java; MS SQL Server and C#. While some experiments were conducted using MySQL, most of the empirical results were obtained using MS SQL Server and C#, which were chosen for the following properties:

1. C# is a modern, well-performing, and reliable language; it inherits many good features of Java, such as platform independence and a strong object-oriented model; as a newer language, it is less bound by the legacy, and was able to adopt newer and better practices in many areas. At the same time it is a higher-level

- language than C or C++, which significantly reduces the development time.
2. MS SQL Server is a widely used in enterprise context DBMS.
 3. C# and .NET are deeply integrated into MS SQL Server, which makes it possible to create UDFs in any of the .NET languages, including C#.

As a basis for many components of the system a structure that implements a big integer is required. The .NET framework itself provides a BigInteger implementation; also there are several external implementations, such as the Bouncy Castle¹ BigInteger. After some research on the topic, we have created our own implementation of BigInteger, which was less generalized and thus was significantly more efficient, as the tests have shown. The implementation of BigInteger was thoroughly tested for correctness by a comprehensive automatic test suite with full code coverage.

7.1 Arithmetic Operations

Both cryptosystems that were chosen to provide homomorphic operations for the database appeared to be lacking a publicly available implementation, which led to the necessity to implement them for this project. Both cryptosystems were carefully implemented following the definition, and tested for both correctness and performance. However, the main focus was on correctness rather than performance optimizations, so it is possible that the results could be significantly improved.

7.1.1 Multiplication

ElGamal cryptosystem was implemented as an extension for the cryptographic module of the .NET framework, and was using our BigInteger implementation as a foundation.

The implementation was thoroughly tested for correctness. The implemented cryptosystem is capable of encrypting both textual and numeric data. To check correctness of text encryption, Algorithm 2 was used. Key lengths were taken from the set {384, 512, 640, 768, 896, 1024}. Correctness check was conducted in two stages.

For the first stage, text lengths were taken from the set {0, 1, 2, 10, 16, 47, 48, 63, 64, 79, 80, 95, 96, 111, 112, 127, 128, 512, 2500}; the algorithm was called 1 000 times for each combination of key length and text length, in order to have a wider coverage of tests due to randomness of text and keys in each iteration.

For the second stage, for each key length, the text length was generated at random in the range [0; 2500], and algorithm was called with these parameters. Each key length was tested with 500 random text lengths.

¹Bouncy Castle is an established cryptographic API for Java and C#.

Figure 7.1: ElGamal homomorphic multiplication performance drop ratio

First stage was required to specifically test the implementation with edge cases: 0-length text, short text, power of 2-length text, text with a length that *is* a multiple of a block size, text with a length that is *not* a multiple of a block size, long text. For text encryption the encryption algorithm was set to “trailing zeros” padding mode.

Algorithm 2 Correctness test for ElGamal text encryption

Require: key length k (bits), text length n (characters)
 $\text{key} \leftarrow \text{GENERATEKEY}(k)$
 $\text{text} \leftarrow \text{random sequence of } n \text{ ASCII characters}$

 $\text{bytes} \leftarrow \text{ENCODING.ASCII.GETBYTES}(\text{text})$
 $\text{ciphertext} \leftarrow \text{ELGAMAL.ENCRYPT}(\text{bytes}, \text{key})$
 $\text{newBytes} \leftarrow \text{ELGAMAL.DECRYPT}(\text{ciphertext}, \text{key})$
 $\text{newText} \leftarrow \text{ENCODING.ASCII.GETSTRING}(\text{newBytes})$

 $\text{ASSERTEQUAL}(\text{text}, \text{newText})$

To check correctness of numeric encryption and correctness of homomorphic multiplication, Algorithm 3 was used. In the first stage, for each key length from the set $\{384, 512, 640, 768, 896, 1024\}$, the algorithm was called 3 000 times. In the second stage, for each key length, the algorithm was supplied with specific pairs of value_1 and value_2 from the set $\{0, 1, 2\} \times \{0, 1, 2\}$ to test for edge cases of multiplication—multiplication by 0 and multiplication by 1. Each such combination was iterated 500 times to account for different keys, which were generated at random for each iteration. For numeric encryption, the padding mode of the encryption algorithm was set to “leading zeros”.

Since the homomorphic multiplication operation is intended to be used in lieu of the plaintext operation, it is reasonable to estimate the performance drop caused by

Algorithm 3 Correctness test for ElGamal numeric encryption and homomorphic multiplication

Require: key length k (bits)

key $\leftarrow$ GENERATEKEY(k)

value₁, value₂ $\leftarrow$ random unsigned integers

ciphertext₁ $\leftarrow$ ELGAMAL.ENCRYPT(value₁, key)

ciphertext₂ $\leftarrow$ ELGAMAL.ENCRYPT(value₂, key)

cipherMult $\leftarrow$ ELGAMAL.MULTIPLY(ciphertext₁, ciphertext₂)

decValue₁ $\leftarrow$ ELGAMAL.DECRYPT(ciphertext₁, key)

decValue₂ $\leftarrow$ ELGAMAL.DECRYPT(ciphertext₂, key)

decValueMult $\leftarrow$ ELGAMAL.DECRYPT(cipherMult, key)

ASERTEQUAL(value₁, decValue₁)

ASERTEQUAL(value₂, decValue₂)

ASERTEQUAL(value₁ $\times$ value₂, decValueMult)

that. In order to do so, the tests were carried out with both the plaintext operation and the homomorphic operation, and the ratio of the resulting total execution times was used as an estimate for the performance drop.

Tests were conducted in 12 identical stages, each stage consisted of 250 000 iterations. Each iteration consisted of 1 execution of the operation. Tests were run with a varying key length from the same set {384, 512, 640, 768, 896, 1024}. So, the resulting number for each key length is basically a total time of executing $12 \times 250000 = 3 \cdot 10^6$ homomorphic multiplications of random encrypted numbers divided by the total time of execution of $3 \cdot 10^6$ plaintext multiplications of random numbers. The results of the performance benchmarking are presented in Figure 7.1. As it is seen in the picture, homomorphic multiplication with a 1024 bit key is approximately 128 000 times slower than plaintext operation.

7.1.2 Addition

Paillier cryptosystem was implemented as an extension of the .NET cryptographic subsystem and was based upon our implementation of the BigInteger class.

The implementation of the cryptosystem was extensively tested for correctness. Similar to ElGamal, the Paillier encryption implementation is capable of encrypting both textual and numeric data. To check correctness of textual encryption, Algorithm 4 was used. Correctness check was conducted in two stages, for both stages the set of key lengths was {384, 512, 640, 768, 896, 1024}.

For the first stage, algorithm was supplied with text lengths from the set {0, 1, 2, 10, 16, 47, 48, 63, 64, 79, 80, 95, 96, 111, 112, 127, 128, 512, 2500} in order to ensure tests

Figure 7.2: Paillier homomorphic addition performance drop ratio

for specific edge cases—empty and short texts, long texts, text lengths equal and not equal to the multiples of block sizes. For each combination of key and text length, the algorithm was called 1 000 times to have a sufficient distribution of random keys and texts.

For the second stage, for each key length from the set, 500 text lengths were chosen at random from range $[0; 250]$, and the testing algorithm was called with these parameters. Text encryption with the Paillier algorithm was done with “trailing zeros” padding mode.

Algorithm 4 Correctness test for Paillier text encryption

Require: key length k (bits), text length n (characters)

```

key ← GENERATEKEY( $k$ )
text ← random sequence of  $n$  ASCII characters

bytes ← ENCODING.ASCII.GETBYTES(text)
ciphertext ← PAILLIER.ENCRIPT(bytes, key)
newBytes ← PAILLIER.DECRYPT(ciphertext, key)
newText ← ENCODING.ASCII.GETSTRING(newBytes)

ASSERTEQUAL(text, newText)

```

To check correctness of numeric encryption and correctness of homomorphic addition in the Paillier implementation, Algorithm 5 was used. In the first stage, for each key length, the algorithm was called 3 000 times.

In the second stage, for each key length, the algorithm was supplied with specific pairs of $value_1$ and $value_2$ from the set $\{0, 1\} \times \{0, 1\}$ to test specifically for the edge case of adding a 0, which should not change the result. Each such combination was iterated 500 times to account for different random keys. For numeric encryption, the

padding mode of the encryption algorithm was set to “leading zeros”.

Algorithm 5 Correctness test for Paillier numeric encryption and homomorphic addition

Require: key length k (bits)

key $\leftarrow$ GENERATEKEY(k)

value₁, value₂ $\leftarrow$ random unsigned integers

ciphertext₁ $\leftarrow$ PAILLIER.ENCRIPT(value₁, key)

ciphertext₂ $\leftarrow$ PAILLIER.ENCRIPT(value₂, key)

cipherAdd $\leftarrow$ PAILLIER.ADD(ciphertext₁, ciphertext₂)

decValue₁ $\leftarrow$ PAILLIER.DECRYPT(ciphertext₁, key)

decValue₂ $\leftarrow$ PAILLIER.DECRYPT(ciphertext₂, key)

decValueAdd $\leftarrow$ PAILLIER.DECRYPT(cipherAdd, key)

ASSERTEQUAL(value₁, decValue₁)

ASSERTEQUAL(value₂, decValue₂)

ASSERTEQUAL(value₁ + value₂, decValueAdd)

Implementation of the Paillier cryptosystem was also tested for performance of the homomorphic addition as compared to plaintext addition. Similarly to how the ElGamal cryptosystem was tested, performance tests for the Paillier cryptosystem were conducted in 12 stages, each containing 250 000 iterations, each of which was an execution of a single operation of addition—encrypted or plaintext. The results, which are a ratio of total execution times of encrypted and plaintext performance measurements, are presented in Figure 7.2. With a key length of 1024 bit, as it is seen on the chart, the homomorphic operation of addition appears to be approximately 224 000 times slower than the plaintext one.

7.2 Equality Matching and Search

As the focus of this model is on the construction and usage of fingerprints to query data, performance was evaluated based on different scenarios of data retrieval, leaving encryption and decryption out of scope. The proposed model was implemented and various performance tests were conducted. The implementation consists of 3 parts: a Proxy Server, a Hashing Server, and a program that emulates the user activity, all of which are communicating through a network. MySQL v5.6 was used as the underlying database system for the tests.

Fingerprint length m was set to 16 bit. Hashing functions were constructed using HMAC based on MD5. They produce a 32 digit hexadecimal number, whose remainders from division by $m, (m - 1), \dots, 1$ are then used as indices of the bits to be set to 1 in the resulting fingerprint.

The database consisted of tables `t1` and `t2`. Table `t1` consisted of 4 columns `a_1 INT`, `a_2 SMALLINT`, `b_1 INT`, `b_2 SMALLINT`. Table `t2` consisted of 4 columns `c_1 INT`, `c_2 SMALLINT`, `d_1 INT`, `d_2 SMALLINT`. The index was built for columns `t1.a_1`, `t1.a_2`, `t2.c_1` and `t2.c_2`. Columns which have “_1” in their names contain original plaintext values; columns which have “_2” contain fingerprints and represent $T \langle 2 \rangle$.

Tests were run in several stages, each with different sets of sensitivity values for each column. Each table was populated with 30 000 rows of uniformly random integer values from $\{0, 1, \dots, 500\}$ and their fingerprints were created with respect to the current sensitivity setting. Each test stage consisted of the following tests:

1. Constant matching on a column with an index (`t1.a`):
`SELECT SQL_NO_CACHE2 COUNT(*) FROM t1 WHERE t1.a = <const>`
2. Constant matching on a column without an index (`t1.b`):
`SELECT SQL_NO_CACHE COUNT(*) FROM t1 WHERE t1.b = <const>`
3. Intra-table column matching (i.e., `t1.a = t1.b`):
`SELECT SQL_NO_CACHE COUNT(*) FROM t1 WHERE t1.a = t1.b`
4. Inter-table column matching (join) on two columns with an index (`t1.a = t2.c`):
`SELECT SQL_NO_CACHE COUNT(*) FROM t1,t2 WHERE t1.a = t2.c`
5. Inter-table column matching (join) on two columns, only one of which has an index (`t1.a = t2.d`):
`SELECT SQL_NO_CACHE COUNT(*) FROM t1,t2 WHERE t1.a = t2.d`
6. Inter-table column matching (join) on two columns, none of which has an index (`t1.b = t2.d`):
`SELECT SQL_NO_CACHE COUNT(*) FROM t1,t2 WHERE t1.b = t2.d`

Each test was executed repeatedly: tests 1 and 2—200 times, tests 3–6—50 times for the first 8 test stages, and only tests 4–6 were run for the stages 9–15, each was repeated 25 times. Test stages differentiated in sensitivity values set for columns, which values are shown in Table 7.1.

Tests were conducted in order to estimate two performance metrics: the rate of false positives (it was estimated as an amount of *all* rows returned by a secure query divided by the amount of rows which *should* have been returned, i.e., returned by plaintext version of the query—as it was mentioned earlier, client should filter out those rows that actually do not comply with `WHERE` predicate); and query execution

²Query results caching disabled by specifying `SQL_NO_CACHE`. This is MySQL-specific.

Test stage	Sensitivity values			
	t1.a	t1.b	t2.c	t2.d
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
⋮	⋮	⋮	⋮	⋮
8	8	8	8	8
9	1	1	8	8
10	2	2	8	8
11	3	3	8	8
⋮	⋮	⋮	⋮	⋮
15	7	7	8	8

Table 7.1: System configuration sets used for search benchmarking

time as compared to executing same queries on plaintext values, which were obtained as a ratio of execution times of secure and plaintext queries. Results are calculated as an average among several repetitions of the test and are shown in Figure 7.3.

The charts show that, as expected, the amount of false positives decreases with the increase in the number of different hashing functions used (which corresponds to lowering data privacy level for the column) and eventually becomes negligible.

For query execution time, for test stages 1–8 we observe that in most cases, query execution time is relatively the same for both plaintext and secure versions. Exceptions are the test `t1.a = const` in stage 1, where the secure version is about 3 times slower; and queries with joins `t1.a = t2.c` and `t1.a = t2.d` on the first several stages. All the cases involve one indexed column `t1.a` (in a plaintext query it is `t1.a.1`, and in a secure query it is `t1.a.2`), so longer execution time is most likely caused by a much larger amount of returned rows (especially for joins, where there are up to 60 times more rows to return). In cases where indexed columns are not involved at all, the query execution times are virtually the same for all stages.

For test stages 6–10 we observe that tests involving indexed column query execution time is constantly much longer. Clearly, this is caused by the fact that when sensitivity levels are different for two columns involved in matching in the `WHERE` predicate, this predicate is transformed to use bitwise operations, which effectively blocks the database from using index in any way. For tests with no indexed columns the execution time is relatively the same for secure and plaintext queries; the small difference is caused by the data transfer overhead and by the slightly more complex `WHERE` predicates in the secure version.

(a) Stages 1–8, False positives ratio

(b) Stages 1–8, Time ratio

(c) Stages 9–15, False positives ratio

(d) Stages 9–15, Time ratio

Figure 7.3: Performance testing results

7.3 Full System

An implementation of an encrypted database system was written using a wrapper approach with the MS SQL Sever as an underlying DBMS, using C# and .NET framework. The architecture of the system closely follows the architecture of the model described in this work. The implementation was designed with the idea of modularity in mind, so adding or modifying specific encrypted operations is done relatively hassle-free.

As for the moment of this work being written, the implementation supports homomorphic addition based on Paillier cryptosystem and equality matching based on the fingerprints technique. The currently supported subset of SQL is limited to `INSERT` and `SELECT` queries only. The implementation thus supports any queries of these kinds with any combinations of supported operations.

A more or less universal relational DBMS has a multitude of possible usage profiles, ranging from purely OLTP to heavy OLAP, and any combinations in between. Moreover, in many cases interaction with the DBMS is not a continuous process and only happens occasionally. This, and the fact that the model implementation is under further development and is currently in an early prototype phase, led us to deeming performance tests on the model implementation unreasonable and pointless.

However, for any specific usage scenario, it is possible to estimate how the model affects performance based on the analysis of performance of basic operations over encrypted data, which was done and presented earlier in this chapter.

However, correctness of the currently implemented functionality was extensively tested, including higher load tests involving processing of relatively large amounts of data. Overall amount of tests done on the implementation of the model is too big to be fully covered in this work, so only some of them are described in the following sections.

7.3.1 Correctness of Implemented SQL Queries

INSERT and Simple SELECT Queries

Implementation of **INSERT** queries is rather straightforward and does not have edge cases that need to be specifically tested for. A working assumption was that if an **INSERT** query is followed by a simple **SELECT** query that retrieves all inserted rows and decrypts them, and the decrypted result of the **SELECT** query is identical to what was inserted, then the data was inserted correctly. The implementation has consistently been showing this behavior. It was specifically tested for this in tests with thousands of iterations with both small and large (several millions rows) inserts.

SELECT Queries

For the tests in this category, two separate databases were used—one for encrypted queries and one for plaintext queries; the latter was used for verification purposes. The same queries were sent to both databases, but for the encrypted one all the queries were first transformed. The results of the queries were then compared and if they appeared to be equal³, the test was counted as passed. Both databases were populated with the same data (several millions rows with random unsigned integer values) using **INSERT** queries.

All tests that were conducted have passed. In this section we will cover some types of queries that were tested and verified to yield equal results.

Starting with simple **SELECT** queries, a most basic type of the query was tested:

```
SELECT [col1], ..., [coln]  
FROM   [Table]
```

And a full-join **SELECT**:

³Disregarding the order of the returned rows, according to the specification of SQL, since no explicit ordering hints were supplied in the queries.

```

SELECT [col11], ..., [coln1], [col12], ..., [colm2]
FROM   [Table1], [Table2]

```

Single-table queries with a simple WHERE predicate:

```

SELECT [col1], ..., [coln]
FROM   [Table]
WHERE  [colk] = <const>

SELECT [col1], ..., [coln]
FROM   [Table]
WHERE  [colk] = [colj]

```

And with a more complex WHERE predicate:

```

SELECT [col1], ..., [coln]
FROM   [Table]
WHERE  [colk] = [colj] AND/OR [colp] = <const> AND/OR ...

```

Multi-table queries with WHERE predicates:

```

SELECT [col11], ..., [coln1], [col12], ..., [colm2]
FROM   [Table1], [Table2]
WHERE  [colk1] = [colj2] AND/OR [colp1] = <const> AND/OR ...

```

Queries with operation of addition in the select expressions:

```

SELECT [colj] + <const>, [colk] + [colp], <const> + <const>, ...
FROM   [Table]

SELECT [colj] + <const> + [colk] + ...
FROM   [Table]

```

Queries with operation of addition in the predicate:

```

SELECT [col1], ..., [coln]
FROM   [Table]
WHERE  [colk] + [colp] = [colj] + <const> AND/OR ...

```

And queries that combine all of the features:

```

SELECT [coli1] + [colj2] + <const> + ..., ...
FROM   [Table1], [Table2]
WHERE  [colk1] + [colp1] = [colq2] + <const> AND/OR ...

```

These were some examples of the notable types of queries that were used to test the implementation for correctness.

7.3.2 Correctness of OLTP

In order to test the correctness of OLTP, the atomicity and isolation of both individual queries and transactions were put to test. Additionally, transactions were tested to work correctly from the point of view of producing expected results in terms of both query results and the resulting state of the database. These tests are described in the section considering atomicity of transactions below.

Atomicity of Individual Queries

To check the atomicity of individual queries, a possibility to abort the queries mid-execution was required. Since single queries typically require a short time to be executed, several approaches were employed to slow queries down.

First, the DBMS was run in a virtual machine with the resources allowance only slightly (10–15%) larger than the minimum requirements. For a typical SQL Server installation the minimum requirements are a single-core x64 CPU with a clock speed of 1.4 GHz, and 1 GB of RAM, as stated in the documentation [52].

Second, the queries were designed to run longer. `INSERT` queries were made large, around 15–20% of maximum query size, which under normal circumstances is 256 MB [51]. `SELECT` queries were artificially slowed down by being run against very large tables, by including large and complex expressions, and by adding redundant joins; e.g., a query `SELECT * FROM [Table]` becomes `SELECT [T].* FROM [Table] [T], [Table] [Y]`, which increases the amount of data the server needs to go through from n for the first query to $2n^2$ for the second one.

These tests were done by hand, and a typical workflow was as follows:

1. Dump the original state of the database into one or several files (DDL definitions and data)
2. Start execution of a query
3. Terminate (kill) the Proxy process before the query is finished
4. Dump the current state of the database into another set of files
5. Analyze these files for differences

The database state was dumped using a free database management tool HeidiSQL with its feature of exporting a database into a SQL file. Differences were analyzed using such tools as `sort` and `diff`.

If the analysis was showing that the database state has not changed, then the test was counted as passed. Eligible changes of the database were amendments to

the data schema—added, removed, or changed relations or attributes; and modified data—added, removed, or changed records.

Tests were conducted with `INSERT` and `SELECT` queries (including multiple-pass `SELECT` queries), which were interrupted at varying moments of execution. All tests have passed.

Isolation of Individual Queries

To test queries for isolation, two simultaneous database connection sessions were used, which were imitating two users working with the database concurrently. Each session was connected to the DBMS through its own instance of the proxy.

Among the supported types of queries, there are 4 possible pairs of queries to test: `INSERT-INSERT`, `INSERT-SELECT`, `SELECT-INSERT`, and `SELECT-SELECT`. However, since `SELECT` queries were shown not to affect the state of the database, apart from creating temporary relations, which possibly can not affect isolation since they are guaranteed to only be visible to the session they were created in, there is no sense in testing the pair `SELECT-SELECT`.

The other 3 pairs were tested in a similar way. For a pair q_1 – q_2 , the query q_1 was designed to run for a relatively long time, and the query q_2 was designed to run for a relatively short time, so that the second query was started after the first one was started and finished before the first one was. On start, the first client was initiating the query q_1 , and then was spawning the second client, who immediately was sending the query q_2 . Knowing the state of the database before the first client was started, the results of the queries and the resulting state of the database were completely predictable, so correctness was easily checked by direct analysis of the results and the database.

The tests were conducted and have met the expected results.

Atomicity of Transactions

Testing whether the transaction atomicity holds was done by ensuring that the database state was not changed until the `COMMIT` query was issued.

In order to test the atomicity of transactions, two types of tests were conducted.

In the tests of the first type, a database client was initiating a transaction with `SERIALIZABLE` isolation level consisting of multiple queries from the supported subset of SQL. The `COMMIT` query was never issued, and the client process was terminated. The database was then checked to have an unchanged state. The tests results were as expected.

In the tests of the second type, a database client was initiating same transactions but this time it *was* issuing a **COMMIT** query in the end of the transaction, and only after that it was aborting the connection to the DBMS. The database was then checked to have the expected state. This tests were also used to verify that transactions are actually bringing the database into a correct expected state. Tests appeared to be successful.

Isolation of Transactions

Checking isolation of transactions for correctness was done with the use of two concurrent users connected to the DBMS. Each of them was starting a serializable transaction with a controlled and synchronized issuing of queries in these transactions, which is illustrated in Figure 7.4.

Figure 7.4: Queries schedule for testing the isolation of transactions

During the test execution, data for analysis was obtained at points Point 1, Point 2, and Point 3: results of **SELECT** queries at these points were analyzed in the context of known database state before both transactions started.

Expected results were that both **SELECT** queries at point Point 1 and Point 2 would not reflect the **INSERT** query from another transaction, and the **SELECT** query at Point 3 was expected to reflect the changes to the database made by the **INSERT** query. The tests matched the expectations.

7.4 Summary

This chapter covers the practical work that was done in the direction of implementing a practical system based on the model, and its components.

As the homomorphic cryptosystems are not widely used, it was required to first implement the Paillier and ElGamal cryptosystems. In order to do so, an efficient implementation of BigInteger was developed. The cryptosystems were then built on top of that implementation, and both the cryptosystems and the BigInteger implementation were tested for correctness and performance.

The equality matching module was implemented based on the model of data fingerprints that was introduced in earlier chapters. It was systematically checked for performance and the results were discussed.

On top of these components an implementation of a model of an encrypted database was created as a wrapper for MS SQL Server. The implementation is still under further development, but the functionality that was already implemented was thoroughly checked for correctness.

Chapter 8

Conclusions

A database is more than a “dumb” data storage. Unlike the latter, a database is able to **1)** retrieve precisely the data that is required using compound queries, as opposed to retrieving “raw” data that cannot be directly used by the querier; **2)** aggregate and process the data in order to create a new meaning. Additionally, a database typically takes full care of concurrent sessions, provides means to ensure atomicity of complex procedures, checks the data to be compliant with predefined rules.

Specific database systems are oriented to serve their own subset of purposes. Some are targeting very narrow problem sets, a good example would be geospatial database systems like GeoMesa. Others try to be a more or less universal solution, e.g., PostgreSQL, SQL Server, or MongoDB. Regardless of that, they aim at providing the features of a “smart” data storage. Migration of such systems to the cloud is more and more often considered as a business advantage tool [1]. However, new risks that are inherent to the cloud stop, limit, or, at least, slow down this process for many and many enterprises [63]. Introducing cryptography into the picture is able to mitigate many of the risks and sway those who are still on the fence towards a more efficient and agile cloud solution [40]. As it was stated in the research objective, this research project was an attempt to investigate the abstract features of data storage and processing in an encrypted environment. In the beginning of this work a trifold approach to achieve this objective has been stated:

1. To develop an abstract theoretical model of an encrypted database. This was called a *methodology*, as the model clearly shows how to create a practical encrypted database system tailored for specific requirements;
2. To show how the abstract model could be instantiated as a practical system: how a specific procedure of data manipulation could be implemented in an

encryption-aware fashion, how the model integrates with the DBMS and the querying workflow;

3. To develop certain basic operations over encrypted data and analyze their traits.

Creating a more or less versatile encrypted database system requires a set of issues to be deeply investigated. First, we needed to find answers to a set of high-level questions: how cryptography could be brought into data processing? How does that translate to a cloud database system scenario? What topology would such a system have, what workflow would it follow?

Certain cryptosystems have been known for their homomorphic properties: such cryptosystems preserve one or more operations. The most well-known example of such cryptosystem is the original RSA cryptosystem, which preserved multiplication. Unfortunately, the original RSA—without random padding—has been shown to be insecure, and its modification with random padding loses its homomorphic properties. However, multiple other cryptosystems were devised that had homomorphic properties. Most of them preserved only one operation—either addition or multiplication—and a *fully* homomorphic cryptosystem (such that preserves both operations) was considered a holy grail of cryptography for years [75]. At some point in time, well known researchers Boneh, Goh, and Nissim came up with an *almost* fully homomorphic cryptosystem, the Boneh-Goh-Nissim (BGN) cryptosystem. Later, in 2009 Craig Gentry published a work that described a fully homomorphic encryption scheme [32]. Although this was a theoretical breakthrough, the suggested scheme is hardly usable in practice due to its performance.

Still, there are homomorphic cryptosystems that are feasible to use in practice, e.g., ElGamal, which preserves multiplication; Paillier, which preserves addition; and many others¹. And even if a practically usable fully homomorphic cryptosystem existed, there still are other operations that we would typically like to see in a database, which such cryptosystem could not provide: searching capabilities or greater/less comparisons, to name a few. On the other hand, there is a number of approaches to privacy-preserving search, there are order-preserving encryptions, which are handy for greater/less comparisons. Clearly, a direction that is worth research is combining this individual operations and using them as building blocks to construct a system that can execute more or less arbitrary sequences of such operations over encrypted data.

A property of the proposed model that was considered important was its prac-

¹Benaloh, Okamoto-Uchiyama, Boneh-Goh-Nissim, Schmidt-Samoa-Takagi, Damgård-Jurik, etc.

ticality, i.e., feasibility of creating a system based on the model that could be used in practice. This was a key argument to choose a wrapper approach—to extend existing DBMSs with new functionality—instead of designing an encryption-oriented DBMS from scratch. Another practical point is that the user (the application he uses) possibly and even likely is limited in computational resources: consider a mobile platform, a thin client, a web application—typical points of entry for an enterprise user. All these considerations led to a clear view of a topology of the proposed system.

It is given that there are two entities in the system: a trusted authorized user, and an untrusted regular DBMS located in the cloud. According to the threat model, the DBMS is being constantly observed by a malicious adversary, who has infiltrated the cloud platform provider. The adversary has full access to the DBMS and the hardware it runs on, but in order to stay unnoticed, he never actively interferes with the system: he does not modify data, block or amend execution of protocols, etc. He can run data retrieval queries, granted he is able to construct a meaningful one.

Taking into account the user’s limited resources, we introduce into the system one or several intermediate agents—proxies—that are trusted and computationally powerful. Proxies have access to cryptographic secrets (thus they need to be trusted) and are intended to perform parts of encrypted queries that require knowledge of these secrets and hence can not be delegated to the untrusted DBMS. So, a typical query execution workflow in the proposed model would be as follows:

1. User issues a query and sends it over to the proxy.
2. Proxy knows how to execute the query using the DBMS, so the proxy initiates an interactive challenge–response session with the DBMS.
3. Eventually, the proxy gets all the results from the DBMS, decrypts them, and sends over to the user.

8.1 Theoretical Model

Having decided on the general direction this research would follow, we needed to find answers to more specific questions. Namely, the questions were **1)** assuming we know how to perform certain basic operations, how can we execute complex queries based on combinations of these operations? **2)** if we know how to execute these queries, how do they behave in an environment with an SLA² less than 100% and in multi-user

²SLA stands for Service Level Agreement. Real systems can go down for various reasons—planned maintenance, power outage, software bugs, networking issues, etc. Informally, an SLA, typically given in percents, is a guaranteed by the provider up-time ratio. SLA of 99% means that for a given time span—typically, a month—the service is guaranteed to be up and operational for 99% of that time span.

scenarios.

Concerning the first question, an abstract model of transforming plaintext SQL (or SQL-like) queries was proposed. Defining the model we have considered all types of supported queries, and for each type defined how exactly a plaintext query is transformed into its encryption-aware equivalent. We introduced a formal definition of a space of possible plaintext queries and defined a mapping of elements of this space into ordered sets (basically, compositions) of elements of a formally defined space of encryption-aware queries. Some elements of the plaintext space are mapped into a single element of the encryption-aware space, and some are mapped into a composition of such elements, in which case the proxy performs its part of intermediate computations.

Having a fully defined mapping of plaintext queries into their encryption-aware equivalents, we have thoroughly investigated how these equivalents act from the point of view of ACID properties. For a subset of queries that supposedly was ACID compliant without additional efforts, we have shown that it indeed is. For the rest of queries, we have shown how to achieve ACID compliance for them. We have also analyzed how the now ACID-compliant queries behave in a transaction setting, and shown that transactions are fully and correctly supported.

8.2 Implementation and Empirical Results

The practical implementation part of the work comprises of two parts. First is to implement some of the “building blocks”, the basic modules that enable a specific operation to work over encrypted data. Second is to create an implementation of the encrypted database model based on these modules.

Regarding the operations in O_v , the modules that implement addition and multiplication were developed based on the Paillier and ElGamal cryptosystems, respectively. Regarding operations in O_b , the equality matching operation module was developed. This module is based on a novel flexible approach to performing search over encrypted data, which was thoroughly investigated, analyzed, and metered for performance, which shown very viable from the practical point of view results. The arithmetic operations modules were tested for correctness and performance as well. However, in development of these modules only a minimal effort was made towards performance optimization, so the results, which now show a significant performance degradation as compared to plaintext operations, are likely to have room for significant improvement.

An implementation of the model was developed in an iterative fashion, gaining new functionality with every iteration, and is still in active development. A prototype implementation of the model by the time this text was written supported **INSERT** and **SELECT** queries with operations of addition and equality matching in arbitrary combinations, and was in further development. The implementation was designed using principles of modularity so that it is easier to extend it or modify certain parts without affecting other parts; it was thoroughly tested for correctness, and the test suite is growing along with the added functionality to ensure good functionality coverage and that no new functionality breaks older behavior.

8.3 Other Results

In other results that were achieved while this project was carried out, were 3 top-tier conference publications:

- Vasily Sidorov, Wee Keong Ng, “**Complex Queries in a Shared Multi User Relational Cloud Database**”, 2013, *2013 IEEE 6th International Conference on Cloud Computing (CLOUD)*, In *Proceedings of*, pp. 903–909, doi:10.1109/CLOUD.2013.49
- Vasily Sidorov, Wee Keong Ng, “**Model of an Encrypted Cloud Relational Database Supporting Complex Predicates in WHERE Clause**”, 2014, *2014 IEEE 7th International Conference on Cloud Computing (CLOUD)*, In *Proceedings of*, pp. 667–672, doi:10.1109/CLOUD.2014.94
- Vasily Sidorov, Wee Keong Ng, “**Transparent Data Encryption for Data-in-Use and Data-at-Rest in a Cloud-Based Database-as-a-Service Solution**”, 2015, *2015 IEEE 11th World Congress on Services (SERVICES)*, In *Proceedings of*, (pending publication)

Apart from that, a paper “A Privacy-Preserving Relational Cloud Database Supporting Complex Search Queries” by Vasily Sidorov and Wee Keong Ng has been submitted to a top-tier journal IEEE Transactions on Cloud Computing Special Issue on Cloud Security Engineering. As for the moment of writing this, it has been reviewed, and is currently in the edits.

In order to make the implementation part of this project more feasible, an MOE Academic Research Fund Tier 1 research grant was successfully applied for, and is partially being used to facilitate the development of the practical solution.

8.4 Limitations

As it typically happens, time and resources constraints appeared to be a limiting factor in this project, and forced narrowing of the scope of the research.

A subset of SQL that was considered in this work was limited to the essentials, leaving out such useful types and families of queries, as `UPSERT`³ queries, nested queries, `INSERT . . . SELECT` queries, and some others. Among the types of queries that *were* investigated, some of their typical functionality was left out; e.g., aggregation operations, grouping, ordering were not in the scope of this research.

Practical implementation of the model, though being actively developed and already achieving results, is still in an early prototype stage, as the development of such system requires a detailed design and a substantial amount of development time.

8.5 Future Research

While we aimed for this work to describe a concise and consistent, finished model, there is a number of directions for future research based on this work.

The model described in this work is only defined to work with a single domain of plaintext values. The model is abstracted from knowing what specific domain that is, so it could be any scalar domain (or a domain that is reducible to a scalar one). The practical part of the work was implemented around a domain of integer values. A logical direction for future research would be to extend the model to support multiple domains.

Going further in the multiple-domains direction, it would be logical to be aimed at supporting a *partially* encrypted database. In many cases there is no real need to encrypt the whole database, as a significant fraction of the information in the database could be of no interest for the adversary: it could be some service data, it could be publicly available data, etc. Working with such information in the encrypted form would create an unnecessary overhead.

As the model is currently designed, it is more oriented towards correctness rather than anything else. However, if considered in a less abstract fashion, execution of certain subsets of queries might be optimized. An obvious example is a query with an expression like `col1 + col2 + col3`, which, if the model is followed precisely, would be converted into a multiple-pass query, which first computes `col_tmp = col1 + col2` and then computes `col_tmp + col3`, but the expression clearly could be

³A query that atomically either updates a record if it exists, or creates a new one otherwise. Term originates from a combination of words “update” and “insert”.

computed in a single pass. Performance optimization in general and in specific cases is an important direction for future research.

This work was considering transactions in their textbook definition based on serializability. Serializable isolation of transactions is often unnecessary and many DBMSs support weaker isolation policies, and some even default the isolation level of transactions to a weaker one. If the model was analyzed in the context of weaker-isolated transactions and adjusted to support them, it could have highly increased its performance in a high-load multi-user scenario.

The proposed implementation of search is able to work with multiple proxies, which was shown to both lower trust requirements for the proxy and increase the security of the system. An investigation of means to do the similar secret distribution between multiple proxies for other building blocks, e.g., the homomorphic arithmetics ones, could lead to an increased security of the overall system—that is if the threat model is extended to also consider attacks on proxies.

Relational database systems dominated the market for a long time. However, in many cases the relational model might be too strict and complex for the actual data the system works with; additionally, relational database systems typically are hard to scale horizontally, and quickly hit the limit of vertical scaling with the rates data amounts grow, especially in data-driven enterprises. Scalable DBMSs with simpler data models—document-oriented or key-value databases, the NoSQL and NewSQL solutions—attempt to satisfy this demand. Investigation of ways to apply approaches described in this work to such DBMSs is a very attractive direction for the future research.

This work considered an encrypted relational database system in the Database-as-a-Service usage scenario. It provided a theoretical discussion on the topic and empirical results of the proposed models. Approaches presented in this work allow for flexibility and modularity, which gives this model chances to evolve in a useful practical solution to an open and important problem of preserving privacy and confidentiality of data-in-use.

References

- [1] Business Agility in the Cloud. Whitepaper, Harvard Business Review, June 2014.
- [2] The Zettabyte Era: Trends and Analysis. Whitepaper, Cisco, May 2015.
- [3] *Information Security Handbook*, chapter Confidentiality, Integrity & Availability. IS-RMC, LLC, web resource: <http://ishandbook.bsewall.com/risk/Methodology/CIA.html>, accessed on 24 May 2015, published in 2009.
- [4] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thom, L. Valenta, et al. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. 2015. web resource: <https://www.weakdh.org/imperfect-forward-secrecy.pdf>, accessed on 25 July 2015.
- [5] G. Amanatidis, A. Boldyreva, and A. O’Neill. New Security Models and Provably-Secure Schemes for Basic Query Support in Outsourced Databases. In *Working Conference on Data and Applications Security (DBSec’07)*. LNCS. Springer, Heidelberg, pages 14–30. Springer-Verlag, 2007.
- [6] Asay, Matt. NoSQL databases eat into the relational database market. Web resource: [https://msdn.microsoft.com/en-us/library/ms143432\(v=sql.110\).aspx](https://msdn.microsoft.com/en-us/library/ms143432(v=sql.110).aspx), accessed on 27 Dec 2015, published on 4 Mar 2015.
- [7] ASPG Inc. The three states of digital data. Web resource: <http://aspg.com/three-states-digital-data/>, accessed on 16 April 2015, published on 3 September 2014.
- [8] F. Bao, R. Deng, X. Ding, and Y. Yang. Private Query on Encrypted Data in Multi-user Settings. In L. Chen, Y. Mu, and W. Susilo, editors, *Information Security Practice and Experience*, volume 4991 of *Lecture Notes in Computer Science*, pages 71–85. Springer Berlin Heidelberg, 2008.
- [9] E. Bertino, S. Jajodia, and P. Samarati. Supporting multiple access control policies in database systems. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 94–107, May 1996.
- [10] A. Boldyreva, N. Chenette, Y. Lee, and A. O’Neill. Order-Preserving Symmetric Encryption. In A. Joux, editor, *Advances in Cryptology — EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 224–241. Springer Berlin Heidelberg, 2009.
- [11] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Advances in Cryptology-Eurocrypt 2004*, pages 506–522. Springer.
- [12] D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. J. Wu. Private Database Queries Using Somewhat Homomorphic Encryption. 2012.
- [13] D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-DNF Formulas on Ciphertexts. In J. Kilian, editor, *Theory of Cryptography*, volume 3378 of *Lecture Notes in Computer Science*, pages 325–341. Springer Berlin Heidelberg, 2005.

- [14] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. *Theory of Cryptography*, pages 535–554, 2007.
- [15] R. Brinkman, B. Schoenmakers, J. Doumen, and W. Jonker. Experiments with queries over encrypted data using secret sharing. *Secure Data Management*, pages 33–46, 2005.
- [16] C. Bösch, R. Brinkman, P. Hartel, and W. Jonker. Conjunctive wildcard search over encrypted data. *Secure Data Management*, pages 114–127, 2011.
- [17] C. Bösch, Q. Tang, P. Hartel, and W. Jonker. Selective Document Retrieval from Encrypted Database. *Information Security*, pages 224–241, 2012.
- [18] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.
- [19] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally Composable Two-Party and Multi-Party Secure Computation. In *Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*, pages 494–503. ACM.
- [20] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.
- [21] Columbus, Louis. Roundup Of Cloud Computing Forecasts And Market Estimates Q3 Update, 2015. *Forbes*; Web resource: <http://www.forbes.com/sites/louiscolumbus/2015/09/27/roundup-of-cloud-computing-forecasts-and-market-estimates-q3-update-2015/>, accessed on 28 Dec 2015, published on 27 Sep 2015.
- [22] J. M. Correia, L. F. Wurster, C. Eschinger, Y. Dharmasthira, and B. Igou. Buyers Reveal Cloud Application Adoption Plans Through 2017. Survey analysis, Gartner, October 2014.
- [23] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 79–88. ACM.
- [24] E. Damiani, S. De Capitani di Vimercati, and P. Samarati. *Computing range queries on obfuscated data*. Università La Sapienza, 2004.
- [25] E. Damiani, S. D. C. di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Key management for multi-user encrypted databases. In *Proceedings of the 2005 ACM workshop on Storage security and survivability*, StorageSS’05, pages 74–83, New York, NY, USA, 2005. ACM.
- [26] E. Damiani, S. D. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS ’03, pages 93–102, New York, NY, USA, 2003. ACM.
- [27] C. Dong, G. Russello, and N. Dulay. Shared and searchable encrypted data for untrusted servers. *Journal of Computer Security*, 19(3):367–397, 2011.
- [28] T. ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In G. Blakley and D. Chaum, editors, *Advances in Cryptology*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer Berlin Heidelberg, 1985.

- [29] A. Feldman, W. Zeller, M. Freedman, and E. Felten. SPORC: Group collaboration using untrusted cloud resources. *OSDI*, 2010.
- [30] R. Finos. Public Cloud Market Shares 2014 and 2015. Survey, Wikibon, August 2015.
- [31] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.
- [32] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.
- [33] C. Gentry and S. Halevi. Implementing Gentry’s Fully-Homomorphic Encryption Scheme. In K. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer Berlin Heidelberg, 2011.
- [34] E.-J. Goh. Secure indexes. *Cryptography ePrint Archive, Report*, 216, 2003.
- [35] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.
- [36] P. Golle, J. Staddon, and B. Waters. Secure conjunctive keyword search over encrypted data. In M. Jakobsson, M. Yung, and J. Zhou, editors, *Applied Cryptography and Network Security*, volume 3089 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2004.
- [37] H. Hacigümüş, B. Hore, B. Iyer, and S. Mehrotra. Search on Encrypted Data. *Secure Data Management in Decentralized Systems*, pages 383–425, 2007.
- [38] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD ’02, pages 216–227, New York, NY, USA, 2002. ACM.
- [39] H. Hacigümüş and S. Mehrotra. Performance-Conscious Key Management in Encrypted Databases. In C. Farkas and P. Samarati, editors, *Research Directions in Data and Applications Security XVIII*, volume 144 of *IFIP International Federation for Information Processing*, pages 95–109. Springer US, 2004.
- [40] E. C. Hayden. Extreme cryptography paves way to personalized medicine. *Nature*, 519:400–401, March 2015.
- [41] K. Henry. The Theory and Applications of Homomorphic Cryptography. Master’s thesis, University of Waterloo, 2008.
- [42] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu. Secure multidimensional range queries over outsourced data. *The VLDB Journal*, 21(3):333–358, 2012.
- [43] B. Hore, S. Mehrotra, and G. Tsudik. A Privacy-Preserving Index for Range Queries. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB ’04, pages 720–731. VLDB Endowment, 2004.
- [44] N. T. Hung, D. H. Giang, N. W. Keong, and H. Zhu. Cloud-Enabled Data Sharing Model. *Intelligence and Security Informatics (ISI), 2012 IEEE International Conference on*, 2012.
- [45] M. Kantarcioglu and C. Clifton. Security Issues in Querying Encrypted Data. In S. Jajodia and D. Wijesekera, editors, *Data and Applications Security XIX*, volume 3654 of *Lecture Notes in Computer Science*, pages 325–337. Springer Berlin Heidelberg, 2005.

- [46] S. Lei, D. Zishan, and G. Jindi. Research on Key Management Infrastructure in Cloud Computing Environment. In *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, pages 404–407. IEEE, nov. 2010.
- [47] J. Li, M. Krohn, D. Mazieres, and D. Shasha. *Secure untrusted data repository (SUNDR)*. Defense Technical Information Center, 2003.
- [48] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou. Enabling Efficient Fuzzy Keyword Search over Encrypted Data in Cloud Computing. Technical report, Cryptology ePrint Archive, Report 2009/593, 2009.
- [49] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity. Report, McKinsey Global Institute, June 2011.
- [50] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC press, 1996.
- [51] Micorosoft Corporation. Maximum Capacity Specifications for SQL Server. Web resource: [https://msdn.microsoft.com/en-us/library/ms143432\(v=sql.110\).aspx](https://msdn.microsoft.com/en-us/library/ms143432(v=sql.110).aspx), accessed on 28 Jul 2015.
- [52] Microsoft Corporation. Hardware and Software Requirements for Installing SQL Server 2012. Web resource: [https://msdn.microsoft.com/en-us/library/ms143506\(v=sql.110\).aspx](https://msdn.microsoft.com/en-us/library/ms143506(v=sql.110).aspx), accessed on 28 Jul 2015.
- [53] E. Mykletun and G. Tsudik. Aggregation Queries in the Database-As-a-Service Model. In E. Damiani and P. Liu, editors, *Data and Applications Security XX*, volume 4127 of *Lecture Notes in Computer Science*, pages 89–103. Springer Berlin Heidelberg, 2006.
- [54] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In J. Stern, editor, *Advances in Cryptology — EUROCRYPT ’99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer Berlin Heidelberg, 1999.
- [55] M. Paul and A. Saxena. Proof Of Erasability for Ensuring Comprehensive Data Deletion in Cloud Computing. In N. Meghanathan, S. Boumerdassi, N. Chaki, and D. Nagamalai, editors, *Recent Trends in Network Security and Applications*, volume 89 of *Communications in Computer and Information Science*, pages 340–348. Springer Berlin Heidelberg, 2010.
- [56] M. Paul and A. Saxena. Zero Data Remnance Proof in Cloud Storage. *International Journal of Network Security & Its Applications (IJNSA)*, 2(4), 2010.
- [57] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality With Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.
- [58] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Processing Queries on an Encrypted Database. *Communications of the ACM*, 55(9):103–111, 2012.
- [59] R. Popa, N. Zeldovich, and H. Balakrishnan. CryptDB: A Practical Encrypted Relational DBMS. Technical report, MIT-CSAIL-TR-2011-005, MIT, 2011.
- [60] C. Ramaswamy and R. Sandhu. Role-based access control features in commercial database management systems. In *Proc. 21st national Information Systems security Conference, 1998*, pages 503–511, 1998.
- [61] M. Raykova, A. Cui, B. Vo, B. Liu, T. Malkin, S. Bellovin, and S. Stolfo. Usable, Secure, Private Search. *Security Privacy, IEEE*, 10(5):53–60, Sept.–Oct. 2012.

- [62] G. Russello, C. Dong, N. Dulay, M. Chaudron, and M. van Steen. Providing data confidentiality against malicious hosts in Shared Data Spaces. *Science of Computer Programming*, 75(6):426–439, 2010.
- [63] L. Santos, K. C. Yeoh, John anf Scoboria, and E. Scoboria. Cloud Usage: Risks and Opportunities Report. Survey, Cloud Security Alliance, September 2014.
- [64] R. Shirey. Internet Security Glossary, Version 2. RFC 4949 (Informational); web resource: <http://www.ietf.org/rfc/rfc4949.txt>, accessed on 2 Feb 2015, published in Aug 2007.
- [65] V. Sidorov and W. K. Ng. Complex Queries in a Shared Multi User Relational Cloud Database. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 903–909, June 2013.
- [66] V. Sidorov and W. K. Ng. Model of an Encrypted Cloud Relational Database Supporting Complex Predicates in WHERE Clause. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 667–672, June 2014.
- [67] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on*, pages 44–55, 2000.
- [68] Q. Tang. Search in Encrypted Data: Theoretical Models and Practical Applications. Cryptology ePrint Archive, Report 2012/648, 2012. <http://eprint.iacr.org/>.
- [69] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou. Secure Ranked Keyword Search over Encrypted Cloud Data. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 253–262, June 2010.
- [70] C. Wang, K. Ren, W. Lou, and J. Li. Toward publicly auditable secure cloud data storage services. *Network, IEEE*, 24(4):19–24, July–August 2010.
- [71] C. Wang, Q. Wang, K. Ren, and W. Lou. Ensuring Data Storage Security in Cloud Computing. In *Quality of Service, 2009. IWQoS. 17th International Workshop on*, pages 1–9. IEEE, July 2009.
- [72] H. Wang. Privacy-preserving data sharing in cloud computing. *Journal of Computer Science and Technology*, 25(3):401–414, 2010.
- [73] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li. Enabling Public Auditability and Data Dynamics for Storage Security in Cloud Computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(5):847–859, May 2011.
- [74] W. K. Wong, D. W.-l. Cheung, B. Kao, and N. Mamoulis. Secure kNN computation on encrypted databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, SIGMOD '09*, pages 139–152, New York, NY, USA, 2009. ACM.
- [75] D. J. Wu. Fully Homomorphic Encryption: Cryptography’s Holy Grail. *XRDS*, 21(3):24–29, March 2015.
- [76] Y. Yang. Towards Multi-User Private Keyword Search for Cloud Computing. pages 758–759. IEEE.
- [77] Y. Yang, X. Ding, R. Deng, and F. Bao. Multi-User Private Queries over Encrypted Databases. *Int. J. High Performance Computing and Networking*, 1(1/2/3):64–74, 2008.

- [78] Z. Yang, S. Zhong, and R. Wright. Privacy-Preserving Queries on Encrypted Data. *Computer Security—ESORICS 2006*, pages 479–495, 2006.
- [79] S. Yu, C. Wang, K. Ren, and W. Lou. Achieving Secure, Scalable, and Fine-grained Data Access Control in Cloud Computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, March 2010.
- [80] Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *Proceedings of the first ACM conference on Data and application security and privacy*, CODASPY '11, pages 237–248, New York, NY, USA, 2011. ACM.