Filipe Miguel Silva Correia

Privacy-Enhanced Query Processing in a Cloud-Based Encrypted DBaaS (Database as a Service)

Relatório intermédio para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Henrique João Lopes Domingos, Full Professor,
Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

10 de Julho de 2018

Abstract

In this dissertation, we researched techniques to support trustable and privacy-enhanced solutions for on-line applications accessing to "always encrypted" data in remote DBaaS (data-base-as-a-service) or Cloud SQL-enabled backend solutions.

Although solutions for SQL-querying of encrypted databases have been proposed in recent research, they fail in providing: (i) flexible multimodal query facilities including online image searching and retrieval as extended queries to conventional SQL-based searches, (ii) searchable cryptographic constructions for image-indexing, searching and retrieving operations, (iii) reusable client-appliances for transparent integration of multimodal applications, and (iv) lack of performance and effectiveness validations for Cloud-based DBaaS integrated deployments.

At the same time, the study of partial homomorphic encryption and multimodal searchable encryption constructions is yet an ongoing research field. In this research direction, the need for a study and practical evaluations of such cryptographic is essential, to evaluate those cryptographic methods and techniques towards the materialization of effective solutions for practical applications.

The objective of the dissertation is to design, implement and perform experimental evaluation of a security middleware solution, implementing a client/client-proxy/server-appliance software architecture, to support the execution of applications requiring on-line multimodal queries on "always encrypted" data maintained in outsourced cloud DBaaS backends. In this objective we include the support for SQL-based text-queries enhanced with searchable encrypted image-retrieval capabilities. We implemented a prototype of the proposed solution and we conducted an experimental benchmarking evaluation, to observe the effectiveness, latency and performance conditions in supporting those queries. The dissertation addressed the envisaged security middleware solution, as an experimental and usable solution that can be extended for future experimental testbench evaluations using different real cloud DBaaS deployments, as offered by well-known cloud-providers.

Keywords: *Data confidentiality; Encrypted databases; Encrypted Query Processing; Cloud DBaaS;*

Searchable Encryption Methods; Partial Homomorphic Encryption, Content-Aware Searchable Encryption; Multimodal Search

Resumo

Nesta dissertação foram investigadas técnicas para suportar soluções com garantias de privacidade para aplicações que acedem on-line a dados que são mantidos sempre cifrados em nuvens que disponibilizam serviços de armazenamento de dados, nomeadamente soluções do tipo bases de dados interrogáveis por SQL. Embora soluções para suportar interrogações SQL em bases de dados cifradas tenham sido propostas anteriormente, estas falham em providenciar: (i) capacidade de efectuar pesquisas multimodais que possam incluir pesquisa combinada de texto e imagem com obtenção de imagens online, (ii) suporte de privacidade com base em construções criptograficas que permitam operações de indexacao, pesquisa e obtenção de imagens como dados cifrados pesquisáveis, (iii) suporte de integração para aplicações de gestão de dados em contexto multimodal, e (iv) ausência de validações experimentais com benchmarking dobre desempenho e eficiencia em soluções DBaaS em que os dados sejam armazenados e manipulados na sua forma cifrada.

A pesquisa de soluções de privacidade baseada em primitivas de cifras homomórficas parciais, tem sido vista como uma possível solução prática para interrogação de dados e bases de dados cifradas. No entanto, este é ainda um campo de investiigação em desenvolvimento. Nesta direção de investigação, a necessidade de estudar e efectuar avaliações experimentais destas primitivas em bibliotecas de cifras homomórficas, reutilizáveis em diferentes contextos de aplicação e como solução efetiva para uso prático mais generalizado, é um aspeto essencial.

O objectivo da dissertação e desenhar, implementar e efectuar avalições experimentais de uma proposta de solução middleware para suportar pesquisas multimodais em bases de dados mantidas cifradas em soluções de nuvens de armazenamento. Esta proposta visa a concepção e implementação de uma arquitectura de software client/client-proxy/server-appliance para suportar execução eficiente de interrogações online sobre dados cifrados, suportando operações multimodais sobre dados mantidos protegidos em serviços de nuvens de armazenamento. Neste objectivo incluímos o suporte para interrogações estendidas de SQL, com capacidade para pesquisa e obtenção de dados cifrados que podem incluir texto e pesquisa de imagens por similaridade. Foi implementado um prototipo da solução proposta e foi efectuada uma avaliação experimental do mesmo, para observar

as condições de eficiencia, latencia e desempenho do suporte dessas interrogações. Nesta avaliação incluímos a análise experimental da eficiência e impacto de diferentes construções criptográficas para pesquisas cifradas (searchable encryption) e cifras parcialmente homomórficas e que são usadas como componentes da solução proposta.

A dissertaçao aborda a soluçao de seguranca projectada, como uma solução experimental que pode ser estendida e utilizavel para futuras aplcações e respetivas avaliações experimentais. Estas podem vir a adoptar soluções do tipo DBaaS, oferecidos como serviços na nuvem, por parte de diversos provedores ou fornecedores.

Palavras-chave: *Confidencialidade de dados; Bases de Dados Cifradas; Pesquisa e Consulta de dados Cifrados; Bases de Dados como Serviços na Nuvem; Cifras Pesquisáveis; Cifras Pesquisáveis orientadas ao Conteúdo; Pesquisas de Dados Multimodais*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation

Cloud computing and storage have increasingly become usual solutions for individuals and enterprises alike. The necessary infrastructure to host data management, data storage and computation services is present as products and solutions offered by different cloud-providers. These solutions are outsourcing solutions, managed and maintained by the cloud-provider personnel. Thus, clients need not be burdened with the costs of acquiring equipments, software licenses and hiring IT staff. The deployment options for those solutions are flexible and easily addressable. This gives a jump start to companies with smaller initial investment budget. However, even well established organizations are also migrating data and computations to the cloud for its immediate financial benefits, and to better address: (i) ubiquitous and universal access; (ii) business visibility; (iii) collaboration and information sharing; (iv) standardization of IT management functions; (v) scalability and elasticity support for business improvement and growing requirements, with "pay-per-use" models [4, 11, 60].

However, despite these advantages, there are also growing concerns regarding the security and trustability of cloud solutions. Confidentiality guarantees of operations and data-privacy management conditions are in the main concerns. The privacy of outsourced data cannot be guaranteed under the entire user control [12, 15, 24, 64], and it is not protected by current effective Service Level Agreements or required Regulation conditions, for many critical applications that manage sensitive data with on-line processing requirements.

As such, news related to disclosure of sensible information is often discussed in the press and different facts or incidents have been extensively reported in the literature. Aside from potential attacks of adversaries outside the cloud domain, as internet hacking activities against cloud-based software solutions, the cloud provider's own employees have access to costumer data.

These "internal adversaries" can compromise data from legitimate owners, either intentionally or accidentally. Data-leakage, other confidentiality breaks or the violation

of data-privacy expectations can be the result of illicit actions from system-administrators, or from abusive operations performed by operational personnel. For example, in 2013, an attacker with insider knowledge stole personal data of two million Vodafone costumers from a server in Germany [37]. In 2014, a temporary consultant for three credit card firms in South Korea leaked personal data of twenty million of their clients [1]. More recently, Equifax discovered a giant breach which exposed personal data of 143 million american people [27].

Accidental leaks, mostly due to misconfiguration of access control, have also taken place frequently [3, 51, 74], the most impactful being the breach that exposed over 60% of US population's electoral data [74]. Overall, the media have widely covered data privacy issues in cloud platforms [21, 33, 77].

One of the most prevalent cloud models is the IaaS, which allows administrators to remotely provision the underlying IT infrastructure for end-users, on demand. A recent study [79] reveals that the haste with which cloud solutions are being adopted and developed, while overlooking privacy management and access control functions, allows for anyone to potentially access sensitive information behind 16000 cloud domains. By enumerating domain prefixes with words from a dictionary, over 500 domains with guessable folder structures lead to more than 11000 publicly accessible files. Among these files, highly sensitive information could be found, such as usernames, their passwords, emails, credit card transaction logs and business information. In the same study, 8% of the data breaches of 2014 resulted from insider theft.

1.2 Background and Opportunity for the topic

With the tendency for data and computation to move from traditional, corporation owned IT setups, to cloud solutions, both security researchers and cybercriminals have started to pay closer attention to clouds. While the amount of data stored and computations executed in the cloud sharply increase, new cloud solutions arise with PaaS (Platform as a Service) and SaaS (Software as a Service) paradigms, ranging from simple file storage services, key-value stores, to databases offered as services (DBaaS solutions), or SQL-enabled backends for big-data analytics provision.

In line with this direction, academic research [5, 7, 18, 31, 50, 59, 73, 78] as well as industrial efforts [36, 67] have been addressing confidentiality of data and queries performed on remote databases, through novel contributions that leverage existing cryptographic constructions as well as new encryption algorithms with homomorphic properties [5, 18, 50, 59, 78]. These contributions are, in general, mainly focused on privacy protection conditions as countermeasures against external attackers to outsourced database or data storage servers, as well as, possible "honest-but-curious" (HbC) system-administrators. In some research contributions in the literature, these adversaries are also considered in cloud-provided solutions, where HbC adversaries can exist as operation

personnel in each cloud service provider. By keeping the outsourced data always encrypted, restricting access to the respective keys, and transforming and adapting queries so that they can be performed, as encrypted queries over encrypted data, its confidentiality is safeguarded against potential disclosure from HbC system administrators and confidentiality breaks resulting from possible intrusions from external attackers. There are however some relevant limitations in the state of the art solutions, which open the opportunity for ongoing research for the improvement of their effectiveness and scope of use. These limitations are particularly relevant in addressing the requirements and objectives of the dissertation, as we will discuss in more detail in the related work section. Among the limitations and drawbacks we emphasize the following:

- Support for SQL-querying facilities combined with multimodal query functions, including online searchable text and image searching and retrieval.
- Lack of searchable cryptographic constructions for image-indexing, searching and retrieving operations.
- Limitations on the reusability assumptions of searchable encryption and partial homomorphic encryption primitives and reusable libraries, for the transparent integration of such tools and methods in the context of multimodal searching functions and different applications.
- Lack of performance and effectiveness validations and experimental studies, in real Cloud-based DBaaS integrated deployments, as relevant benchmarking evaluations towards the practical and effective adoption of many solutions that have been proposed.

1.3 Objectives and Contributions

To overcome the investigated limitations, including those described above, in the analysis of the researched proposals and state of the art solutions, we have envisioned and then developed a middleware system that acts as a proxy between applications on a client machine and databases or data-repositories outsourced to a cloud. Our proposal aligns a client/client-proxy/serer-appliance software architecture and related system model, to support the execution of client applications performing SQL-remote operations and on-line multimodal queries on "always encrypted" cloud storage backends. In order to safeguard the confidentiality of sensitive information (database data or indexed images), a set of cryptographic constructions involving searchable encryption primitives for image-similarity searches and partial homomorphic encryption constructions are used, as core components in our solution. This allows operations to be made on-line upon always cloud-based encrypted data. Therefore, cloud administrators cannot compromise data confidentiality and the proposed solution can be a key-factor for privacy-enhanced guarantees for the management of sensitive data in cloud-enabled provided data repository solutions.

The design of the proposed middleware solution has been extended to support the indexing, search and retrieval of images, with searchable indexes and images also protected with encryption and stored in the cloud-supported repositories. Most of the developed components of our solution are directly adaptable to different standard DBMSs, and deployable to different commercial clouds, even that in our prototype we don't provide an entire full-fledged SQL enabled product. However the modularity and the generic assumptions beyond the proposed system model allow that only those components of the prototype which are not directly adaptable should be re-implemented for a specific DBMS or for a specific deployment with a different cloud interface, although with a minimal effort. The prototype has been deployed to a cloud DBaaS, and evaluated in terms of effort to adapt to that platform as is. Furthermore, we have tested the solution for efficiency and scalability, by conducting an experimental observation of such criteria to evaluate the performance and impact of different cryptographic constructions, and to measure the latency and thoughput conditions in a benchmark evaluation of the implemented prototype. Beyond the achievement of the dissertation objective, the following important contributions are made:

Beyond the realization of the dissertation objective, the following important novel contributions are made:

- The design, implementation and deployment of the proposed middleware solution, according to the defined system model and architecture, including the design and implementation of the related processing modules, materialized in two main components in the generic software architecture: the client-proxy and the cloud-server remote appliance.
- Implementation of hybrid support for SQL-querying (involving the design and implementation of SQL extended operations) involving Content Based Image Search/Retrieval functions, consisting of SQL operations combining indexing, search, and retrieval capabilities for multimodal confidential images, stored and maintained always encrypted in the cloud side.
- Deployment and validation of the integrated solution in a a real cloud-based pilot with a cloud-repository including a database as a service solution and a key-value store repository service.

Those contributions involve a set of novel techniques and an experimental benchmark evaluation that differentiates our approach form the related work and contributions found in the literature.

1.4 Report Structure

The document is structured in five chapters, this being the Introduction and first chapter. The remaining chapters are organized in the following manner:

- Chapter 2 presents the set of related work references, selected as relevant contributions in addressing work directions related to the research dimensions in the the dissertation objective. Considering such solutions, we overview their related foundations and techniques, identifying differences and drawbacks, taking into account our objectives and specific goals.
- Chapter 3 addresses the system and adversary models, as well as the software architecture of the developed solution.
- Chapter 4 presents the implementations aspects in terms of utilized software libraries and frameworks, and concrete implementation choices in order to materialize the architecture of section 3. Furthermore, it addresses the incremental prototype phases, the functionality and objectives of each, and an overview of their validation and experimental evaluation.
- Chapter 5 is dedicated to the experimental evaluation results of the different prototype phases, their careful analysis and conclusions pertaining the validation of the developed solution as a whole.
- Chapter 6 concludes the document with a balance of the contributions of this thesis work, and suggested directions for future work following the obtained results.

Chapter 2

Related Work

For the purpose of the thesis we organize the related work according to the following structure:
Section 2.1 covers the cryptographic primitives for Homomorphic Encryption and Searchable Symmetric Encryption. Section 2.2 presents Encrypted Database Systems, such as CryptDB [59] and Cipherbase [6], and a brief look at drawbacks and limitations, namely those particularly related with the requirements beyond our objectives. Section 2.3 addresses cloud-of-clouds storage systems meant to safeguard availability and integrity. Section 2.4 presents other data repository solutions, which are not directly related to databases, or in the case of Big Query [76] and Encrypted Big Query [30], are not full-fledged SQL systems.

2.1 Encryption Schemes

The essential construct behind computer systems and networks security, Encryption Schemes have long been used to encrypt data in order to guarantee its confidentiality. Among all the "conventional" algorithms that exist, two broad categories can be identified. Symmetric and asymmetric algorithms.

A **symmetric** scheme, such as the decades old and today considered unsafe DES [68], and the more modern variants of AES [69], have been tipically used to protect stored data. What characterizes a symmetric scheme is the secret key being used both by its encryption and decryption functions.

Asymmetric cryptography, also known as **public-key** cryptography is prevalent in critical services such as email and online banking. Besides capable of data encryption, these algorithms (e.g RSA [61]) allow for authentication of principals. They are characterized by the use of two keys, the public and the private key, the first being known to everyone and the latter only to a principal. If the public key is used to encrypt data, then the private key is needed to decrypt it, and vice-versa. Thus encrypting data with the public key ensures confidentiality, because the private key is needed, while the reverse serves as proof of authenticity, as only someone in possession of the private key could

have generated a valid ciphertext.

In recent years, new constructs have been developed and are subjects of ongoing research, to allow operations over encrypted data with a wide range of domains and applications. In this section we discuss the general concepts of Searchable Encryption, Homomorphic Encryption the subcategories of each, specific implementations and their use and practicality.

2.1.1 Searchable Encryption

Searchable Encryption is a relatively recent proposal [66]. It addresses a field of computations for the search and retrieval of relevant subsets of data from data repositories, comprising text documents, images, and possibly other media formats as well (e.g. a search engine). In the plaintext domain, different approaches exist to deal with different media formats. In this subsection we start by introducing approaches for both text documents and images, before moving on to the subject of Searchable Encryption for the ciphertext domain, divided in the key points of Public-Key Searchable Encryption (PKSE), Symmetric Searchable Encryption (SSE), and Content Based Searchable Encryption (CBSE).

2.1.1.1 Searching Text in the Plaintext Domain

When dealing with text documents, a query to perform a searh over a set of text data is specified through one or more keywords, and different relevance metrics can then be applied to obtain results. The simplest is the existence of those keywords in a document, in which case it is added to the result set. Scoring functions, such as the widely used TF-IDF [40], and BM25 [41] combine different relevance metrics to rank the documents and return a smaller and more relevant result set.

Extracting relevance metrics at runtime, however, is unfeasible in terms of performance. Best case scenario, the query performance will be linear with the number of documents and respective sizes. The solution is to preemptively extract and index the metrics, to achieve sub-linear performance. A well known and widely used indexing structures for text is the inverted list index [80], which maps keywords to a list of documents in which they occurr.

2.1.1.2 Content Based Image Retrieval

Performing image search can also be done with keywords in a similar manner to text, if images in the repository have associated tags [38], but this sort of approach can be fairly limited. Better results are attainable by providing an example image (query by example), and extracting image features to calculate distance functions [20], thus searching by similarity. This approach is known as Content Based Image Retrieval (CBIR).

Different features may be extracted from an image (e.g. color, texture, or other information). Features can be global (i.e. a single vector is used to represent the image) or local

(i.e multiple vectors represent different image regions). As an example, a global color histogram [71] counts the number of pixels of each color value, while a local counterpart [39] does the same for a region of the image only. Multiple local features can be combined to represent the entire image. Among local features, SURF [9] and SIFT [48] generally provide better precision than other options.

As with text documents, searching can be done by linearly comparing the extracted query features to the features of the entire image repository, in this case calculating distance functions between each. To achieve sub-linear performance, indexing structures are also required, and different structures are used for global and local feature vectors. M-Trees [34] are an example of an indexing method for global features. It divides the feature vector by regions according to a distance function, and indexes the regions in a dictionary. A common approach for local features is the Bag Of Visual Words [54]. An initial clustering phase builds a tree of visual points of interest. At index time, new images feature vectors are compared to the tree and represented as the closest leaf node. Finally, for each image an histogram of its leaf nodes is built, recording their frequencies.

Searchable Encryption schemes allow the described search and retrieval operations on encrypted datasets, and they have become an increasingly important topic and subject of research with the rise of cloud computing and storage. A generally unwanted, but necessary property at the core of Searchable Encryption is determinism. A deterministic encryption scheme always generates the same ciphertext for a given plaintext, and so it allows an attacker to establish certain patterns. In the context of Searchable Encryption, the commonly revealed patterns are:

- **Search patterns** - since queries themselves are encrypted deterministically, they are also uniquely identified. An adversary observing the issued queries is able to see repetitions of any query.
- **Access patterns** - result set elements are also uniquely identified, and the adversary can see which items correspond to each query.

Although potentially dangerous, the leakage of these patterns is an inevitability, due to the necessity of employing deterministic encryption. Plus, the actual risk is also highly dependent on the background information that an adversary possesses.

2.1.1.3 Public Key Searchable Encryption

One way to achieve Searchable Encryption is through Public Key Cryptography [2, 10, 14, 22, 58]. The advantage of this approach is that it naturally supports multiple writers to a repository through the public key, while a single principal can perform search operations using the private key. Some schemes also support multiple searchers [22, 58].

However, Public Key Searchable Encryption has two disadvantages. Asymmetric algorithms are considerably slower than symmetric counterparts, and these schemes in particular require multiple elliptic curve pairing operations [2, 14], or modular exponentiations [10, 22]. The other disadvantage is security. Since keys are public, and the scheme

is deterministic, it is extremely vulnerable to dictionary attacks, so much that query privacy cannot be guaranteed. From the related literature presented, only the algorithm by [58] is resistant to these attacks, since data is encrypted with private keys.

2.1.1.4 Symmetric Searchable Encryption

Symmetric Searchable Encryption (SSE) is based on symmetric key algorithms, as the name implies. When compared to Public Key Encryption Schemes, SSE achieves better performance due to their symmetric groundings, and its deterministic properties do not pose such a glaring security risk, as there is no public key. Thus they have received much greater research focus [8, 17, 19, 29, 32, 42, 43, 45, 52, 70], ever since its first conception by Song [66].

Exact-Match Approaches

Song's scheme allows exact-match linear search. It extracts all distinct words from the document, using standard delimiters. Then, words are randomly permuted, deterministically encrypted, and concatenated with some separator. Finally they are appended to the document itself, which is encrypted with a probabilistic scheme. In [59] a variant implementation of this scheme also padded all keywords to the same length. Therefore an attacker can learn the number of distinct keywords in a document, but not their order nor their frequency. The scheme provides linear search performance with the number of keywords.

Further research ensued, with proposals for the need of indexing structures [29], and achieving sub-linear search performance with indexes per keyword [19], albeit limited to static repositories (i.e. documents could not be added, dynamically updated, or removed). Kamara later improved upon this scheme with the support for dynamic updates of documents with two proposals. The first [43] introduced update pattern leakage. Upon insertion or dynamic update of a document in the repository, the attacker could learn the identifiers of its keywords, and match these with previously leaked identifiers. The second [42] proposal corrected this issue, at the cost of increased spatial and computational complexities.

More recent works explored performance and storage overhead tradeoffs [17, 32, 70], the most notable being the proposal by Cash et al. [17], which is a dynamic scheme with no update pattern leakage, and more efficient than the alternatives. It provides sub-linear search performance with the requirement that clients build and encrypt a feature index, and maintain some local storage linear with the unique number of keywords.

Ranked Search Approaches

As earier discussed, exact-match approaches for data retrieval are limited regarding the relevance of the returned results, as all matched documents are returned in no particular order. Scoring functions combine several relevance metrics to generate better result sets.

Unfortunately, extending SSE to support Ranked Search has seen limited progress. The first proposal by Wang [75] allowed Ranked Search for a single keyword, but to attain efficient performance, the frequencies of keywords were encrypted with an OPE scheme, and appended in the index entries. This approach reveals the order relations between plaintexts. A later proposal by Cao [16] overcomes the single keyword limitation, but at the cost of quadratic time complexities in both the client and server. A fuzzy keyword SSE scheme (i.e. search by proximity with gramatical error accountability), proposed by Kuzu [45], involves a lot of client computations and communication rounds with the server.

All three schemes, besides other shortcomings, leaked frequency patterns. Baldimtsi [8] proposed a solution that leaked no frequency patterns, but involved a secure coprocessor deployed to a cloud server. The coprocessor and cloud server performed the search collaboratively, while being unable to determine frequency patterns. The client maintained a partially homomorphic encrypted index. This approach is arguably more restrictive than the others previously mentioned, due to the need of deploying a secure coprocessor to the cloud.

Another limitation all of the Ranked SSE schemes share is the inability to handle dynamic datasets. The difficulty in developing such a scheme supporting dynamic update of documents, stems from the need to pre-calculate ranking scores, which must be refreshed upon a new insertion, update or removal.

2.1.1.5 Content Based Searchable Encryption

To provide as input for a query an entire document, instead of keywords, is what can be referred to as *query by example*. We've already addressed a form of *query by example* earlier in this subsection, namely Content Based Image Retrieval (CBIR). In the encrypted domain, CBIR solutions are based on Public Key Partially Homomorphic Encryption or Symmetric Searchable Encryption.

In SSE solutions, clients extract the necessary features from their images for indexing purposes. Both the images and the index are encrypted, usually with probabilistic and a combination of probabilistic and deterministic schemes, respectively. While these approaches attain fairly efficient search performance, they also share some disadvantages, namely:

- Either a trusted proxy is employed, or clients will be required to maintain local index structures for the dataset. The latter is undesirable as it burdens the clients with more processing and storage overhead. Furthermore, is a dataset is dynamic, multiple rounds of communication and even more processing still is needed to recompute the local index.
- SSE relies on deterministic tokens to achieve practical preformance, which reveal search, access, similarity (which documents are similar to the given query document) and update patterns (which documents are similar to a newly added one).

Public Key Partially Homomorphic Encryption (PKHE) approaches (see subsection

2.1.2 on homomorphisms) are an alternative to SSE schemes, based on homomorphic constructs such as Paillier [57] or ElGamal [23]. In CBIR, clients encrypt images pixel by pixel, upload them, and all processing related to construction and maintenance of indexing structures is performed by the server. Nonetheless, PKHE suffers from its own disadvantages, namely much higher time and space complexities when compared to SSE. As an exmaple, the approach by Hsu [35] expanded a pixel 24 bit representation to 2048 ciphertext bits. Thus, PKHE offers limited practicality.

All advantages and disadvantages considered, SSE approaches seem more practical and are more widely adopted in the field. One particular SSE scheme, IES-CBIR [24], stands out as an interesting solution that addresses some of the common issues in SSE. We present a more detailed overview of IES-CBIR, in regards to its objectives and what it accomplishes compared to other state of the art CBIR solutions.

2.1.1.6 IES-CBIR

IES-CBIR is an Image Encryption Scheme with Content Based Image Retrieval properties. Its premise is that computations regarding the processing of images, training tasks, and index construction, maintenance and storage should be moved to the cloud, and spare the client from such expensive operations. Mobile clients especially benefit from this design, as their resources are limited.

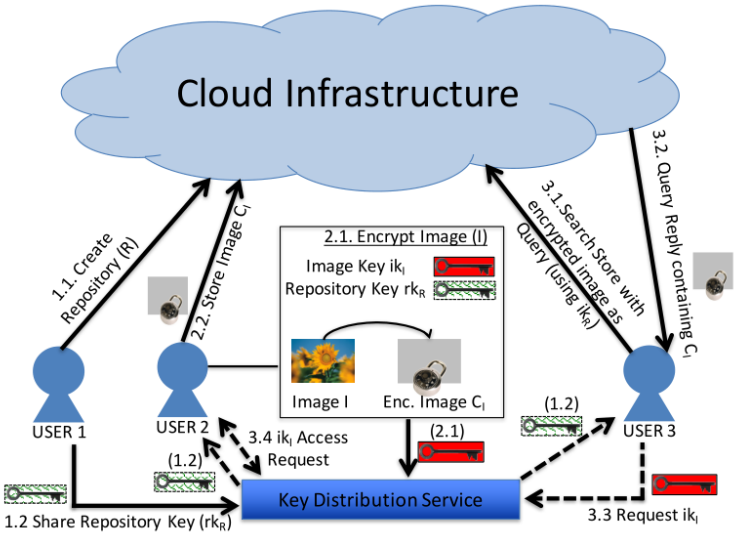


Figure 21: The IES-CBIR system model.

The IES-CBIR system model, seen in figure 21, encompasses multiple clients/users and a cloud. Users create repositories in the cloud, to where images are then outsourced. In the scope of a repository, images can be dynamically added, updated and searched, by any user with access to the repository key r_k. Adding, updating, encrypting or decrypting images further requires an image key i_k, which is image specific. The model also features a key distribution service, but that specific component is considered to be orthogonal.

To take advantage of determinism without compromising confidentiality, the authors of IES-CBIR made the following insight. Image data is composed of color and texture information, both can be independently encrypted and either one is eligible for processing. While each is generally ambiguous without the other, texture is considered more relevant towards object recognition. And so, texture is protected with probabilistic encryption, while color data is deterministically encrypted and processed upon.

To allow image processing to be naturally performed, color values are encrypted by mapping each of its HSV channel values to a new value in the same range, using the repository key r_k. This key is composed of three independent permutations, for each HSV channel, rk_H, rk_S and rk_V. Texture is then probabilistically encrypted, permutating pixels positions by shifting rows and columns using an image key i_k. The end result is an image which suffered no ciphertext expansion, with the same width and height as the original, and color values mapped to new values in the HSV space.

When the cloud receives an image encrypted in the aforementioned manner, it can naturally process and index its features, represented by color histograms. A Bag Of Visual Words model [54] is used, to build a vocabulary tree and an inverted list index for the repository.

IES-CBIR assumes an *honest but curious* system administrator as its main adversary. This is a passive adversary, who has access to all storage in disk and RAM in the cloud domain, as well as data circulating from and to the cloud. He is not assumed to actively disrupt the integrity or availability of the cloud services and stored data.

When compared to the two most relevant state of the art alternatives, namely an SSE scheme by Lu et al. [49] and a PKHE approach by Hsu et al. [35], IES-CBIR displays comparable security guarantees, while relieving the client from the maintenance of a local index (and as a direct result, bandwidth usage) necessary in the SSE scheme, and offering much better space and time complexities than those of the PKHE scheme.

2.1.2 Homomorphic Encryption

In this subsection we will present an overview of Homomorphic Encryption algorithms and methods related work. We begin by presenting the notion of Full Homomorphic Encryption, namely the recent discoveries by Gentry, following our discussion with Partial Homomorphic Encryption methods, sometimes also referred to in the literature as "Somewhat Homomorphic Encryption" schemes).

2.1.2.1 Fully Homomorphic Encryption

Given plaintexts $p_1,...,p_n$ and their encryptions, a Fully Homomorphic Encryption Scheme can yield the encrypted result of any function $f(p_1,...,p_n)$, without revealing the plaintexts, intermediate values, or details about f itself. Furthermore, the key is not required to perform this operation.

The concept has existed for some time [55, 61, 62], yet it was not until recently that Craig Gentry proposed the first implementation of a Fully Homomorphic Scheme on his PhD thesis [26]. By supporting the basic logical operations, it was possible to derive the necessary circuits to perform arbitrary computations.

In his thesis introduction, Gentry presents a "somewhat homomorphic" toy scheme that performs addition and multiplication as the starting point for the construction of the Fully Homomorphic Scheme. In this initial construct, each ciphertext has a noise parameter, and it can be decrypted so long as the noise is below a certain threshold. For newly obtained encryptions the noise is relatively small, but adding or multiplying values results in a ciphertext with increased noise parameter. Therefore, chaining operations will eventually return a ciphertext with too much noise, past the point where decryption is possible. The toy scheme is illustrated below:

- We choose an odd integer p as the secret key.
- A bit b is encryted to an integer c through $c = b + 2x + kp$, where:
  x is a random integer in the domain $[-(p-1)/2, (p-1)/2]$, and
  k is a random integer.
- Decryption is performed by making $b = ((c \bmod p) \bmod 2)$, where:
  $\bmod p$ eliminates the kp term;
  $\bmod 2$ removes the $2x$ term.

Given the domain of x, $c \bmod p = b+2x$, which is the aforementioned noise component. Decryption will only work as long as it is kept below p. We can ascertain the homomorphic properties of the scheme under that condition. For two ciphertexts $c1$ and $c2$, let us first add, and then multiply them:

$$c1 + c2 = b1 + b2 + 2(x1 + x2) + (k1 + k2)p = b1 \oplus b2 + 2x + kp$$

As we can see, by adding $c1$ and $c2$, we are returned the encrypted result of the binary addition (xor) of the respective plaintexts. As for multiplication:

$$c1c2 = b1b2 + 2(b1x2 + b2x1 + 2x12x2) + kp = b1b2 + 2x + kp$$

which gives us the encryption of the plaintexts multiplication.

For both operations, as long as $b1 + 2x1 + b2 + 2x2$ and $(b1 + 2x1)(b2 + 2x2)$, respectively, are kept within the $[-(p-1), (p-1)]$ bounds, decryption is possible. It is also worth noting that multiplying ciphertexts will inscrease the noise more quickly than addition.

Gentry's actual scheme, based on ideal lattices, still suffers from the noise growth issue. To overcome this limitation, after each operation the ciphertext is refreshed by obtaining a new encryption of the same value, with reduced noise, as follows:

- **1.** Encrypt the secret key sk_1 from the original key pair, with a public key pk_2 from a second key pair, obtaining sk'_1.
- **2.** Encrypt the ciphertext c_1 with pk_2, obtaining c'_1.
- **3.** Evaluate the decryption function over c'_1 homomorphically, which gives us a new ciphertext of the same value, with reduced noise.

The depth of the decryption function's circuit must not surpass the capability of the somewhat homomorphic scheme. Through a sufficiently efficient decryption circuit, a Fully Homomorphic Scheme is achieved by chaining operations with ciphertext refreshes. Sadly, the scheme has prohibitive time complexities, and is inadequate for interactive applications.

2.1.2.2 Partially Homomorphic Encryption

Given the inefficiency of Fully Homomorphic Schemes, another approach has been to develop algorithms that display homomorphic properties for one specific operation or a subset of operations. Partially Homomorphic Encryption Schemes (PHES) are reasonably efficient, and so provide a feasible alternative.

Although in related literature Partially Homomorphic Encryption Schemes are usually understood as schemes with homomorphic properties regarding arithmetics, throughout the rest of the document we refer to them as algorithms enabling any operation over ciphertext. To clarify, any construct that allows the following reasoning:

Given a plaintext p and a computation f, $f(p)$ yields the result r. A PHES provides the f equivalent operation f', and for a ciphertext c of p, $f'(c)$ produces r'. Upon decryption of r', we obtain r.

We now summarize some of the most relevant partially homomorphic constructs, and how they can be useful in the context of a software solution where data is outsourced to an untrusted remote server to which we want to delegate operations on said data, while protecting its confidentiality. We also discuss the patterns revealed by some of the constructs, and how to mitigate their potential vulnerabilities.

Comparison of Encrypted Data:

To enable comparison of encrypted values, an encryption scheme displaying deterministic properties (DET) is used. If a ciphertext $c1$ of a plaintext value $p1$, equals another ciphertext $c2$ of plaintext $p2$, the relation $p1 = p2$ can be established without knowledge of the plaintext values.

For this very reason, determinism is a generally unwanted property, but needed here to allow the comparison. Symmetric schemes based on pseudo-randomness are usually employed, as implemented in [59] and exemplified in table 21.

Plaintext size (bits)	Used Scheme	Description of use
≤ 64	Blowfish	When plaintext size is smaller than 64 bits, add padding
> 64 and ≤ 128	AES	When plaintext size is smaller than 128 bits, add padding
> 128	AES	Use a fixed initialization vector to achieve a deterministic encryption. To avoid equal ciphertext prefixes of different plaintext values with equal first block, encrypt twice, back to front and vice versa

Table 21: Encryption Schemes used in DET

Order Preserving Encryption:

An Order Preserving Encryption (OPE) scheme maintains, for a given dataset, the order relations from the plaintext values in the ciphertexts. If plaintexts $p1$ and $p2$, such

that $p1 < p2$, are respectively encrypted to $c1$ and $c2$, then $c1 < c2$. OPE allows addressing inequality operations, sorting and range queries. Like DET, OPE is deterministic, thus it is at least as vulnerable as DET. Additionally, since it reveals order relations between encrypted items, it is even more vulnerable.

Boldyreva [13] presents a method to build an OPE scheme that reveals no more than order relations. Considering a plaintext space of M elements, we could build a table of M randomly extracted elements from a ciphertext space N, where $N < M$, and then order the table. To encrypt a plaintext x, we'd simply obtain the $x - th$ element from the table. This is of course very impractical, due to the size such a table would take.

Ideally, we would like to obtain only the necessary subset to encrypt and decrypt elements as needed. A function following a hypergeometric distribution, that could generate a pseudo-random value y for every x, given a key k, would be perfect, but there is no known efficient such function. However, there is a known method to calculate the inverse, given y, generate x. With this method, we can recursively generate elements from M, performing a binary search until we reach x.

Linear search over encrypted text data:

This particular form of homomorphism has already been addressed in subsection 2.1.1.4. In [59], Song's scheme is adapted to provide linear search over encrypted text.

Addition over ciphertext and the Paillier cryptosystem:

The Paillier cryptosystem [57] supports addition of values which are encrypted. The encrypted modular addition of two plaintext values, $p1$ and $p2$, is obtained in this scheme by multiplying their ciphertexts, $c1$ and $c2$. The homomorphic property is expressed in the following equation:

$$E(p1 + p2) = E(p1).E(p2) = c1.c2$$

We note that since the addition operation is achieved through multiplication of ciphertexts, it is also possible to multiply a single encrypted value by a plaintext constant a, through exponentiation of the ciphertext to the power of that constant:

$$E(p.a) = E(p)^a = c^a$$

Multiplication over ciphertext: Unpadded RSA and **ElGamal** [23] are public-key cryptosystems, both supporting multiplication. Like in Paillier, encrypted values are multiplied, but the resulting ciphertext corresponds to the encryption of the respective plaintext values multiplication.

$$E(p1.p2) = E(p1).E(p2) = c1.c2$$

As Paillier allows multipliction by a plaintext constant, ElGamal supports exponentiation of a single value to a constant.

$$E(p^a) = E(p)^a = c^a$$

An important difference between Unpadded RSA and ElGamal is that, while Unpadded RSA is a deterministic scheme, and thus vulnerable to chosen plaintext attacks, ElGamal is probabilistic and resistant to such attacks.

Joining encrypted data:

Join operations are common in relational databases. They are naturally supported if DET is used, but the same key must be used for both columns. However, columns may be encrypted with different keys, and in this case additional work is required.

In [59], this is dealt with by using keyed hash functions that can adjust their hashes to a different key, without accessing the plaintext. An auxiliary field containing this hash is concatenated with the corresponding DET field. The DET field cannot be used to compare against a column encrypted with a different key, but if requested, the auxiliary hash field can be adjusted to use the same key as the auxiliary field in the other column, thus allowing the comparison.

Random encryption schemes and their role: The presented partially homomorphic constructs can introduce vulnerabilities, due to their inherent deterministic properties. To mitigate this risk, the ciphertexts obtained from those algorithms may be encrypted with a probabilistic scheme, such as AES with CBC operation mode, resulting in an onion encryption model. The outer layer is "removed" by decryption, operations are executed on the homomorphic ciphertexts, and re-encrypted afterwards. With proper block modes and padding, this offers the strongest security guarantees.

2.1.3 Summary

The discussed Encryption Schemes cover a wide range of domains. Fully Homomorphic Encryption Schemes would be perfect to delegate computations on encrypted data to an untrusted remote party, but performance stands as an obstacle to its adoption for interactive applications, and in particular for our proposed solution. Even before Gentry's proposal [26], already alternatives were being developed to address computations on encrypted data.

Since Song introduced exact-match linear search over encrypted text [66], many other constructs have come about to address the extraction of relevant subsets of text data in the encrypted domain, both in form of Public Key Searchable Encryption [2, 10, 14, 22, 58] and Symmetric Searchable Encryption [8, 17, 19, 29, 32, 42, 43, 45, 52, 70]. SSE schemes have gained more traction, being a more secure alternative despite their deterministic properties, which are exploited to naturally enable the use of indexing structures.

SSE has also seen development for Content Based Searchable Encryption [24, 49]. While pattern leakage and indexing structure overheads on clients are an issue on most SSE schemes, Public Key Homomorphic Encryption approaches [35] don't have these problems. Still, they have higher time and space complexities compared to SSE. IES-CBIR [24] is particularly interesting to our solution, with comparable performance to other state of the art CBIR solutions, while freeing clients of local index structure maintenance.

Partially Homomorphic Encryption schemes for order relations [13], addition [57] and multiplication [23], equality comparison and joins as implemented in [59] are sufficiently efficient alternatives to FHES, each addressing specific operations. For our purposes of a solution supporting multimodal text and image queries, Partially Homomorphic Encryption schemes can be leveraged alongside IES-CBIR.

2.2 Encrypted Databases

Encrypted Database Systems aim to preserve the confidentiality of data which is outsourced, for instance, to a cloud service. To achieve this, outsourced data (or part of it) is encrypted and queries are performed over encrypted data, so that plaintexts are not accessible from the server side at any moment. Such systems are addressed in this section, namely TrustedDB, CryptDB and Cipherbase. To provide the execution of operations over encrypted data, these solutions can use Partially Homomorphic Encryption Schemes.

2.2.1 TrustedDB

TrustedDB provides a full-fledged DBMS solution with confidentiality guarantees for outsourced databases. It employs the use of Secure Co-Processors (SCPUs), which are trustable hardware modules running verified code, and assumed to be tamper resistant. In simple terms, SCPUs can be thought of as black boxes that perform specific functionalities, and do not allow external access to their memory state.

In the TrustedDB paper, three deployment scenarios are considered for outsourcing an integer dataset to a cloud provider, which is then subjected to aggregated queries involving sums. The deployments are each evaluated in terms of their per-transaction monetary costs of the CPU cycles and data transfers. The three scenarios are:

- **(A)** the dataset is encrypted before being outsourced to the server. Querying the dataset entails transferring it back to the client, where it is decrypted before search operations can be performed upon it. Potentially, large subsets or even the entire dataset must be transferred back.
- **(B)** the server side employs homomorphic primitives (such as Paillier) capable of performing computations directly over the encrypted dataset, in this case arithmetic sums of integers.
- **(C)** data is either classified as public or private, the latter being encrypted such that only the client or the SCPU can decrypt it. When a query is issued, a plan is generated in order for public data to be processed in the untrusted cloud hardware, while private data is decrypted and processed inside the SCPU, and its result set encrypted and returned.

In scenario **(A)**, the networking costs alone for transferring just a subset of the data becomes extremely expensive. In **(B)**, the used cryptographic primitives incur a large

number of CPU cycles, thus becoming significantly expensive as well. Lastly, the use of SCPUs in **(C)** features amortized costs lower than **(A)** and **(B)**.

This evaluation, however, focuses on the cost-efficiency metrics for a reduced set of operations (aggregated SUM queries). Furthermore, the TrustedDB architecture has a number of drawbacks. While the server runs a standard SQL engine, the SCPU runs a heavily modified SQLite engine. Due to its memory limitations, it pulls data from external storage (the server's untrusted hardware storage) but cannot hold great data volumes at once. Performance is inevitably hindered, thus while TrustedDB offers good confidentiality guarantees, scalability is a major issue.

2.2.2 CryptDB

CryptDB [59] is a DBMS, aiming to protect data confidentiality in the context of a remote database server to which the data is outsourced. Particularly, CryptDB addresses two specific threats, the "honest but curious" system administrator, and an adversary who compromises the entire application and server infrastructures.

To protect data confidentiality, a number of partially homomorphic constructs are implemented in CryptDB. SQL queries are rewritten, so that table and column names, tuple values and constants are encrypted, and typical SQL operations are performed directly over ciphertext. As tuples may potentially be involved in different queries (e.g. equality select, range select, updates of numeric values), requiring different homomorphisms, CryptDB implements an onion encryption model (depicted in figure 22).

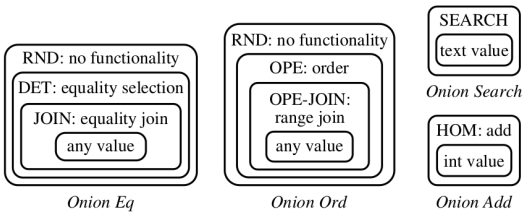


Figure 22: Onion encryption layers and the operations they provide.

For each class of computation (equality, order, text search and addition) there is a corresponding onion. Data items are encrypted in one or more onions, where applicable, thus an operation requiring a certain computation class will use the corresponding onion. The Eq and Ord onions have multiple layers, with varying levels of security, and queries may adjust the onion by "peeling" (decrypting) the layers until they reach the one needed to support the operation to perform. The necessary keys to peel or encrypt the onion back are provided to the server, but the key for accessing the plaintext is never accessible to the DBMS.

Table 22 below presents an overview of the partially homomorphic algorithms comprised in the onion model implemented in CryptDB, regarding their supported operations.

Designation	Description of use
DET	Supports Select operations with equality predicates, Join operations on columns with the same key, Group By, Count, Distinct, etc.
OPE	Used to perform range selects, Order By, Min, Max, Sort, etc.
SEARCH	Implementation based on the Song et al. algorithm, for linear exact-match search over text. Useful for queries involving operations suck as MySQL's LIKE.
JOIN and OPE-JOIN	JOIN and OPE-JOIN offer support for joining table columns by equality, and by order relations, respectively.
HOM	Allows operations involving addition, namely Sum aggregates, incremental Updates of values (e.g. SET x = x + 4).
RANDOM	RANDOM is not a partially homomorphic scheme. It is simply a probabilistic encryption algorithm, not meant to support any operation over ciphertext. Its utility in CryptDB is to encapsulate ciphertexts of other algorithms, to provide maximum security.

Table 22: Encryption Schemes implemented in CryptDB

As mentioned in the table overview, RANDOM is meant to provide maximum security by encapsulating ciphertexts produced by the inner cryptographic constructs. As some of the partially homomorphic algorithms reveal information (e.g. OPE reveals order relations between values), this outermost layer is useful when data is not being operated upon.

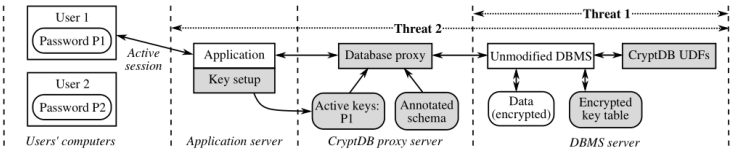


Figure 23: The CryptDB architecture model.

The CryptDB architecture model is displayed in figure 23). Components in white denote those found in a normal DBMS and application environment, while shaded components are those added by CryptDB. A user first logs in to an application using his/her password, and from this password all encryption keys for that user are derived. As long as the user maintains an active session, its encryption keys will remain present in the active keys dataset used by the database proxy. Issued queries are intercepted by this proxy, and the data (names and values) encrypted using the active keys according to an annotated schema. If a query requires the onions to be readjusted, the DBMS is provided with the necessary keys to perform the adjustment through a User Defined Function (UDF) implementing the cryptographic algorithms for such purposes. Once an onion is at the desired layer, the query is naturally performed by the unmodified DBMS.

Figure 23 also illustrates the range of action of two threats, here described:

- **Threat 1** or the "honest but curious" system administrator, is a passive adversary targeting data confidentiality in the DBMS server. It has full access to the DBMS machine, can snoop storage (including RAM), and observe all traffic arriving and departing from the DBMS machine, but will not tamper with the integrity or availability of the data and the computations. Against this adversary, CryptDB guarantees the confidentiality of data content (names and values of tables, columns, tuples), although the overall table structure and numbers of rows, columns and data size

are revealed.

- **Threat 2** comprises arbitrary threats not included in the first threat. This second adversary has a wider range of action, the DBMS, proxy and application servers. Therefore it has access to the active keys in the proxy server, and can potentially access the entire database. To mitigate this problem, different users should have different passwords (and keys), thus data belonging to a user who is not logged in is protected because the adversary doesn't have that user's keys.

Efficiency wise, CryptDB attains good performance, introducing an overall overhead of about 26% in the TPC-C benchmark evaluation, and 14.5% for phpBB, when compared to the same tests with standard MySQL. Unfortunately, full SQL support is not achieved. Of a large trace from a production MySQL server, CryptDB supports 99.5% of the operations.

2.2.3 Cipherbase

Cipherbase [6] is a DBMS compliant system with full support for SQL. One of its main features is the inclusion of FPGAs (Field Programmable Gate Array) to implement a Trusted Machine (TM) alongside an Untrusted Machine (UM), in a similar fashion to TrustedDB. Both the TM and the client are trusted, while the UM is not, as the name implies. The main concern of Cipherbase is data confidentiality, assuming an adversary that has super user privileges over the UM and can monitor its disks, memory and network traffic. The adversary cannot see the internal state of the TM, nor tamper with it.

While TrustedDB has a loosely coupled architecture, where all encrypted data is processed in the TM, Cipherbase features a tightly coupled design. Partially homomorphic, and other property-preserving encryption schemes are supported in the UM. Fully homomorphic encryption is simulated in the TM with its implemented primitives, over non-homomorphic schemes. When a primitive for a corresponding operation is invoked in the TM, the data is decrypted and processed, the results encrypted and returned. The workload of a query can also be distributed between UM and TM, so that logic parts of the query that require access to plaintext of sensitive data, are performed in the TM, and those who do not, are performed in the UM. The UM, being more powerful, is explored to the fullest extent possible.

The bottleneck of Cipherbase is the TM. Its limitations might negatively impact the scalability of the system, depending on the specified confidentiality levels.

2.2.4 Inference Attacks

A number of attacks against Encrypted Database Systems has been identified and tested. In [53], a CryptDB system for a database of medical records from all hospitals in the US was subjected to Inference Attacks. Two types of attacks against DET columns, and other two targeting OPE columns, disclose an alarming quantity of data.

The threat model assumes an adversary with access to the server, who can see the ciphertexts but cannot issue queries. The database is in steady state, so queries issued by the application server will peel the onions to the lowest layer needed for the query to execute. The adversary also resorts to auxiliary data, such as application details, public statistics or information gathered from previous attacks.

The attacks to DET columns recovered the mortality risk for 100% of the patients for 99% of the hospitals. Deaths were disclosed for 100% of patients and 100% of hospitals. Other columns had a lower, but still relatively high percentage of recovery.
Attacks to OPE columns also obtained 100% of the mortality risk, for 100% of the hospitals, but additionally 80% of patient records for 95% of hospitals, 99% for the age of patients in 82.5% hospitals, and overall very high success in other columns.

In the light of these results, the author concludes that EDBs are not fit for the purpose of Electronic Medical Record databases. Not only were the attacks highly successful, the adversary was actually very restricted. A stronger adversary would potentially obtain even more information. However, the success of such attacks is dependent on the background information the adversary already has. An attacker with no domain knowledge would certainly not disclose data so easily.

2.2.5 Summary

Of the three solutions presented (TrustedDB, CryptDB and Cipherbase), TrustedDB is clearly limited in most respects, but especially in terms of scalability. CryptDB and Cipherbase are more solid in terms of functionality, but are still incomplete for our purposes. Neither one provides multimodal search mechanisms over encrypted text and images, namely proximity based searching functions for image retrieval (CryptDB only implements SQL-enabled operations with the onion encryption facility for text data, which is fairly limited).

Another noteworthy criticism is the vulnerability of CryptDB to resist against Inference Attacks identified in 2.2.4. Cipherbase can be more resistant to these attacks, however at the cost of damaging its scalability and efficiency by executing more operations in the provided TM solution. Anyway, despite recognizing this last fragility, we considered it to require some background knowledge, thus it is outside the scope of the thesis and is not addressed in the solution.

2.3 Dependable Data Storage Solutions

Related work in this section does not directly involve databases. Instead, these are solutions concerning availability and integrity of data. They share the common approach of replicating or partitioning data across a cloud of clouds, so not only do they offer the two previous guarantees, the vendor lock-in issue is also addressed. By keeping replicas

of his data in different providers, the client can dismiss unsatisfactory services with no backlash.

2.3.1 HAIL

Hail [15] (High-Availability and Integrity Layer) is a proactive distributed storage system. Files are stored in a number of clouds, and periodically verified. If corruption is detected, a file is properly restored. To demonstrate to a client that a file is intact, uncorrupted, and can be retrieved, Hail makes use of a Proof of Retrievability (POR) challenge-response protocol. Responses to POR challenges are efficiently computed and very compact. Hail only offers protection of static data, but the authors assume dynamic data support as a future direction.

Hail is intended to resist a mobile adversary, one that can corrupt the full set of n servers over time. However, he can only corrupt a bounded number of b servers in each time step (an epoch). Clients perform integrity checks in each epoch, and restore corrupted files in faulty servers. Given the constraint on the adversary, he would need $\lceil n/b \rceil$ epochs to successfully corrupt a file beyond recovery.

A simple approach to achieve availability would be to replicate a file F in each server. Instead, to reduce storage overhead, Hail distributes blocks of F to servers, and makes use of error-correcting codes (or erasure codes). For a total set of n servers, there are l primary servers and $n-l$ secondary servers. F is split into l segments, which are distributed to the primary servers. Each of the segments is encoded with an error-correcting code called the server code, protecting that segment against a fraction of corrupted blocks. Then, blocks at position j of each segment are grouped to form rows, which are also encoded, resulting in what is called the dispersal code. The dispersal code is distributed across the secondary servers.

Through the described approach, it can be verified if the rows of F are self consistent. But to verify that they actually match the original contents of F, Hail implements a construction its authors call Integrity-Protected Error-Correcting Code (IP-ECC). The idea behind IP-ECCs is to integrate Message Authentication Codes (MACs) in parity blocks of the dispersal code. The integrity of multiple positions of F can then be verified simultaneously by computing a composite MAC, only obtainable from valid MACs for the individual blocks.

2.3.2 DepSky

DepSky [12] provides a storage system across a cloud of commercial clouds. Its main focus is to improve availability, integrity and confidentiality, and to prevent vendor lock-in. Data replication in different clouds for availability incurs additional costs, so reducing these costs is also a concern.

Storage clouds are assumed to not support execution of client code. DepSky algorithms are implemented in the clients as software libraries, and communication with the

clouds is performed through Remote Procedure Calls to their standard interfaces. Due to heterogeneity of those interfaces, the system's data model consists of three abstraction levels, pictured in Figure 24.

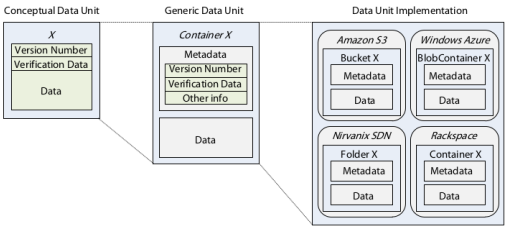


Figure 24: DepSky Data Model

The Conceptual Data Unit (left) constitutes a basic storage object, which has a version number, verification data such as a cryptographic hash, and the data itself. The Generic Data Unit (middle) defines an abstract cloud container holding several files, each with its multiple versions. Finally, the Data Unit Implementation (right) implements the cloud specific containers (Buckets, Folders, etc.).

Clouds are also assumed to fail in a Byzantine way [46]. In this fault model, failures might occur in clouds, preventing the others from reaching consensus. Failures include loss, corruption, non-authorized creation or disclosure of data. DepSky implements Byzantine Fault Tolerant protocols requiring $n = 3f + 1$ storage clouds, with at most f faulty clouds. The protocols provide read and write operations for clients.

DepSky-A (Available DepSky) is the first protocol. Its write algorithm replicates the data across a minimum of $n - f$ clouds, then writes the respective metadata. The read algorithm gets the metadata from $n - f$ clouds, checks the version numbers and cryptographic hashes from each and then gets the most recent version. Having stored the data in at least $n - f$ clouds, the read operation will succeed since $(n - f) - f > 1$.

DepSky-CA (Confidential and Available DepSky) is the more complete protocol. It is similar to DepSky-A, but encrypts data with a key generated by a secret sharing scheme [65]. Each cloud, along with the ciphered data, receives only a share of the secret. Authorized clients can obtain all the shares needed to rebuild the data. Data is also encoded with an information-optimal erasure code algorithm, which reduces its size. This last step decreases monetary costs.

2.3.3 Summary

The presented cloud enabled data storage systems offer integrity and availability guarantees. The DepSky [12] system is an interesting reference providing confidentiality, integrity and availability, as conjugated properties in a dependable multi cloud storage architecture, neutral to the integration of heterogeneous cloud storage solutions, as currently offered by well known providers. The HAIL solution [15] is more limited, focusing on a single storage cloud and integrity guarantees.

Both solutions can inspire the design of dependability properties for cloud storage data back-ends, particularly addressed by key-value object stores. Such dependability properties are not addressed by the studied Encrypted Database solutions, more directly related to our objectives and contributions. However, these solutions don't address the specific requirements for our goals, not being developed to integrate SQL querying enhanced with multimodal indexing, search and retrieval capabilities. Another drawback is that those solutions are not designed, developed and evaluated for Cloud DBaaS deployments, an important issue in our case.

Although not our top priority, a possible future direction envisioned from our contributions could be to provide a transparent platform (that would not require changes to the DBMS) to ensure integrity and availability properties for our solution, using SQL enabled replicated data stores.

2.4 Other Cloud-based Data Repository Solutions

In this section are presented a number of cloud-based systems, which provide functionalities different from those of a DBMS or perform queries across the data repositories in an SQL-like fashion, but do not constitute a full fledged DBMS. Among these solutions we summarize the following: Google's Big Query [76], Encrypted Big Query [30], SQL-enabled RAMCloud [56], Partiqle [72] (as representative references for SQL-enabled solutions built on top of elastic key-value stores), TrustedSKY [63] and TMS [64] (as representative solutions providing dependability properties for specific applications).

2.4.1 Google's BigQuery

Google's BigQuery [76] is a Database as a Service platform meant to provide efficient querying over large quantities of data, often referred to as Big Data. Users create **Projects**, which are top level containers. Before data can be loaded, the user must create a **Dataset**, a container for **Tables**, which in turn will contain the data. To operate upon the data, the user specifies **Jobs**.

Tables in BigQuery are described by a schema, with column names, their data types and additional information. Records can contain nested and repeated columns. Tables are append-only, and can be generated from various sources. Data is loaded from a CSV or JSON file, copied from other tables, defined over a file in cloud storage, or generated from a query result. Users can also create empty tables from the BigQuery API or command tools. Loaded data must match the defined schema, which can be updated. Since tables are append-only, data cannot be updated nor deleted.

Jobs manage asynchronous tasks, and multiple jobs can be executed concurrently. There are four different types of jobs, according to their tasks. Loading data into a table, running a query, exporting data to cloud storage and copying tables.

Queries support a subset of SQL operators. As mentioned, tables are append-only, so the syntax is very much limited to SELECT clauses, with JOINS, ORDER BY and few more. It does however feature a rich set of functions, for Internet Protocol addresses, JSON expressions, mathematical functions, regular expressions, and others.

2.4.2 Google's Encrypted Big Query

Encrypted Big Query [30] is an extension to Big Query, but supports only a subset of the query types from Big Query. Users define schemas for tables, where a new field "encrypt" is now present for columns, for an encryption scheme to be specified. Examples of available schemes are "homomorphic" and "searchwords". Input data is encrypted before being uploaded, according to the schema, with a key stored in a local key file. When querying encrypted columns, data is downloaded to the client and decrypted with the same key from the key file.

2.4.3 RAMCloud

RAMCloud [56] is a general purpose distributed storage system. It keeps all data stored in DRAM (Dynamic RAM), and backup copies in secondary storage. It provides very low latency, for large scale systems. In a pictured scenario of clusters growing to at least 10000 servers, RAMCloud offers 1 PetaByte of storage, with access times 50 to 1000 times faster than what are common storage systems today.

Previous solutions using DRAM for general purpose distributed storage have been used as caches for applications such as databases, but developers needed to manage cache consistency. Performance was also an issue due to cache misses and backup storage overheads. Recent work directed towards future implementations of databases is in motion [44, 47], but whether a full SQL/ACID relational database can be implemented in RAMCloud remains an open question.

2.4.4 Partiqle

Partiqle [72] is an SQL execution engine which has a key-value store as the underlying storage, allowing it to benefit from the elasticity of cloud services. Concurrency control is maintained by a cluster of query execution nodes, through atomic operations on the key-store values, to provide strong consistency beyond SQL transactions with linearizability semantics on top of the base key-value store (already enabled as a geo-replicated storage service).

2.4.5 Trusted Sky (TSKY)

Trusted Sky [63] is a middleware system, which intermediates final applications and multiple storage clouds. Its architecture contemplates indexing, distribution and replication of data across multiple storage clouds, in order to guarantee resistance to faults

and attacks on the cloud service providers. Cryptographic mechanisms are used for data encryption and authentication. Adversaries in the threat model are unlikely to coalesce, nevertheless TSKY defines a Byzantine model where a collusive attack is supported to a threshold.

The system can operate in three variants: a local mode in which the middleware is running on a local trusted machine; a proxy mode, where it is executed on a trusted server; a cloud mode, where the middleware runs in an untrusted domain.

To evaluate TSKY, an email repository system called TSKY-TMS (Trusted Mail System) was developed.

2.4.6 Trusted Mail Service (TMS)

Today, e-mail is critical for both companies and individuals, and security of such services is increasingly relevant. TMS [64] proposes a model where data is replicated across multiple clouds to guarantee availability and reliability. Since these clouds are untrusted, confidentiality and integrity are also addressed.

A middleware component is placed between the Mail User Agent (MUA) and a cloud of clouds. It provides a transparent interface for MUAs, and handles the interaction with the clouds. Data is indexed and encrypted before multiple replicas are sent to different clouds. A reference index maps message ids to tokens containing a cloud reference, a pointer to an object in the cloud, and the encryption key, stored in the middleware server. An index for multi keyword search, and a boolean index for searches over header fields are also kept.

2.5 Critical Analysis

Regarding Encrypted Databases, CryptDB [59] and Cipherbase [6] are state of the art solutions. Neither offer the possibility of searching encrypted images. We favor CryptDB for not requiring a FPGA device on the server side, an imposition upon the cloud providers, particularly in the provisioning of DBaaS solutions. Our approach will be to extend a CryptDB-like platform with IES-CBIR [24] for multimodal proximity search of images, assuming a similar adversary model to those of CryptDB and IES-CBIR literatures. We also note that neither CryptDB nor Cipherbase have presented any experimental test benches of a deployment as a cloud provided solution.

An alternative for our developed solution is to replace the DBMS component with a Big Query service [76], while maintaining support for the same features of Content Based Image Retrieval (a key challenge and criterion in our approach). From the baseline achieved by our contributions, it is possible to address enhanced availability and integrity support by evolving the cloud data store back-end. Although we won't address this as a requirement of the current objectives in the dissertation, we intend to discuss this

research direction by sketching a possible design to implement such a solution in the future.

Chapter 3

System Model and Architecture

As stated before, the thesis goal is to design, implement, and evaluate a Middleware solution providing client applications with privacy enhanced access to remote cloud database backends. The Middleware offers support for SQL operations, as well as image storage, searching and retrieval, and multimodal SQL and image searches upon the outsourced backends, where data is kept encrypted at all times.

The proposal aligns a client/client-proxy/server-appliance software architecture, with the Middleware solution mainly reflected in the client-proxy and server appliance components. In order to safeguard the confidentiality of sensitive information (database data and indexed images), a set of cryptographic constructions involving searchable encryption primitives for image-similarity searches and partial homomorphic encryption constructions are used. This cryptographic support is at the core of the proposed solution, allowing operations to be made on-line upon cloud based always encrypted data. As the data is stored encrypted at all times, even when processed as a result of client operations, cloud administrators cannot compromise data confidentiality. In this way, the proposed solution provides privacy enhanced guarantees for the management of sensitive data in cloud enabled provided data repository solutions.

In order to minimize development efforts for both new and existing applications, the processing of queries, encryption and decryption of data, and cloud integration are all transparently consolidated into the Middleware solution.

This chapter addresses the architecture and system model of the proposed solution. Initially we present an overview of the System Model (section 3.1), followed by the considered Adversary Model definition related to the security services provided by the proposed solution (section 3.2). Then, we describe the Software Architecture and its main components (section 3.3), and finally we present the definition of the extended SQL syntax and its processing within the Middleware Proxies (section 3.4), enabling searching and retrieving multimodal operations over the encrypted databases, maintained in the server side and related cloud storage backends.

3.1 System Model

The system model is represented in figure 31, and features two main groups of components and the communication support between them. The figure also displays a scope of different threats, later described in the Adversary Model (section 3.2), represented by dotted red lines, each extending along its range of actions.

The left side group of components in figure 31 represents components supporting the client side execution environment, and comprises a Middleware Client Proxy. In the considered System Model, client applications and the Client Proxy can either be executed in the same machine, or different machines in the same network. Client applications communicate with the Client Proxy through the Proxy Connector Interface. This interface provides an API to issue multimodal queries, that can involve text data and images. These queries will be supported transparently for the client applications. Aside from the extended support for image related functionalities (which are closely related to the support of image searching by equality or by similarity), the provided API resembles a standard SQL connector. It is quite simple to use the SQL connector to support queries from client applications.

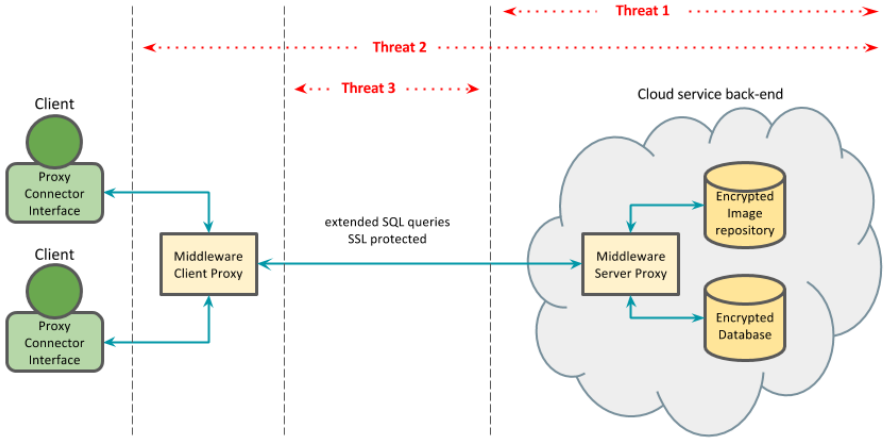


Figure 31: System model

The Middleware Client Proxy handles all the heavy lifting, namely the encryption of data, image feature extraction (for similarity searches), transforming the operations in a proper way to be executed in te server side, as querying operations over encrypted data.

The communication support between the Client Proxy and the Server Proxy components implements a secure communication channel supporting remote queries from the client side to be executed in the server side. All data flows involved in such queries are encrypted, namely data sent in the transformed queries (by the Client Proxy) and the obtained query results (returned from the Server Proxy). These results correspond to the results of queries executed by the Server Proxy on encrypted data, stored in the server

side.

The components in the represented System Model enable the server to execute queries on encrypted data almost as if it were executing the same queries on plaintext data, from the client application perspective. This means that with the provided support, the query plan for an encrypted query is similar to the original query over plaintext data, as locally executed by a client application, except that the operators comprising the submitted query to the Middleware Client Proxy API will be transformed to be performed on ciphertext data (in the Middleware Server Porxy), and can use modified operators and encrypted input data and returning encrypted results, according to each query type.

The Client Proxy stores and manages cryptographic keys used by the cryptographic support for the encryption and decrytion operations executed by the Middleware Proxies, according to the specific encryption transformations that must be executed to dispatch different query types.

These System Model design principles aim to benefit the clients, both in terms of software and processing power requirements. By placing the bulk of computational overhead on a reasonably powerful machine running the Client Proxy, even the most constrained client devices (e.g lightweight applications requiring a minimal set of memory, processing and energy requirements, running in smartphones or tablets) should be able to easily use our proposed solution.

The component group on the right side of figure 31 is located in a cloud computing environment. In this case, the Middleware Server Proxy can execute in a cloud computing node, with the encrypted database and image storage backends running in the same computing node or in a cloud-storage instance. We must notice that the purpose of the Middleware Server Proxy is to handle encrypted queries received from the Client Proxy, to be executed in the data repository backend. SQL based queries with encrypted input arguments are forwarded to the Encrypted Image repository. This allows the support for Multimodal queries that can use SQL based text and image searches. The extended queries are supported by a query language that we will present later on in this chapter.

The results from the referred extended queries are combined in a meaningful way, according to the interpreted query semantics. It is relevant to emphasize that all cloud based components cannot access plaintext data, for any protected data, because it is always encrypted in the server side. Thus, the Server Proxy employs partial homomorphic cryptography to be able to perform queries directly on encrypted data.

In the discussion of the System Model we notice that there is flexibility to address the development and deployment of the discussed components. In practice, those components may be deployed across different machines on a local network infrastructure, or in an internet deployment where the server side components can be deployed in a cloud provided infrastructure and/or cloud enabled services.

3.2 Adversary Model

The considered Adversary Model for the security requirements of the proposed solution is identified in figure 31 by a scope of different opponents, along with the range of action of each one of such opponents. Our Adversary Model is the same as defined in the CryptDB Adversary Model definition [59], but accounts for some relevant additional factors. We now describe each identified threat in our Adversary Model, and how our solution deals with such threats.

Threat 1 is related to an "honest but curious" system administrator. This is an all seeing passive adversary with full access to the server side or cloud storage backends and memory states, as well as incoming and outgoing network packets transporting queries related results. We assume that this adversary might try to disclose private information stored in the cloud, but will not tamper with stored data to compromise data integrity conditions, nor the availability of server side or cloud processing and data access operations. To guarantee data confidentiality against this threat, text information is encrypted with partial homomorphic constructs, while image data is protected with IES-CBIR (as initially discussed in the related work chapter). The Middleware Server Proxy is provided with keys to perform certain operations, such as addition of encrypted integers, but only when necessary and is never provided with keys that would allow access to plaintext data.

Threat 2 is related to an adversary present in the full extent of the system environment, both the local and server side or cloud domain. This adversary behaves as the "honest but curious" system administrator, but in the local environment he can access and obtain keys used to encrypt data from the Middleware Client Proxy memory state. Against this adversary we can only minimize damage, and offer privacy guarantees for data not encrypted with keys captured from memory. Data belonging to different principals are encrypted with different keys. Users who log into the system must authenticate themselves to an orthogonal authentication service, obtaining credentials for data the user has authorized access to. Therefore, only data belonging to logged users is vulnerable, for the time window from the moment the keys are placed into memory, until they are discarded.

Threat 3 is related to a man in the middle attacker, whose range of action is the communication channel between the middleware proxies. Although data items in our SQL based queries and the images themselves are encrypted (after the transformation of queries in the Client Proxy), the proxies still exchange some metadata in the clear. Furthermore, there is no reason not to prevent this adversary from learning query and access patterns. As such, this channel can be protected with a TLS enabled connection, where we can use, if required, mutual authentication. For security purposes we suggest that TLS must be configured in the endpoints of the Client Proxy and Server Proxy with TLS v1.3, according to the last recommendations on ciphersuites for TLS.

In the next subsection we will discuss in more detail the software architecture design principles and components that are behind the main components in the System Model

discussed before.

3.3 Software Architecture

This section presents a fine grained view of each of the main components introduced in the System Model. We address it in a bottom-up approach, starting with the Middleware Server Proxy, followed by the Middleware Client Proxy, and finally, the Proxy Connector Interface for client applications.

3.3.1 Middleware Server Proxy

The Middleware Server Proxy is located in the cloud environment, and is the entry point for accessing the backend structures. It does very little processing other than to forward requests to the backend components. All incoming data has been previously encrypted, so the Server Proxy's work is resumed to: opening connections to the backend components, and for stateful connections maintain a client identifier mapping to conections; dispatching the requests to the appropriate backend components; for multimodal queries, implement the query semantics (i.e how to combine the query results returned from the different backends). The Middleware Server Proxy API provides the REST calls for the various remote calls. Figure 32 shows the Middleware Server Proxy software architecture, which we describe in detail in the following paragraphs.

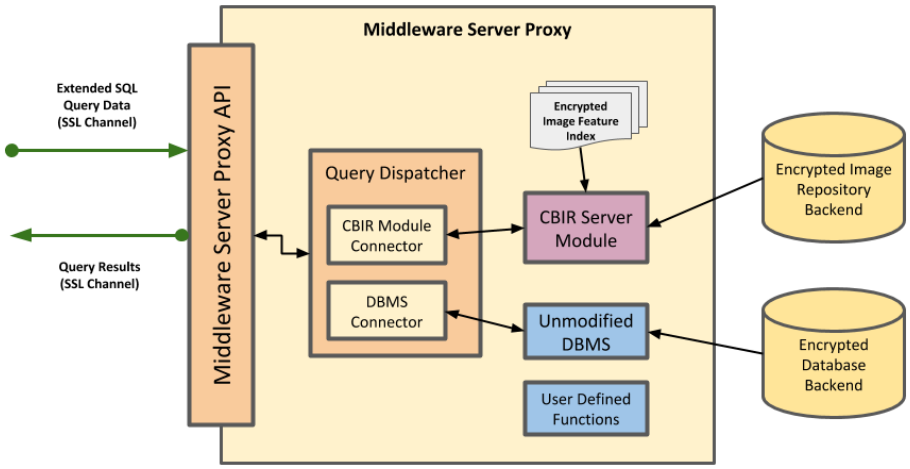


Figure 32: Middleware Server Proxy

Encrypted Database Backend

The Encrypted Database Backend is the persistent storage backend of the cloud environment where all relational database information is stored. Depending on the environment setup, it can be for example, the Amazon RDS service's provisioned storage facilities, or an EC2 instance's disk if the Middleware Server Proxy connects to a DBMS

manually installed on said instance. Regardless, all data is kept encrypted at all times in the storage backend.

Unmodified DBMS

The Unmodified DBMS is a standard relational Database Management System, as is, with no modifications to source code nor additional requirements from the cloud environment to function. Theoretically, it can be any cloud Database-as-a-Service, or manually set up DBMS in a cloud virtual machine instance. What should change, if necessary, is the adaptation of connectors for these services in the Middleware Server Proxy.

User Defined Functions

Some of the encryptions schemes' homomorphisms are implicitly supported by regular DBMSs. For instance, when operating on fields encrypted with OPE, the DBMS engine is oblivious to the fact that fields are encrypted and performs equality or inequality comparisons of values as if they were plaintext. Other operations, such as adding two encrypted numbers, are not implicitly supported and require specific functions to be defined. User Defined Functions (UDF) are defined in portable external libraries which can then be provided to the DBMS, not requiring any changes from the latter. Thus we define our homomorphic algorithms in UDFs, an idea borrowed from CryptDB.

Encrypted Image Repository Backend

The Encrypted Image Repository Backend is the storage facilities wherein the encrypted images are persisted. Possible options for this are a storage service provided by the cloud, like Amazon S3, or a virtual machine instance's disk. Images are previously encrypted with probabilistic encryption schemes, such as AES with a CBC chaining mode, and are stored with a unique identifier for retrieval.

Encrypted Image Feature Index

The Encrypted Image Feature Index is stored in the virtual machine instance's disk where the CBIR Server Module runs. After a number of images and accompanying features have been uploaded, the Encrypted Image Feature Index must be trained by the CBIR Server Module.

CBIR Server Module

The CBIR Server Module implements the cloud side processing of image related requests: receiving and persisting encrypted images and their encrypted extracted features; training and indexing the image features in the Encrypted Image Feature Index structure; performing image similarity searches on the Encrypted Image Feature Index given example image features, returning a list of identifiers of the top scoring images for posterior retrieval; retrieval of an encrypted image from the Encrypted Image Repository Backend, given an identifier returned from a previous search.

Query Dispatcher

The Query Dispatcher is the request decomposer and forwarding component, which wraps connectors to the backend components and implements the multimodal query semantics. Requests arriving at the Query Dispatcher are processed as follows: the request

is either an SQL, CBIR, or a multimodal SQL text and image query call. For SQL or CBIR calls, simply forward them to the Unmodified DBMS or CBIR Server Module through the appropriate connectors (described below). For multimodal queries, decompose the query into SQL and CBIR calls. Issue the calls to the appropriate backend components and combine the results. A more in-depth description of the results combining process is discussed in section 3.4.

CBIR Module Connector

The CBIR Module Connector implements the communications protocol with the CBIR Server Module. The existing client implementation was not suited to the needs of our Middleware. In the existing implementation, the client performs image encryption, decryption, feature extraction, and communication with the CBIR Server Module, all in one place. Our specification requires that everything must be previously protected by encryption before arriving to the Middleware Server Proxy. As such, the existing client was adapted and its functionalities distributed across our system. The CBIR Server Module is merely the adapted communications component from that client.

DBMS Connector

The DBMS Connector is a wrapper for an industry standard connector such as a JDBC API for Java. A standard connector API allows program interoperability with database backends, by managing connections to the DBMS and providing methods to issue SQL statements to query and update the database. Naturally, our DBMS Connector must adapt to the backend setup. Whether the DBMS is a cloud Database-as-a-Service, MySQL, PostgreSQL or any other standard solution, the DBMS Connector wraps around the appropriate API for it, and provides an easy to use interface for the Query Dispatcher.

Middleware Server Proxy API

As already mentioned, the Middleware Server Proxy API provides the remote call interface to issue commands and queries from the Middleware Client Proxy, which are forwarded to the Query Dispatcher. It is designed to resemble a standard DBMS connector's provided methods, and indeed some of the calls map directly to a standard DBMS call. For each method, a description of its processing in the Middleware Server Proxy components is provided below:

- *connect (database, user, password)*

  The Server Proxy establishes a connection to the Unmodified DBMS, with given *database*, *user* and *password* arguments, and generates a unique client connection identifier. The identifier is added to an indexing structure mapping identifiers to DBMS connections, and then returned to the Middleware Client Proxy. The identifier is necessary for subsequent calls, since connections to the DBMS maintain some state.

- *disconnect (id)*

Receives a client connection identifier. Terminates the associated connection to the Unmodified DBMS and removes the identifier entry from the client index structure.

- *setAutoCommit (id, autocommit)*

  This method is a wrapper for an actual setAutoCommit method of a DBMS connection. It sets the connection's auto-commit mode to the received boolean parameter *autocommit*.

- *setTransactionIsolation (id, level)*

  Wrapper for an actual setTransactionIsolation method of a DBMS connection. It receives an integer parameter and issues the call on the connection object to set the transaction isolation level to the received parameter *level*.

- *commit (id)*

  Wrapper for the commit call on a DBMS connection. Makes the changes of the current transaction since the last commit or rollback call on the connection permanent.

- *rollback (id)*

  Wrapper for a rollback call on a DBMS connection. Undoes all changes performed in the current transaction since the last commit or rollback call.

- *executeUpdate (id, statement)*

  Wrapper for an executeUpdate method through the DBMS connector. Receives an encrypted SQL *statement* string. Statements with calls to homomorphic functions such as sums on encrypted values also include the necessary key to perform those operations. Issues the executeUpdate call to the DBMS, which returns an integer value indicating the number of rows modified by the statement in the database. Return the integer value to the Client Proxy.

- *executeQuery (id, query)*

  Wrapper for an executeQuery method through the DBMS connector. Like the executeUpdate wrapper, receives an encrypted SQL *query*, with an included key (embedded in the query) if any homomorphic function calls are present. Calls to executeQuery return a Result Set, which in some cases may be very large. To avoid storing the entire Result Set in memory, the rows are streamed through the communication channel as they are pulled from the database.

- *indexImages (id)*

  Wraps a call to the index operation on the CBIR Server Module, which builds an index of the image features uploaded to the repository. Indexing very large datasets of images can take a long time, dependeing on the processing power of the machine.

- *addImage (id, imageBytes)*

  Receives an encrypted array of bytes, *imageBytes*, containing the image, the respective encrypted features, and a unique image identifier, previously generated. The identifier can be anything as long as it uniquely identifies an image, such as an integer (e.g the image belongs to a dataset of numbered images), or a hash code of the image bytes. The image, features and identifier are sent to the CBIR Server Module to be stored.

- *searchImage (id, imageFeatureBytes)*

  Receives the encrypted features of a query image, *imageFeatureBytes*, for a search operation. The features are forwarded to the CBIR Server Module, which after performing the search, returns a list of the top scoring matching image identifiers and respective scores. The list is returned to the Middleware Client Proxy.

- *getImage (id, imageId)*

  Receives a unique image identifer *imageId*, obtained from a previously issued search or multimodalQuery operation, and requests the image from the CBIR Server Module.

- *multimodalQuery (id, query, conjunction, imageFeatureBytes)*

  The multimodalQuery method combines an image search on the CBIR Server Module and an executeQuery on the DBMS to form a multimodal SQL text and image query. The parameters for this method are: an encrypted SQL statement *query* string, where one of the fields on the SELECT clause is an image field (its values are image identifiers); a *conjunction* string specifying how results are to be combined (intersection, conjunction, or none if the SQL query has no WHERE clauses); the encrypted image features *imageFeatureBytes* for the query. Streams the rows of the Result Set to the Middleware Client Proxy much like in the *executeQuery* method.

3.3.2 Middleware Client Proxy

The Middleware Client Proxy is in a Local Environment, in the same network as the client applications or otherwise within relative proximity. Its computational requirements justify its deployment to a reasonably powerful machine, as it must process queries and perform heavy cryptographic operations to relieve client devices from that burden. The Middleware Client Proxy API provides the remote call interface to clients for issuing queries and commands. Figure 33 displays a detailed view of the Middleware Client Proxy components, that we now describe:

Proxy Engine

The Proxy Engine is the high-level component of the Middleware Client Proxy. It receives the calls issued through the Middleware Client Proxy API and implements the logistics of all processing in the Client Proxy, namely: accepting and managing client

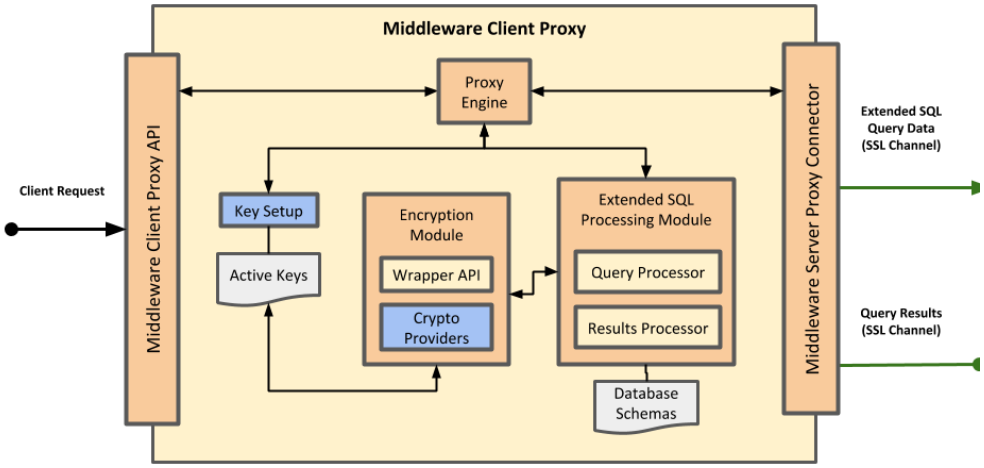


Figure 33: Middleware Client Proxy

connections to the Middleware; setup and management of client encryption keys necessary for cryptographic operations; the processing of SQL, Content Based Image Retrieval, and multimodal SQL and image operations; after processing, issuing the calls to the Middleware Server Proxy; and processing returned data and delivering it to the client applications.

Key Setup

The Key Setup process is ideally orthogonal to our solution. A user would authenticate itself to an external service to obtain credentials for system use. The credentials would be supplied in a *connect* call to the client proxy, and kept in memory for the duration of a client's session. In our implementation keys are loaded from disk memory, or generated and persisted if they do not exist.

Active Keys

Active Keys are the in-memory encryption keys of active client connections.

Encryption Module

The Encryption Module is a wrapper for the Crypto Providers offering the necessary cryptographic constructions for query processing. It features a simplified Wrapper API, callable from the Extended SQL Processing Module, and the Crypto Providers themselves.

Wrapper API

Provides simple method calls that wrap the Crypto Providers operations.

Crypto Providers

Cryptographic Providers for: Partial Homomorphic Encryption constructs, needed to encrypt and decrypt SQL textual data and support the respective homomorphisms in the server side without access to the plaintexts; Searchable Symmetric Encryption constructs for Content Based Image Retrieval encryption, decryption and feature extraction, to allow image similarity search and retrieval operations on the server side.

Extended SQL Processing Module

The Extended SQL Processing Module handles the processing of SQL statement, CBIR functionalities and multimodal SQL and image query calls, as well as processing of returned data before it is delivered to the client. A Database Schemas structure is maintained with local information regarding the remote databases. All cryptographic operations are accessible through the Encryption Module and its Wrapper API.

Query Processor

SQL statements, CBIR calls, and multimodal calls issued by the client are transformed in the Query Processor. SQL statements are anonymized by encrypting all values and database, table and field names with appropriate homomorphic constructs. Images are encrypted with probabilistic schemes for safe storage in the server backends, while their features are extracted and encrypted with Searchable Symmetric Encryption constructs for CBIR. Multimodal queries are decomposed into SQL and CBIR elements, which are processed according to the previous approaches, and combined once more into a single query. Additionally, the Query Processor modifies the Database Schemas structure according to changes to the remote backends: creating new databases, new tables, or deleting them.

Results Processor

The Results Processor decrypts results returned from the Middleware Server Proxy, both text and image data.

Database Schemas

The Database Schemas structure maintains entries with local information regarding all databases created in the remote backend through the Middleware. For each database, the tables structures and field datatypes are stored. This information is necessary to reason about queries in the Query Processor. All identifiers (database, table, and field names) are anonymized, as the Database Schemas structure is persisted on the Middleware Client Proxy environment and is, as per our Adversary Model, accessible to an adversary.

Middleware Client Proxy API

The Middleware Client Proxy API provides the interface for client applications to communicate with the Middleware Client Proxy. Its methods are the same as those of the Server Proxy API, since they are all directly mapped to their counterparts (i.e a call to method *x()* on the Client Proxy is processed, and forwarded by a call to method *x()* on the Server Proxy). Their behaviours, of course, are not the same as in the Server Proxy. For each method, the processing performed at the Middleware Client Proxy is as follows:

- *connect (database, user, password)*

  Upon a *connect* call, the Key Setup process is performed. Ideally, clients would obtain access credentials from an orthogonal third party authentication service, and provide them in this call. In our solution, keys are loaded from disk memory into the Active Keys. After the Key Setup, the *database* name argument is anonymized,

resulting in an *anonymizedDatabase* name. A *connect(anonymizedDatabase, user, password)* call to the Server Proxy is issued, to open a new DBMS connection and return a client connection identifier. Note that *password* is the DBMS user password, not an encryption key for any cryptographic construct. The returned client connection identifier is transmitted to the client.

- *disconnect (id)*

  Issues a disconnect (id) call to the Server Proxy, to terminate the DBMS connection associated with the *id*. The Active Keys belonging to the disconnecting user are discarded from memory.

- *setAutoCommit (id, autocommit)*, *setTransactionIsolation(id, level)*, *commit(id)*, and *rollback(id)*

  A call to any of these methods results in a call to its counterpart on the Server Proxy.

- *executeUpdate (id, statement)*

  The received plaintext SQL *statement* is encrypted in the Extended SQL Processing Module, resulting in an *encryptedStatement*. An *executeUpdate(id, encryptedStatement)* call is issued to the Server Proxy. The return value is an integer number indicating the number of rows modified by the issued *encryptedStatement* in the remote database. This number is returned to the client.

- *executeQuery (id, query)*

  The received plaintext SQL *query* is encrypted in the Extended SQL Processing Module, yielding an *encryptedQuery*. An *executeQuery(id, encryptedQuery)* call to the Server Proxy is issued, which returns a stream of encrypted Result Set rows. The rows are streamed to the client as they are received from the Server Proxy and decrypted.

- *indexImages (id)*

  Issues an *indexImages(id)* call to the Server Proxy to trigger the training and indexing process on the CBIR Server backend.

- *addImage (id, imageBytes)*

  Receives an array of bytes *imageBytes*, containing a plaintext image. The Query Processor extracts the image features, encrypts the extracted features and the plaintext image, and places both in the *encryptedImageBytes* array. A call to *addImage(id, encryptedImageBytes)* is issued to the Server Proxy.

- *searchImage (id, imageBytes)*

  Receives a plaintext byte array *imageBytes* of an image to be submitted for similarity search. The Query Processor extracts the image features and encrypts them,

yielding *encryptedFeatureBytes*. Only the features are necessary for this operation. The call *searchImage(id, encryptedFeatureBytes)* is issued to the Server Proxy, returning a list of identifiers, ordered by score, for the top scoring matches in the search performed on the CBIR Server backend. This list is returned to the client.

- *getImage (id, imageId)*

  Given an *imageId* obtained from a previous *search* or *multimodalQuery* operation, issues a *getImage(id, imageId)* call to the Server Proxy to retrieve the corresponding image. The retrieved image is decrypted at the Results Processor, and then returned to the client.

- *multimodalQuery (id, query, conjunction, imageBytes)*

  Given: a plaintext extended SQL *query*, where one of the fields on the SELECT clause is an image field; a *conjunction* string specifying how SQL and CBIR results should be combined (intersection, conjunction, or none if the SQL query has no WHERE clauses); and a plaintext byte array *imageBytes*. The Query Processor encrypts the *query*, extracts the image features from the *imageBytes* and encrypts them, yielding *encryptedQuery* and *encryptedFeatureBytes*. The call *multimodalQuery(id, encryptedQuery, conjunction, encryptedFeatureBytes)* is issued to the Middleware Server Proxy. As in the *executeQuery* method, the call returns a stream of encrypted rows, which are decrypted by the Results Processor and streamed to the client. Note that no images are yet retrieved in this operation, only their identifiers in the image field the *query* specifies. Images are retrieved by calls to *getImage*.

Middleware Server Proxy Connector

The Middleware Server Proxy Connector implements the communications protocol of the Middleware Server Proxy, and provides the Proxy Engine the interface to issue calls to the Server Proxy API.

3.3.3 Proxy Connector Interface

The Proxy Connector Interface, shown in figure 34, is the smallest macro component of our software architecture. It is a small library that implements the communications protocol of the Middleware Client Proxy API, and is used by client applications to access the services supported by our Middleware. It is completely oblivious to the processing that is transparently integrated into the Proxies.

The components of the Proxy Connector Interface are:

Middleware Client Proxy Connector

Implements the communications protocol of the Middleware Client Proxy API. Calls to the Client API are directly mapped to this component.

Result Handler

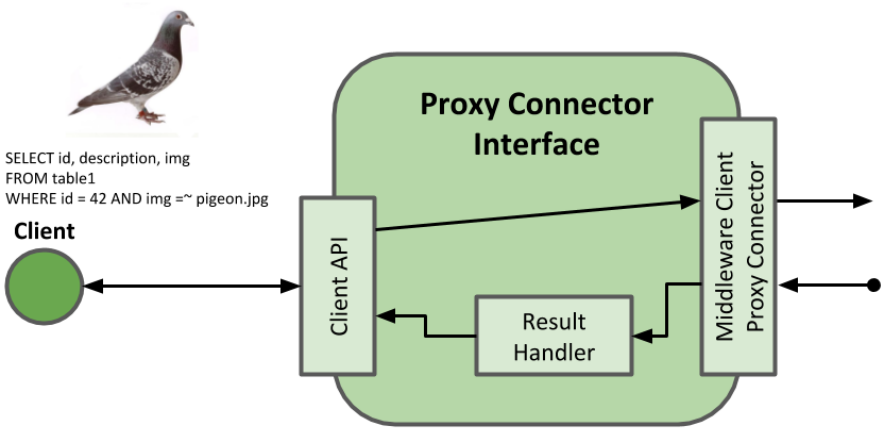


Figure 34: Proxy Connector Interface

The Result Handler implements the handling of results returned from the Middleware Client Proxy. Most returned results are trivial. A *connect* call returns an integer client connection identifier, which is then kept by the Proxy Connector Interface for subsequent calls. Other methods may return integers, lists of data, byte arrays, or nothing, and are passed onto the application. Calls to *executeQuery* and *multimodalQuery* however, return streams of Result Set rows. For this particular case, the Result Handler returns an object that mimics the functionality of a Result Set object in standard DBMS connectors. This object, which we shall call a **StreamedResultSet** consumes rows from the stream into an internal representation of the current row, the values of which are then accessible through a set of methods. The complete method specification for StreamedResultSet is the following:

- *isLast ()*

  Returns the boolean value True if the *current* row is the last in the stream, False otherwise.

- *next ()*

  Pulls a row from the stream and sets *current* to the pulled row.

- *getInt (columnIndex)*

  Retrieves the value of the field at position *columnIndex* of the *current* row as an Integer.

- *getString (columnIndex)*

  Retrieves the value of the field at position *columnIndex* of the *current* row as a String.

We note that there is no *getImage()* method. Rows do not return images, only their identifiers (represented as either an integer or a string).

Client API

The Client API provides methods to client applications for issuing the various calls to the Middleware Client Proxy. When the application issues a call on the Client API, that call is forwarded to the Client Proxy through the Middleware Client Proxy Connector. The methods are as described below:

- *connect (address, database, user, password)*

  Takes the Middleware Client Proxy's *address*, a *database* name, the *user* and respective DBMS *password* as arguments. Starts a new DBMS connection for this client in the server backends. Receives a unique client connection identifier from the Client Proxy for subsequent calls.

- *disconnect (id)*

  Signals the Middleware Client Proxy to terminate the client connection, and discards the client id.

- *setAutoCommit (id, autocommit)*

  Set the connection's auto commit mode to the received *autocommit* boolean value.

- *setTransactionIsolation (id, level)*

  Set the transaction isolation level for the connection to the received parameter *level*.

- *commit ()*

  Issues a call to make the changes in the current transaction permanent.

- *rollback ()*

  Issues a call to undo the changes performed in the current transaction.

- *executeUpdate (id, statement)*

  Given an SQL *statement*, issues a call to the Middleware Client Proxy, where query processing is transparently performed. Returns the number of rows modified by the executed *statement*.

- *executeQuery (id, query)*

  Similar to the *executeUpdate* method, but the received query string corresponds to an SQL statement that returns a stream of Result Set rows. Returns a StreamedResultSet object to pull the rows from the stream. This method supports standard SQL syntax only. Multimodal SQL text and image queries are issued through the *executeMultimodalQuery* method.

- *indexImages (id)*

  Issues a call to initiate the indexing of the encrypted image features on the Encrypted Image Repository backend.

- *addImage (id, imagePath)*

  Receives an *imagePath* to an image file on disk, which is read as a byte array. A call is issued to the Middleware Client Proxy to add the image to the backend Encrypted Image Repository.

- *searchImage (id, imagePath)*

  Receives an *imagePath* to an image file on disk. Issues a call to *searchImage* on the Client Proxy to obtain the list of identifiers of the top scoring image matches.

- *getImage (id, imageId)*

  Takes a unique image identifier, obtained from a previous image *search* or *executeMultimodalQuery*, and issues a call to retrieve the corresponding image.

- *executeMultimodalQuery (id, query)*

  Similar to an executeQuery, but supports the extended SQL syntax for multimodal queries. The extended SQL *query* must select an image field, and a Where clause for image similarity search specifies a path to an image stored on disk. The query is decomposed, removing the image condition from the Where clauses. The condition's conjunction (if any) is extracted, as well as the image path, the latter used to load the image from disk into *imageBytes*. By removing the condition, the resulting *query* becomes *decomposedQuery*, and a call to the Middleware Client Proxy with the form *multimodalQuery(id, decomposedQuery, conjunction, imageBytes)*. Like the *executeQuery* method, returns a StreamedResultSet to pull the Result Set rows from the stream of rows. Images are associated to rows in a field containing image identifiers. Clients obtain images by issuing the *getImage* method.

3.4 Encrypted Multimodal Query Processing

This section presents the design of a hybrid support for SQL querying and Content Based Image Search and Retrieval capabilities. For that purpose, our solution addresses the definition of an SQL syntax which includes composable queries that mix standard SQL statements with additional syntactic elements. This will allow client applications to issue multimodal queries which use a set of homomorphic operations to query encrypted SQL database backends, and image searching and retrieval functionalities on an Image Repository backend, to deliver combined results in a transparent manner.

The choice of the supported operations also follows a design criteria based on the need to have to ability to perform the operations typically found in industry standard database

benchmarks, namely TPCC. This is a relevant criteria because, as we will describe later, we resort to an implementation of the TPCC benchmark for experimental evaluation of our solution.

In the following subsections we explain the relevant aspects of the above support.

3.4.1 SQL Query Syntax

Table 31 summarizes the supported constructions for SQL statements as we support in our proposal. We note that the solution is not concerned with delivering a full-fledged SQL language support implementation, given the specific objectives of the dissertation. Rather, a minimal subset of the SQL language is selected in accordance to the available partial homomorphic constructions [], while allowing reasonable interoperability with a DBMS. This base SQL support is then extended to combine the querying of SQL backends with encrypted image searching and retrieval capabilities through the specific implementations of [24].

Statement Type	Syntax
Use	USE *database_name*
CreateDB	CREATE DATABASE [IF NOT EXISTS] *database_name*
DropDB	DROP DATABASE [IF EXISTS] *database_name*
CreateTable	CREATE TABLE [IF NOT EXISTS] *table_name* (*field_decl* [, *field_decl* ...])
DropTable	DROP TABLE [IF EXISTS] *table_name*
Insert	INSERT INTO *table_name* (*field_name* [, *field_name* ...]) VALUES (*value* [, *value* ...])
Delete	DELETE FROM *table_name* [WHERE *condition* [{AND | OR} *condition* ...]]
Update	UPDATE *table_name* SET *upd_assign* [, *upd_assign* ...]
Select	SELECT [DISTINCT] [FOR UPDATE] *field_selection* [, *field_selection* ...] FROM *table_selection* [WHERE *condition* [{AND | OR} *condition* ....]] [ORDER BY *field_name*]
MultimodalSelect	SELECT [DISTINCT] [FOR UPDATE] *field_selection* [, *field_selection* ...] FROM *table_selection* WHERE *extended_condition* [{AND | OR} *extended_condition* ...]

Table 31: Supported SQL Statement Types

Table 31 contains a number of expressions (in italic), displayed in a separate table 32. The latter ommits the expressions *database_name*, *table_name*, *field_name* and *value*, which are self explanatory.

In our design, the statements types presented in table 31 allow for the operations we now describe. Most statement types are trivial to anyone with minimal knowledge of relational database systems, but for the sake of completeness we provide a short description

Expression Type	Syntax
field_decl	*field_name datatype* [NOT NULL]
condition	*field_name* {= | <| >| <= | >=} *value*
upd_assign	*field_name* = {*value* | *field_name* + *value*}
table_selection	{*table_name* | *join_tables*}
join_tables	*table_name* JOIN *table_name* ON *table_name.field_selection* = *table_name.field_selection*
field_selection	{*field_name* | *table_name.field_name* | COUNT(*field_name*) | MAX(*field_name*) | MIN(*field_name*) | SUM(*field_name*) | COALESCE(*field_selection*,*value*) | IMG(*field_name*)}
extended_condition	{*condition* | *field_name* =~ *'file.jpg'*}
datatype	{INT | VARCHAR(*size*) | CHAR(*size*) | TEXT | IMG}

Table 32: Expressions from SQL statements syntax

for each, along with statement examples, and for those that include syntax extensions beyond standard SQL we provide a more complete description.

- **Use**: Tells the DBMS to use the specified *database_name* as the default database for the subsequent statements.

```
USE example_db;
```

- **CreateDB**: Create a new database.

```
CREATE DATABASE example_db;
```

- **DropDB**: Delete a database.

```
DROP example\_db;
```

- **CreateTable**: Create a new table with the specified fields. The available types for the declaration of fields in the new table are some of the common SQL types, and additionally, the IMG type, as table 32 states.

```
CREATE TABLE people (person_id INT NOT NULL,
  firstname VARCHAR(20), lastname VARCHAR(20));
CREATE TABLE holiday_photos (person_id INT NOT NULL,
  country VARCHAR(20), photo IMG);
```

- **DropTable**: Delete a table.

```
DROP TABLE holiday;
```

- **Insert**: Insert a new row of values into a table. For IMG fields, a value is a path to an image on local storage.

```
INSERT INTO people (person_id, firstname, lastname)
    VALUES (1, 'John', 'Johnson');
INSERT INTO holiday_photos (person_id, country, photo)
  VALUES (1, 'Portugal', 'Caparica.png');
```

- **Update**: Update one or more rows in a table. Updates can set row fields to literal values, or increment them (INT type fields only).

```
UPDATE people SET firstname = 'Rick', lastname = 'Richards'
  WHERE person_id = 1;
UPDATE bankaccounts SET balance = balance + 200;
```

- **Select**: Select rows from a database table that meet the query search criteria. For standard SQL searches only. For multimodal SQL and image queries, see *MultimodalSelect*.

```
SELECT firstname FROM people;
SELECT person_id FROM bank_accounts WHERE balance < 500;
```

- **MultimodalSelect**: Select rows from a database table that satisfy both the SQL query and image search criteria, the latter specified in a WHERE *extended_condition* expression clause.

```
SELECT person_id, photo FROM holiday_photos
  WHERE country = 'Egypt' OR photo =~ 'pyramids.jpg';

/* A multimodal query does not necessarily have to return images...
Example that performs image search without returning images: */
SELECT country FROM holiday_photos
  WHERE photo =~ 'eiffel_tower.jpg'
```

3.4.2 SQL Mapping in the Proxy APIs

Table 33 shows, for each of the Client Proxy API calls, which statement types are supported.

Most statement types are issued through the *executeUpdate* call, while only the Select and the MultimodalSelect are issued through *executeQuery* and *multimodalQuery*, respectively. The rationale for this mapping is the following: Statements issued through *executeUpdate* are statements that somehow modify the database. These statements might create or delete databases or tables, and insert, delete or update rows in a table. The associated return type for this call is an integer value, the number of rows affected by the executed statement. Both the *executeQuery* and multimodalQuery return a stream of

Middleware Client Proxy API call	Supported statements types
executeUpdate()	Use CreateDB DropDB CreateTable DropTable Insert Delete Update
executeQuery()	Select
multimodalQuery()	MultimodalSelect

Table 33: Mapping of Middleware Client Proxy API calls to statement types

Result Set rows, but are distinguished by how they are internally processed by the Middleware Proxies. The subsetions below address the internal processing of the different statement types.

3.4.3 SQL Encryption Support and Internal Runtime Processing

The encryption of SQL statements differs from one type of statement to another. The general structure of a statement, as seen in table 31 is kept. The names of databases, tables and fields, and literal values are the elements that must be encrypted, not only according to what the element is, but also its context, and additionally its datatype. In table 34, for each statement type and respective elements, the datatype and possible encryption schemes are shown.

The Possible Encryption Schemes acronyms refer to partial homomorphic and Content Based Image Retrieval cryptographic constructs, which we use throughout the remainder of the document. They are:

- **DET**: A deterministic encryption scheme for text data. Plaintexts of the same value are encrypted to the same ciphertext, allowing for equality matching on the cipertexts.
- **OPE**: An Order Preserving Encryption scheme for 32 bit Integers. Has the same equality preserving properties of DET, and additionally preserves the order in the ciphertexts.
- **ADD**: ADD is an implementation of the Paillier cryptosystem. Does not preserve equality or order properties. ADD ciphertexts can be homomorphically added, without accessing the plaintexts.
- **CBIR**: CBIR refers to the Content Based Image Retrieval Crypto Provider's constructions, and specifically for the MultimodalSelect statement type, its image feature extraction and encryption functionalities.

Statement Type	Encrypted elements	Datatypes	Possible Encryption Algorithms
Use	Database name	Text	DET
CreateDB	Database name	Text	DET
DropDB	Database name	Text	DET
CreateTable	Table name Field names	Text	DET
DropTable	Table name	Text	DET
Insert	Table name Field names	Text	DET
	Inserted values	Text	DET None
		Integer	OPE ADD None
Delete	Table name WHERE clause field names	Text	DET
	WHERE clause values	Text	DET None
		Integer	OPE None
Update	Table name Assignment field names WHERE clause field names	Text	DET
	Assignment values	Text	DET None
		Integer	OPE ADD None
	WHERE clause values	Text	Text None
		Integer	OPE None
Select	Tables names SELECT field names WHERE clause field names	Text	DET
	WHERE clause values	Text	DET None
		Integer	OPE None
MultimodalSelect	Table name SELECT field names	Text	DET
	WHERE clause field names WHERE clause values	Text	DET None
		Integer	OPE None
	Image features	Bytes	CBIR

Table 34: Middleware Client Proxy internal processing of Statement Types

A general rule applies to all statement types. All database, table, and field names, which are simply text identifiers, are encrypted with DET. This guarantees the encryption of a name will always produce the same ciphertext, and the DBMS is oblivious to this. As far as it is concerned, it is simply operating on text and a ciphertext is as valid as any other name. The contextual processing of each statement type is now described, in terms of how the Middleware Client operates, and how the final execution in the DBMS takes place. For every statement type except the MultimodalQuery, the Server Proxy's only function is to forward the call to the DBMS and stream results back to the Client Proxy, and is thus ommited.

- **Use**

  The database name is encrypted with DET. The Database Schemas structure is consulted for the existence of a remote database with the encrypted name, and if it exists, the call to *executeUpdate* is issued to the Middleware Server Proxy. The DBMS sets the connection's default database.

- **CreateDB**

  The database name is encrypted with DET. The *executeUpdate* call is issued to the Middleware Server Proxy. In the DBMS, a new database with the encrypted name is created. If the call is sucessful, a new database entry with the encrypted name is added to the Database Schemas structure.

- **DropDB**

  The database name is encrypted with DET, and the Database Schemas consulted for the existance of a remote database with the encrypted name. If it exists, the *executeUpdate* is issued to the Server Proxy, and the DBMS deletes the database. The database entry is removed from the Databases Schemas.

- **CreateTable**

  The table and field names are encrypted with DET. Each field is mapped to one or more fields for each of the cryptographic schemes supporting the declared datatype of the field: Text types (CHAR, VARCHAR and TEXT) map to DET; Integer maps to OPE and ADD; the Image datatype is not mapped to any scheme, as image identifiers by themselves do not reveal anything about the image contents. The mapped fields are prepended a prefix for its scheme, and declared with SQL datatypes that can hold their scheme's ciphertexts. Table 35 displays this transformation. The *executeUpdate* call is issued, and if the table is successfully created in the DBMS, a table entry is added to the Database Schemas structure, with encrypted names and the original datatypes of fields.

- **DropTable**

The table name is encrypted with DET, the Database Schemas structure consulted for the existence that table, and the *executeUpdate* call to the Server Proxy is made. If the DBMS successfully deletes the table, it is also removed from the Database Schemas.

- **Insert**

  The table name and field names are encrypted with DET. The existence of the table is checked in the Database Schemas. If it exists, the plaintext datatype of each field is retrieved from the Database Schemas, in order to map to every field supporting the schemes for that datatype as per table 35. This mapping propagates to the values, which are encrypted with the corresponding cryptographic schemes. The *executeUpdate* call is performed, and the DBMS inserts the row in the table.

- **Delete**

  Encrypts the table name with DET, and checks if it exists in the Database Schemas. If it exists and the statement has no WHERE clauses, issues *executeUpdate* to the Server Proxy. If WHERE clauses exist, each condition is accordingly processed:

  – The condition is an equality: check the Database Schemas for the field's datatype. If it's a text type, encrypt the value with DET. If it's an integer, encrypt the value with OPE. If it's an Image type, do not encrypt. Add the appropriate scheme prefix to the encrypted field name.
  – The condition is an inequality: encrypt the value with OPE if the field is of integer datatype, or do not encrypt if it's an Image datatype. Prepend the appropriate prefix to the encrypted field name.

  Finally, issue the *executeUpdate* call to the Middleware Server Proxy. The DBMS deletes rows from the table according to the statement conditions, or all rows if no WHERE clauses were specified. For all other statement types, the processing of the WHERE conditions is the same as here described.

- **Update**

  Encrypts the table name and all field names with DET. Verifies the table's existence, and if confirmed, proceeds as follows. There are two possible cases for an assignment in an Update statement, as previously seen in table 32 for *upd_assignment*, and the processing differs. The first case is an assignment of the form *field_name = value*, where the field is to be set to a new value. The approach is similar to an Insert, which is to provide value encryptions for every scheme for the field's datatype, retrieved from the Schemas. The second case is an assignment of the form *field_name = field_name + 1*, where the datatype is implicitly integer. The corresponding ADD field in the remote database can be homomorphically updated by placing a call to a User Defined Function to sum the ADD field with the provided value, but the OPE field cannot. Since the OPE values can't be homomorphically

updated, we must perform an intermediate *executeQuery* to retrieve them from the remote database, decrypt and perform the sum on the plaintexts, and encrypt them again. WHERE conditions are processed as already explained in the Delete description. The *executeUpdate* call is issued and the DBMS updates the rows matching the conditions.

- **Select**

  Encrypt all names with DET, and verify the table's existence in the Schemas. A Select statement may perform a JOIN on two tables fields, but only if the two fields have been encrypted with the same key. Joining two tables on fields encrypted with different keys requires a cryptographic construct not present in the cryptographic providers we leverage. The WHERE clauses are encrypted as already described. Field selections have a specific processing:

  Field selections need retrieve only one of the mapped fields from the remote database, and logically, the field corresponding to the most lighweight scheme shoud be retrieved. If the selection is of the form *field_name* then we retrieve DET for text fields, and OPE for integer fields. Other types of selections may constrain what scheme the result is encrypted with: COUNT(*field_name*) and SUM(*field_name*) are replaced with calls to their UDF corresponding calls, to perform a count and an aggregate sum, respectively, and return an ADD encrypted value, obtained from the homomorphic sum; MAX(*field_name*) and MIN(*field_name*) return OPE values; a COALESCE(*field_selection*, *value*) is recursive to its *field_selection*. Information regarding the field selection is maintained in order to know how to decrypt the expected stream of Result Set rows.

  After the processing of the query, the *executeQuery* call is issued to the Server Proxy, and the DBMS executes the query. The ensuing Result Set is streamed by the Server Proxy back to the Client Proxy, row by row. Before streaming the rows to the client application, the Client Proxy must decrypt its values according to the information gathered in the previous processing step.

- **MultimodalSelect**

  The MultimodalSelect is an extension of the Select statement type, where field selections may include the IMG(*field_name*) form. Values of this field selection are not decrypted, because as previously stated, Image field values are not encrypted. The syntax presented in table 32 for an *extended_condition* are valid at the client pplication level, but the query is decomposed before it reaches the Middleware Client Proxy. The WHERE condition for the image is not present, instead a conjunction 'AND', 'OR', or none is provided as an argument in the call to the Client Proxy, along with the plaintext image. The image features are extracted and encrypted, and the *multimodalQuery* call is issued to the Middleware Server Proxy.

A *multimodalQuery* call is processed by the Server Proxy depending on the received conjunction argument. Ideally, both the SQL query and the image search components of the multimodal query would be executed in parallel. For the 'AND' conjunction and the lack of conjunction cases, this is straightforward: issue the query and the search to their respective backend components, and when both results are returned, filter the rows before they are streamed back to the Client Proxy by checking if the image field value belongs to the set of image search results. The 'OR' conjunction complicates the parallel execution. If the SQL query is issued before we obtain the image search results, rows that would belong to the final combined Result Set are left out. Hence, two options are available. Serialize the process, by issuing the image search first, and appending a WHERE clause with the set of image identifiers to the query before having the DBMS execute it. Or remove all WHERE clauses from the SQL query, which would yield every row of the selected tables, and issue the query and the search in parallel. When both results are returned, apply the filtering with the removed conditions plus the image search results as the rows are streamed.

Original field name	Plaintext datatype	Schemes	Mapped field names
plaintext_fieldname	Text	DET	DET_encrypted_fieldname
	Int	OPE	OPE_encrypted_fieldname
		ADD	ADD_encrypted_fieldname
	Image	None	IMG_encrypted_fieldname

Table 35: Transformation of a field declaration

Chapter 4

Implementation

Chapter 4 addresses the implementations aspects of the developed solution. We have implemented a prototype of the system model design as described in chapter 3. The validation and experimental evaluation of the proposed solution was prioritized over more technical issues, and as such the instantiation of the system model and architecture in the implemented prototype has some simplifications, which will be addressed in the following subsections.

4.1 Implemented Prototype

The prototype implementation is available in a bitbucket repository [25]. The referenced prototype comes with all the installation instructions and is ready to deploy. It was used for the experimental evaluation reported in Chapter 5.

As mentioned, this prototype was developed with some simplifications, considering the conceptual system model and the reference software architecture components. These simplifications are the following:

The onion model to provide different onion constructions combining different homomorphic encryption layers has not been implemented in the prototype. It would serve as a proof of concept for a complete implementation. However, for experimental assessment purposes, we can avoid the complexity of such implementation, given the concrete evaluation that we are interested in. Concretely, for the purposes of our experimental evaluation, the complete implementation would have little impact for specific onion encryptions, such as, onions encapsulating an homomorphic encryption layer protected by an external random encryption layer (for example, using an efficient symmetric cryptographic algorithm such as AES). For other complex onion constructs with multiple layers of homomorphic encryption this could be a performance bottleneck. In the context of multiple sequential or even concurrent transactions, it would not be feasible to constantly peel and wrap the onion layers. Furthermore, we follow the same principles of evaluation of oter related work. For example, in CryptDB literature [59] the authors state that "there are no onion adjustments during the TPC-C experiments". The onions were adjusted to

the minimum required layers to support the queries prior to the tests.

The management of keys is not in accordance to the Adversary Model as defined in Chapter 3. Keys are kept in local storage at the Middleware Client Proxy for the purpose of experimental evaluation only. This option is justified because we are particularly interested in evaluating the performance and latency of SQL operations from clients executed over the server, when the keys are already obtained in the client side after a user authentication process. In a production deployment, the management of keys would be done via a third party or orthogonal service, and keys would only be kept in memory for the duration of a client's session. However, theses types of deployments are also not addressed by other relevant research proposals [6, 59].

Despite the simplifications, all services and components are functional and offer the required support for SQL operations, image storage, searching and retrieval, and multi-modal SQL and image querying of encrypted remote backends necessary to conduct the experimental evaluations.

4.2 Implementation Environment

Our Middleware was developed in Scala and Java, and is provided as an SBT project - as found in the implementation page [25]. The current prototype is built with Java 8 (version 1.8.0_151) [1] and Scala version 2.12.3 [2]. The solution integrates the HomoLib Java Library (included in the repository), which provides the Homomorphic constructs for SQL encrypted queries. Other cryptographic libraries used are: the MIE Crypto Provider (also included in the repository) which offers the IES-CBIR [24] capabilities; and the BouncyCastle Crypto Provider version 1.59 [3], required by the MIE Crypto Provider. In the prototype development environment we used the Akka framework [4], modules akka-http 10.0.10 and akka-http-spray-json 10.0.10 for HTTP communications and JSON support. Finally, we used MySQL Server (version 5.7) [5] as the server side SQL Database and mysql-connector-java version 5.1.24 [6] for JDBC connectivity. All of the above components are currently integrated in the prototype implementation as available in [25].

Some Middleware components of our solution require additional external libraries that must be installed separately. The Middleware Client Proxy requires OpenCV [7] (version 2.4.10) compiled with Java support. The Middleware Server Proxy communicates with an additional component that implements the image storage, the MIE Server, supported by the server side file system. The MIE Server is written in C++11 and was

[1] http://openjdk.java.net/projects/jdk8/
[2] https://scala-lang.org/
[3] https://bouncycastle.org/
[4] https://akka.io/
[5] https://www.mysql.com/
[6] http://repo.maven.apache.org/maven2/mysql/mysql-connector-java/5.1.24/
[7] https://opencv.org/releases.html

compiled with g++ 5.4.0 [8], and also requires OpenCV, but the version must be 3.0.0. An UDF (User Defined Function) library was written in C and compiled with gcc 5.4.0, implementing functions for addition and aggregate sums of encrypted integers with the Paillier encryption algorithm.

4.3 Implementation Details

The implementation details of the various system components are explained in this section. Some of the leveraged components had little to no changes required to integrate them with our Middleware, such as the Crypto Providers for partial homomorphic constructs and Content Based Image Retrieval primitives, and the CBIR Server Module, where minor changes were made to measure the performance of some operations. An existing client that used the CBIR constructions for image encryption and feature extraction, and implemented the communications protocol with the CBIR Server Module, was heavily adapted. For every macro component presented in the System Model and Software Architecture chapter, we address their specific implementation details in the next subsections, ommiting the components here mentioned that received negligible changes.

4.3.1 Middleware Server Proxy Implementation

The Middleware Server Proxy consists of six Scala classes, implementing the Middleware Server Proxy API, a client connection manager, and the Query Dispatcher, which wraps the two connectors for the CBIR Server Module and MySQL Server, and lastly, a Result Set iterator used to stream the Result Set's rows.

4.3.1.1 Middleware Server Proxy API

The Server Proxy API is a web API provided by an Akka HTTP Server, encapsulating a client connection manager Akka actor. The various calls of the API are exposed through the HTTP server's endpoints, acessible through their URIs. Incoming HTTP requests have an id parameter identifying the client, the exception being a *connect* request. Additional parameters are sent in Scala case classes marshalled to JSON objects. When a *connect* request is received, the case class with the connection request parameters is extracted from the JSON object. It is forwarded to the connection manager, which keeps a ticket variable. A new actor implementing the Query Dispatcher is spawned for the new connection, the current ticket added to a client index mapping it to the newly spawned Query Dispatcher, the ticket returned as the client id for subsequent calls, and incremented. For every other API call, the case class is also extracted from the JSON object and forwarded to the manager actor with the conection id. In turn, the manager forwards the case class to the Query Dispatcher actor the id maps to. Results returned from the Query Dispatcher are marshalled to JSON objects and sent in an HTTP response entity.

[8] https://gcc.gnu.org/releases.html

4.3.1.2 Query Dispatcher

The Query Dispatcher actor encapsulates all processing and communications with the backends. When a new instance is spawned, a MySQL connection is established through the wrapped JDBC Connector. The CBIR Server Module communications protocol is stateless, and requires no connection to be established. Requests to the Query Dispatcher are received as Scala case class messages, which represent a specific call with associated processing. Most calls are straightforward. The Dispatcher gets the parameters from the case class, issues the call on the backends and responds back to the Middleware Server Proxy API with the call return value, then marshalled to a JSON object and sent back as an HTTP response to the Middleware Client Proxy. Calls to *executeQuery* and *multimodalQuery* return Result Sets, which are not so trivially transmitted back to the Client Proxy. An Akka Streams Source is created from an Iterator of the Result Set, and the Source is then sent in an HTTP response, allowing the receiver to pull rows from the stream.

4.3.1.3 Result Set Row Iterator

The Result Set Iterator receives a Result Set in its constructor and wraps the calls to traverse it. Iterators are one of many options from which Akka Sources may seamlessly be created, thus we needed only to extend the Iterator class and implement the *next* and *hasNext* methods. The *next* method moves the Result Set cursor to the next row and writes its columns to byte arrays. Each row is sent through the stream as an array of byte arrays.

4.3.2 Middleware Client Proxy implementation

The Middleware Client Proxy has been implemented in nine Scala classes. These classes implement: the Middleware Client Proxy API, a client connection manager, the Proxy Engine, the Extended SQL Processing Module, a Database Schemas manager, the wrappers for the Crypto Providers that compose the Encryption Module, of which the CBIR wrapper consists in two classes, and a Result Set row iterator. The Middleware Server Proxy Connector is not here accounted for, as it overlaps with the Proxy Connector Interface implementation. An early prototype of our system with no support for CBIR operations consisted of only one Proxy, the Middleware Client Proxy. When the Server Proxy development began, considering how its API was identical to the Client Proxy's API, we repurposed the existing Proxy Connector Interface, which connected client applications to the Client Proxy, to connect the Client Proxy to the Server Proxy, with minimal changes.

4.3.2.1 Middleware Client Proxy API

The Middleware Client Proxy API is practically the same as its Server Proxy counterpart. It is an Akka HTTP server exposing the API calls in its HTTP endpoints, and encapsulates

a manager Akka actor that keeps a ticket variable to assign client connection ids and maintains a mapping of ids to the actors it spawns for the client connections. The actors spawned by the manager actor in the Client Proxy are Proxy Engine instances that manage the logistics of a connection's request processing in the Client Proxy. Request responses returned by the Proxy Engine are marshalled to JSON objects and sent to the client.

4.3.2.2 Proxy Engine

The Proxy Engine encapsulates the Extended SQL Processing Module. In the software architecture it would also wrap the Key Setup process, but this module is simplified to load encryption keys from local memory at the Client Proxy or generate them if they do not exist, and is instead contained in the specific wrappers for Crypto Providers. In our implementation, it receives the Scala case class requests and matches them to calls to the Extended SQL Processing Module's interface, extracting the case class member variables as parameters to the interface method. Returned values are sent back to the API to be sent back to the client as HTTP responses. The Processing Module handles all processing and communications with the Server Proxy.

4.3.2.3 Extended SQL Processing Module

The Extended SQL Processing Module implements the handling of calls made to the Client Proxy API forwarded through the Proxy Engine, and wraps the Encryption Module providing various convenience methods to easily make use of the Crypto Provider functionalities. An instance of this module receives the database name, username and password for the SQL connection on the remote backend in its constructor. It instantiates a Proxy Connector Interface to communicate with the Server Proxy and issues the connection request with the received parameters. All other calls are equally issued through the Proxy Connector Interface. A number of calls require no processing, and are simply forwarded to the Server Proxy (e.g a setAutoCommit call). For the SQL and multimodal SQL and image related calls, *executeUpdate*, *executeQuery* and *multimodalQuery*, statements are matched to defined regular expressions and its elements extracted. Sensitive elements are encrypted and the statement transformed as necessary. Sums and aggregate sums are substituted by calls to the User Defined Functions installed in the MySQL server backend. The encryption with required homomorphic constructs and the image processing in either the *multimodalQuery*, *addImage* and *searchImage* are transparently handled by Encryption Module.

4.3.2.4 Encryption Module

The Encryption Module consists of the wrappers for the Crypto Providers functionalities. The wrapper for homomorphic constructs provided by the HomoLib library implements not only the base methods to perform encryption and decryption with the DET, OPE and ADD schemes, but also the mappings of plaintext SQL datatypes to the schemes

supporting that datatype, and multiple utility methods that try to transparently provide those mappings to the Extended SQL Processing Module for different statement elements. We have also implemented a ciphertext cache as a HashMap with plaintexts as keys and ciphertexts as values. The cache exists only for the duration of the client connection, and while it uses some extra memory, it makes a big difference in performance, especially for populating databases with large amounts of data. The wrapper for CBIR functionalities is materialized in two components, one class that provides only the base CBIR encryption, decryption and feature extraction methods, and another class that wraps around the previous one and adds some additional processing. This additional processing is related to the CBIR Server Module communications protocol, that requires some byte headers for the request metadata, and expects the received image and feature bytes to be zipped.

4.3.2.5 Result Set Row Iterator

The Result Set Row Iterator is the same as the Server Proxy counterpart, with an additional step in the *next* method. For the *executeQuery* and *multimodalQuery* methods, the Extended SQL Processing Module generates information about the expected Result Set (number of columns, their plaintext SQL types, and encryption schemes). When the stream handle for the Result Set rows is returned from the query request to the Server Proxy, it is fed to this iterator along with the generated Result Set information. Each *next* call gets the next row from the stream and decrypts its columns according to the received Result Set information. As in the Server Proxy, an Akka Streams Source is seamlessly created from this iterator and returned to the API, which sets the Source as the HTTP response entity to the client.

4.3.3 Proxy Connector Interface

The Proxy Connector Interface is the smallest macro component and is implemented in three classes, one that implements the Client API, Middleware Client Proxy Connector, another for the Result Handler, and a minimally adapted class with I/O utility functions.

4.3.3.1 Client API and Middleware Client Proxy Connector

The Client API consists of the methods that are exposed to the client application to issue calls through the Middleware Client Proxy Connector. Each method packages the received parameters, if any, in a Scala case class expected by the Client/Server Proxy API. The case classes are marshalled to JSON objects and sent in the entity of an HTTP request to the Proxy. As mentioned, this API and Connnector was initially used to connect clients to the Client Proxy, but since the Server Proxy API is similar and expects the same Scala case classes, with minor adaptations it was repurposed to also connect the Client with the Server Proxy.

4.3.3.2 Result Handler

When a stream of Result Set rows is expected (by a call to *executeQuery* or *multimodalQuery*) the Proxy Connector links the Akka Streams Source received in the HTTP response entity to an Akka Streams SinkQueue and materializes the flow of the stream. From this point onward, rows are accumulated in the Sink as they are transmitted by the remote producer. The Result Handler wraps the SinkQueue, which provides methods to pull elements from the stream, and exposes methods to the client application that mimic a JDBC Result Set's behaviour, *isLast*, *next*, *getInt(columnIndex)*, *getLong(columnIndex)*, *getString(columnIndex)* and *getObject*. This Result Handler is also used in the Client Proxy, in its the Result Set Iterator.

4.4 Prototype Deployment Options

Our prototype was developed with a cloud based environment in mind. Initially, it was developed and tested in a Local deployment, and later relocated to the cloud, where different deployment options were considered. In the following subsections we describe the local deployment and the different cloud based environment deployments.

4.4.1 Local Deployment

The Local Deployment is the starting point of our experimental evaluation, and the primary development environment setup. All components were implemented in this setup and tested before we moved them to the Cloud Deployment setup. An i3-6100 @ 3.70 GHz (2 cores, 4 threads) with 8GB RAM desktop computer running Ubuntu 16.04 in a VirtualBox guest, the host being a Windows 10 OS, is used in this environment and runs every component of our architecture, client applications, Client Proxy, Server Proxy and backend components. With this setup we have minimal latency as all communications are contained inside the same machine.

4.4.2 Cloud Deployment

For the Cloud Deployment, a number of configurations was initially considered. Ideally, we'd like to leverage as many cloud services to facilitate system setup. Furthermore, we cannot modify cloud services to fit our needs, and use them as they are provided. For each of the considered configurations, we describe below the services they consist of, and why they were or weren't used for our experimental evaluation.

4.4.2.1 Pure Database as a Service Solution

A Database as a Service solution is a cloud provided database access service for users and applications. The advantages of such a service are that it does not require any infrastructure or software setup process from the user. Some configurations options are available,

if needed, but the databases are ready to use, out of the box. However, extensibility and advanced configurations are limited, and developers do not have full control of the DBMS. Deployment with such a solution was not possible, due to the need to install our User Defined Functions library that implements the Paillier cryptosystem for homomorphic addition over encrypted integers. As of the development time, this type of service did not allow installment of User Defined Functions, and as such this environment was not used for experimental evaluation.

4.4.2.2 Infrastructure as a Service Solution

In an Infrastructure as a Service solution, the cloud provides virtual machine instances with a variety of operating systems to choose from. Users may create new instances, and run them on cloud provided computing platforms. This solution offers users much more flexibility, as the virtual machine instance is a full blown computing system, only it is remotely accessed. It allows users to install software packages with no restrictions, and thus overcomes the limitation of the Database as a Service solution. We can spin up a new instance, install all software dependencies for our Middleware components, and test the system. The Encrypted Database and Encrypted Image Repositories are persisted in the virtual machine's disk. An additional solution, that we present in the next subsection, is also possible, but for experimental evaluation purposes we used this deployment.

In this setup, the client applications remain at the laptop machine, and the Client Proxy component in the desktop machine in the local network in our labs. The Middleware Server Proxy and server side backends are all moved to the Infrastructure as a Service cloud environment. A major factor is introduced in this setup, the higher latency between the Client Proxy and the Server Proxy. We expect this latency to become the dominant factor in the total time of a query's execution, and thus reduce the perceived overhead of the cryptographic operations taking place in the Client Proxy.

We have resorted to an Amazon Web Services EC2 instance for this deployment. The instance is an m5.xlarge (15 ECUs, 4 vCPU, 2.5GHz, Intel Xeon Platinum 8175 with 16GB RAM) running Ubuntu Server 16.04.

4.4.2.3 Hybrid Deployment Solution

In the Infrastructure as a Service solution, encrypted data is stored in the virtual instance's disk. We may want to separate the storage from the runtime support for our deployment. Cloud service providers offer storage services, such as key-value stores where data items are stored with a key, and are retrieved given the key. This is a perfect fit for our Encrypted Image Repository, where encrypted images are stored and retrievable by their identifiers. The Encrypted Database however, must remain in secondary storage accessible to the DBMS. We have not tested the Hybrid deployment as the adaptation of the CBIR Server Module to persist the encrypted images and retrieve them from the key value store would

require extra work for little benefit to our objectives. The integration with the key value store would offer durability guarantees, but this is outside the scope of this thesis.

4.5 Prototype Implementation Metrics

In order to give some additional information regarding the extension of the prototype implementation in its current version we present some indicative software engineering metrics. In these metrics we consider:

1. The code extension;
2. Software packaging;
3. Sizing metrics of the executables;
4. The repercussion of the metrics in the different components of the global software architecture. As some implemented modules are leveraged from previously existing components, we identify the relevance of such components in the support of the implementation.

The metrics are summarized in table 41.

Software Architecture Module	Lines of source code	Development Language	Prototype
Query Dispatcher	224	Scala	FD
CBIR Server Module	3249	C++	L
User Defined Functions	204	C	FD
Middleware Server Proxy API	310	Scala	FD
Proxy Engine	176	Scala	FD
Extended SQL Processing Module	756	Scala	FD
Encryption Module	3184	Scala Java	ED
Middleware Client Proxy API	369	Scala	FD
Result Handler	88	Scala	FD
Middleware Proxy Connector	323	Scala	FD
Test Code	16646	Scala Java C++	ED

Table 41: Software Engineering Metrics

In table 41, the Prototype column refers to the development effort of the modules in the developed solution. In this column, Fully Developed (FD) indicates a component completely developed from scratch. Leveraged (L) components are those that were integrated

into the solution with minimal or no effort from previous implementations, which is the case for the CBIR Server Module. Extended Development (ED) refers to modules which are based in existing implementations and that required a significant effort to adapt to our final prototype, as is the case for the Encryption Module and Test Code components.

The Test Code module refers to both 1307 lines of source code that were written from scratch to perform experimental evaluation, as well as three distinct extensions of a Java implementation of the industry standard TPC-C benchmark [28], which amount to 15339 lines of code. The details of these three adaptations are provided in Chapter 5.

The CBIR Server Module is completely based in an implementation of the IES-CBIR [24] server side functionalities. Given the dependencies in the software versions that must be matched with the remaining sources of our prototype implementation, we decided to include this module as part of the project repository source tree.

In terms of executable components, the Middleware corresponds to a JAR file of 23.5 MB. This archive includes the necessary Scala libraries, the Middleware Client and Server Proxy executables, the Client library, and all Test Code executables except for the TPC-C benchmark adaptations, provided in the repository but outside this JAR file. The TPC-C adaptations must be compiled independently and generate JAR files with sizes 2.88, 26.4 and 26.4 MB (the last two include the Middleware JAR as a dependency). Finally, the CBIR Server Module executable has a size of 627 KB.

Chapter 5

Experimental Evaluation and Validation

In this chapter we address the results of our experimental evaluation and observations. As mentioned in the previous chapter, there are two deployments used for testing, the Local and the Cloud Deployments. For the Local Deployment, all components including client applications are located in the same machine, and as such latency is minimized. In the Cloud Deployment, the Middleware Server Proxy and backend components are moved to the cloud infrastructure, thus introducing an end-to-end latency between the Client and the Server Proxies. We start by presenting the experimental evaluation results for the Local Deployment, followed by the results of repeating the experiments in the Cloud Deployment. The experimental evaluations focus on the following items:

- The raw costs of cryptographic primitives, with measurements isolated from additional processing. A comparison of encryption operations with and without the ciphertext caching optimization is also presented.
- The costs of different statement types, most of which require composite processing stages and involve multiple cryptographic primitives, and how much overhead is introduced *vs* the same statement types in plaintext, involving no processing.
- Throughput results for three adaptations of an implementation of the industry standard TPC-C benchmark: an adaptation for plaintext data with no processing, an adaptation using our Middleware for encrypted SQL text only, and an extension of the latter including multimodal SQL text and image queries. The three are compared in terms of the overhead introduced by our Middleware.
- The costs of performing an image repository setup.

5.1 Local Deployment Testbench Evaluations

In this section we present our analysis of the results obtained for the previously described evaluations, in the Local Deployment setting.

5.1.1 Raw Costs of Cryptographic Primitives

To establish the raw costs of cryptographic primitives, namely the partially homomorphic constructs and the Content Based Image Retrieval operations, we used the following datasets:

- For the partially homomorphic constructs, two datasets of size 10000 were randomly generated, one consisting of 1KB strings to test the DET scheme encryption performance, and another of 4 byte integers for OPE and ADD. The encryption of these datasets generates the corresponding ciphertext datasets, then used to test the decryption operation. For the ADD scheme's homomorphic sum operation, a separate dataset of 1000 encrypted integers was previously generated, and each consecutive pair is added for a total of 500 sums.
- For Content Based Image Retrieval operations, we resorted to a dataset of 1000 images, previously uploaded to the Encrypted Image Repository, and its feature index structure trained and indexed. The time costs for this setup are addressed in a separate section.

In regards to the partially homomorphic constructs, we additionally present the cost differences of using and not using the ciphertext caching optimization for the encryption operations.

5.1.1.1 Partially Homomorphic Schemes

Table 51 displays the average times of the scheme operations, in milliseconds. Here the ciphertext caching optimization is disabled. Additionally, we note that the homomorphic sum operations were not performed in the context of a MySQL User Defined Function call, but instead in a standalone program for this sole purpose.

Scheme	Encryption	Decryption	Homomorphism	Plaintext size
DET	0.181 ms	0.101 ms	-	1024 byte
OPE	0.048 ms	0.011 ms	-	4 byte
ADD	55.494 ms	52.360 ms	Sum: 0.006 ms	4 byte

Table 51: Average cost of cryptographic primitives

As expected, the ADD scheme is by far the most computationally expensive. It is approximately 200% more expensive than both DET and OPE. DET is the fastest scheme, when we take into account the much higher volume of data it encrypts in this test. Next we repeated the measurements on the same datasets, but with the ciphertext caching optimization enabled for encryption. The results for the average execution times are seen in table 52, also translated in terms of throughput in table 53 where we directly compare the throughputs of each scheme with (*), and without ciphertext caching. For every scheme we observe that the average time of execution has improved tenfold. However, we also note that to verify this improvement, we purposely limited the plaintext domain to 1000 distinct elements, a tenth of the dataset's size. Logically, the improvement introduced

by the ciphertext caching optimization will vary depending on the plaintext domain size. Smaller domains will see a greater improvement compared to sparse domains.

Scheme	No ciphertext caching		With ciphertext caching	
DET	0.181	ms	0.018	ms
OPE	0.048	ms	0.006	ms
ADD	55.494	ms	5.552	ms

Table 52: Comparison of encryption average times without and with ciphertext caching

Scheme	Encryption		Decryption		Homomorphism	
DET	5516.317	kB/s	9903.665	kB/s	-	
DET *	54430.197	kB/s			-	
OPE	81.057	kB/s	328.840	kB/s	-	
OPE *	700.773	kB/s			-	
ADD	0.070	kB/s	0.075	kB/s	Sum: 200683.594	kB/s
ADD *	0.708	kB/s			-	

Table 53: Throughputs of cryptographic primitives, in kB/s

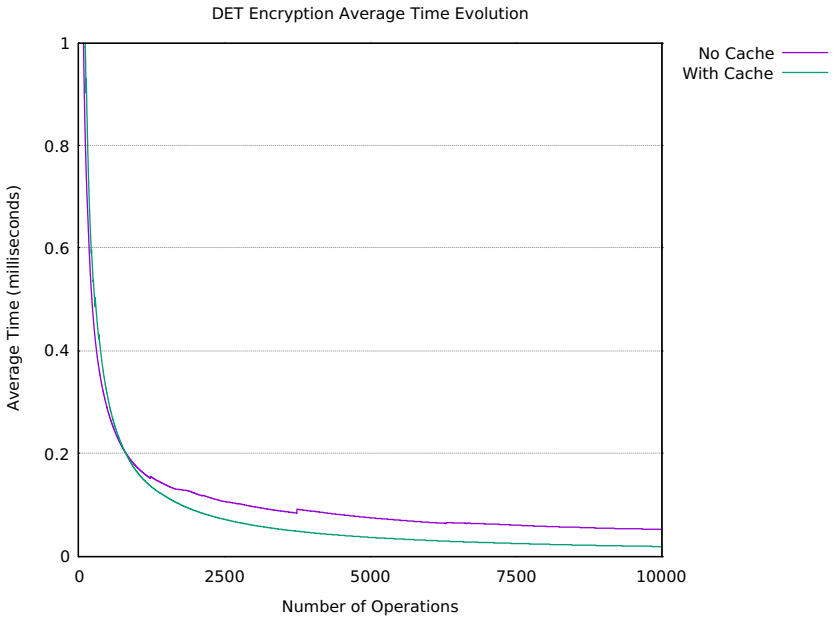


Figure 51: DET Encryption Average Time Evolution

Figures 51, 52 and 53 show the evolution of the average encryption times for DET, OPE and ADD, respectively. For all three, the first call to the *encrypt* method takes much longer than subsequent calls, hence why the average time starts at a disproportionately high value, off the chart. As more and more operations are performed, we can observe that the average times for encryption with ciphertext caching tend to decrease to lower values than with no caching. However, this is most noticeable for the ADD scheme, where

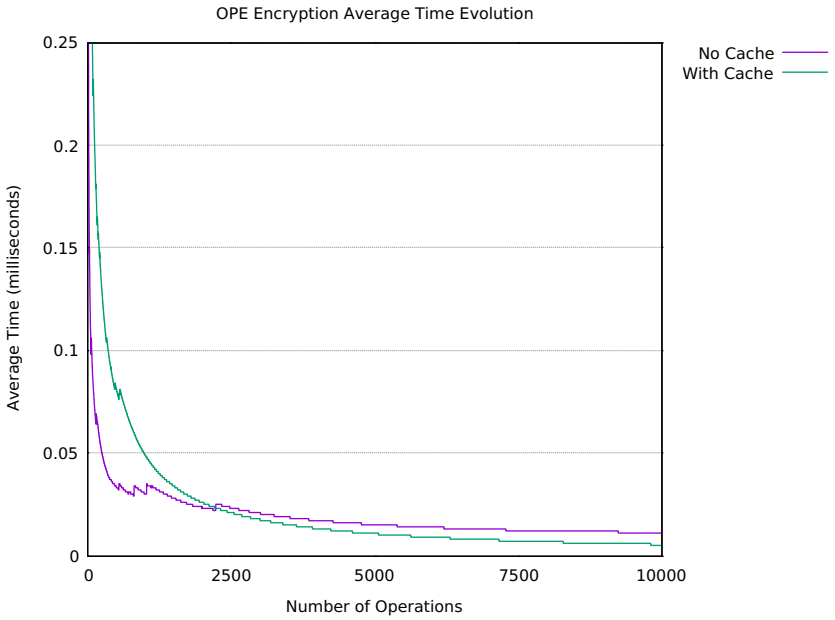


Figure 52: OPE Encryption Average Time Evolution

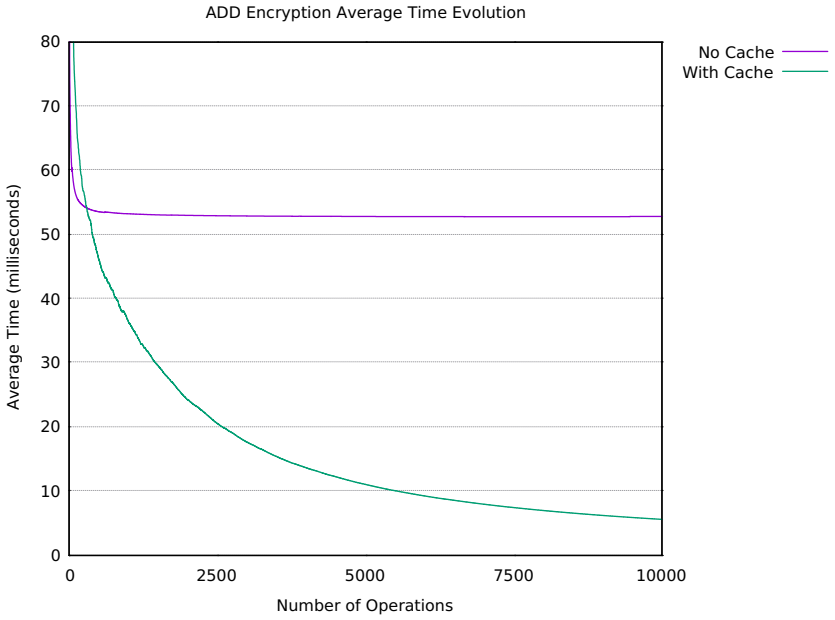


Figure 53: ADD Encryption Average Time Evolution

the average times for encryption without caching stabilizes slightly above 50 ms, but with ciphertext caching, continues to decrease and drops below 10 ms.

In terms of spatial overhead, ADD once again is overwhelmingly expensive, as seen in table 54. DET ciphertexts have a volatile but minimal ciphertext expansion overhead, due to padding, and conversion to Base64 that we perform to store them as text in the SQL databases. In this particular case the overhead is 31.6%. OPE has a fixed size overhead, as the scheme maps plaintext 4 byte integers to 8 byte ciphertext longs. ADD presents the highest percentual overhead of almost 200%.

Scheme	Plaintext size	Ciphertext size	Ciphertext expansion %
DET	1024 Bytes	1408 Bytes	31.6%
OPE	4 Bytes	8 Bytes	66.6%
ADD	4 Bytes	1233 Bytes	198.7%

Table 54: Ciphertext expansion percentages

5.1.1.2 Content Based Image Retrieval operations

The values shown in table 55 are the average execution times for each operation performed on each of the 1000 images in the image dataset, as well as their throughputs for the average image size of 0.112 MB. The Encryption operation, in fact includes the Feature Extraction process, hence the disparity between this operation and the Decryption operation, which would otherwise be odd for a symmetric scheme. The Search operation is measured at the CBIR Server, from the moment a request is received to when it is completed and the result returned to the client. Already we can conclude that the Search operation measured at the caller will take, on average, at least the sum of the Feature Extraction Only and the Search operations to return a list of the top scoring similar image identifiers.

Operation	Average Time	Calculated Throughput for Average Image Size
Encryption & Feature Extraction	427.011 ms	0.262 MB/s
Decryption	1.346 ms	83.209 MB/s
Feature Extraction Only	426.465 ms	0.262 MB/s
Search	279.126 ms	0.401 MB/s

Table 55: Average costs of CBIR operations, in milliseconds, for 1000 images each

5.1.2 Query Type Performance Comparison

To compare the different types of SQL statements identified in Chapter 3, we designed a simple database schema and a set of SQL statements to test each type, with variable

workloads. The statements are executed in a cycle for 1000 runs, where in each run each type of statement is executed multiple times. For this test ciphertext caching is enabled, and the database was populated with a total of 2400 rows distributed between three tables. Figure 54 displays a direct comparison of the average execution times for each statement type, for a MySQL client issuing plaintext queries to MySQL Server through JDBC, and for a client using our Middleware to encrypt the queries.

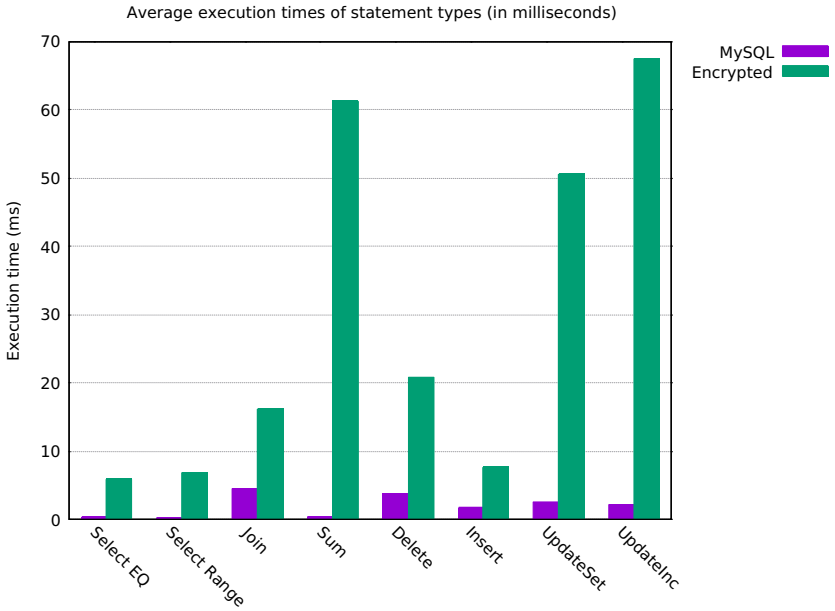


Figure 54: Average execution times of statement types

Even with the ciphertext caching optimization, we can observe that the processing of queries introduces a very significant overhead. It is expected that the overall performance drops, as the processing of queries is not a single cryptographic operation but instead a composite of operations (parsing, encryptions with multiple schemes). This is especially true of the UpdateSet and UpdateInc statement types. While most other statements default to avoiding the ADD scheme, UpdateSets, which set row fields to a literal value, must encrypt values with ADD if the update is issued on integer fields. UpdateIncs process updates of the form $field = field + 1$, which require ADD, and additionally an intermediate query to fetch all values encrypted with OPE that would become stale. Sums (aggregate sums) are the second most costly statement type, on average. From the previous measurements of the ADD scheme's homomorphism, we would believe that Sum would perform better. However, as mentioned, those measurements were not performed in the context of a MySQL User Defined Function call, and did not capture the times to serialize the encrypted numbers from raw data into the necessary representations to enable the homomorphic sum operation. Despite the high overheads, for the Cloud Environment deployment we expect the introduced end to end latency to become the dominant factor of a query's total execution time, thus bridging the gap and reducing the

perceived overheads.

We did not include the Multimodal SQL and Image query in the previous figure, as it has no direct mapping to the pure SQL operations. Instead, we directly compare it in figure 55 against the most expensive statement type excluding itself, UpdateInc, and a Select EQ. The latter was chosen for comparison due to the similar processing of both queries. A Multimodal Query is a Select (EQ / Range) only with the added processing steps of extracting an image's features, the issuing of an image search to the CBIR Server at the Server Proxy, and finally the appending of the extra WHERE condition. As we've seen in previous measurements, the average time to extract and image's features plus the search operation at the CBIR Server amounts to at least 706 ms, approximatelly.

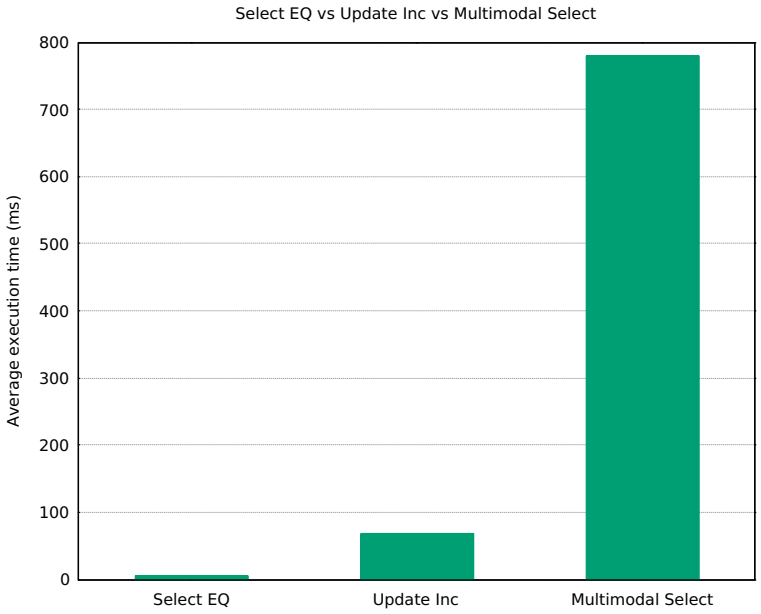


Figure 55: Select EQ *vs* Update Inc *vs* Multimodal Select

5.1.3 TPC-C Benchmarks Comparison

For this experimental evaluation we have adapted a Java implementation of the TPC-C specification, available here [28]. In our repository we include three different adaptations, called *tpcc-plain*, *tpcc-crypt* and *tpcc-multimodal*, for plaintext SQL, encrypted SQL only, and encrypted multimodal SQL and image. Our middleware does not support all datatypes from the original database schema, and as such, floating point fields were changed to integers, and datetime fields to varchar in all adaptations for fairness. The *tpcc-plain* adaptation uses the original JDBC connector while the *tpcc-crypt* and *tpcc-multimodal* use our Middleware. Table 56 displays the number of statement types present in the TPC-C workload, and their percentages in relation to the total number of statements. We note, however, that the statements are distributed across five different transactions. At runtime, transactions are selected for execution, but the selection does not

follow a uniform distribution.

Statement Type	Select Eq	Select Range	Join	Sum	Delete	Insert	Update Set	Update Inc
Occurrences	13	7	1	1	1	4	5	4
%	36.1	19.4	2.8	2.8	2.8	11.1	13.9	11.1

Table 56: TPC-C Query Mix Statement Types

In the *tpcc-multimodal* adaptation, instead of adding Multimodal Select statements to the transactions, some of the existing Select statements were modified. The number of statement types and respective percentages are presented in table 57. Given the computational overheads of our Middleware, as well as the Local Deployment machine's limitations, tests with a higher number of concurrent transactions struggled to complete and often ran into deadlocks. Adding more statements, especially Multimodal Selects which are the most computationally expensive, would further exacerbate this issue. As a side effect of this choice to adapt existing statements in the transactions, a Result Set may be either smaller or bigger than it would originally be, due to its intersection or union with the image results. Some statements depend on the results of previous Selects, and thus their execution times are directly affected and this is reflected in the overall throughput of transactions.

Statement Type	Select Eq	Select Range	Join	Sum	Delete	Insert	Update Set	Update Inc	Multimodal Select
Occurrences	7	7	1	1	1	4	5	4	6
%	19.4	19.4	2.8	2.8	2.8	11.1	13.9	11.1	16.7

Table 57: Extended TPC-C for Multimodal Text and Image Query Mix Statement Types

5.1.3.1 TPC-C Benchmarks for SQL Data

Figure 56 displays the throughput results of the three implementations, measured in transactions per minute (TpmC). We note that the scale for this figure graph is logarithmic. Not only are the throughput differences very evident, the performance of *tpcc-crypt* actually worsens when the benchmark is run with more concurrent clients connections (22.321, 21.823, 20.991, 21.654 TpmC), while the throughput of *tpcc-plain* sharply increases up until 8 concurrent connections (3757.287, 5691.398, 7318.296 and 6387.605 TpmC). However, we should take into consideration that all Middleware and backend components are running in the same machine. Thus, for multiple client connections, the machine must simultaneously handle Client and Server Proxy processing, and SQL queries, while in the Cloud Deployment the workload is distributed across two machines.

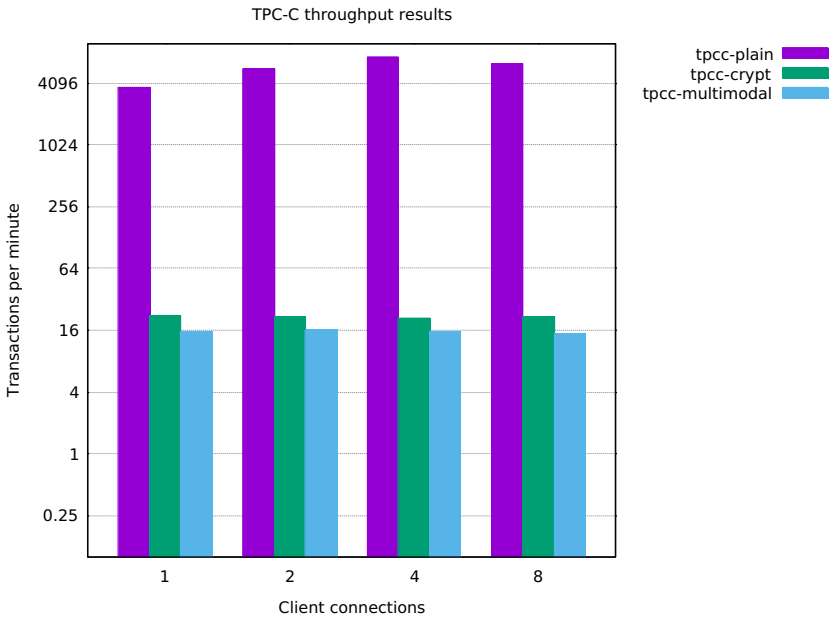


Figure 56: TPC-C throughput results for SQL Data in the Local Deployment

5.1.3.2 Extended TPC-C Benchmark for Multimodal SQL Text and Image Queries

Figure 57 presents a comparison of the *tpcc-crypt* and *tpcc-multimodal* adaptations. As expected for the conditions of the Local Environment, *tpcc-multimodal* has slightly worse performance than *tpcc-crypt*. However, *tpcc-multimodal* appears to scale better with more concurrent connections. For both these benchmarks at 8 concurrent connections, however, the results are not conclusive as the system struggled to keep up with the workload.

5.1.4 Image Uploading and Indexing Costs

A dataset of 1000 images totalling 112 MB is uploaded in this test. The measurements are for the total time to upload the images, and for the training and indexing of the Image Feature Index, which is performed after all images have been transferred. It took a total of 8 minutes and 48 seconds to upload all images from the dataset, and a total of 43 minutes and 36 seconds to perform the training and indexing operations of the Image Feature Index structure. At the CBIR Server Module, there is minimal spatial overhead. The images take up the same 112 MB, while the indexing structures take up 4.5 MB, a total of 116.5 MB.

5.2 Cloud Environment Testbench Evaluations

In this section we repeat the tests performed for the Local Environment Deployment and directly compare the results obtained in both environments. Some tests are not repeated since the results would be redundant. Most tests for the raw costs of cryptographic

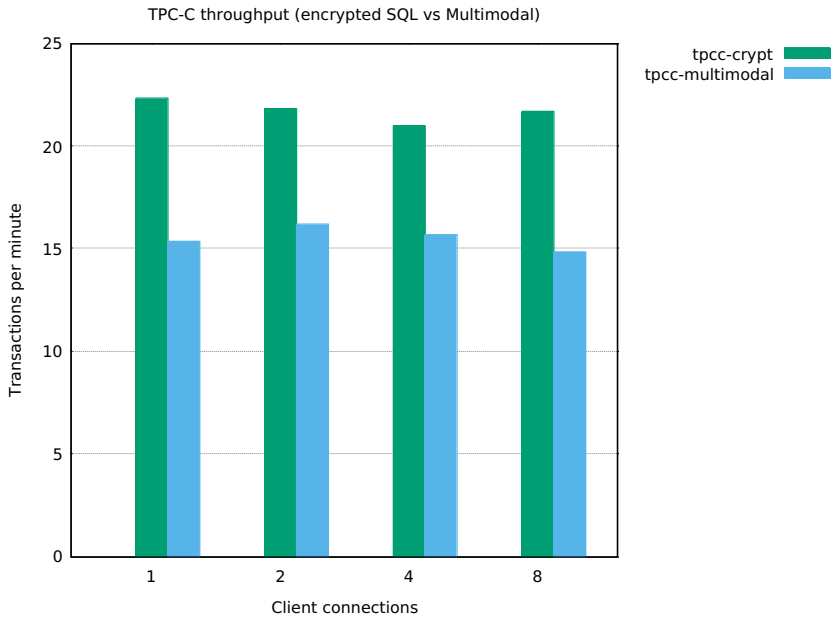


Figure 57: TPC-C throughput results (encrypted SQL vs Multimodal)

primitives are not repeated because the operations for those tests are still performed in the same machine as in the Local Environment. Thus we only perform tests where operations are shifted to the cloud machine of the Cloud Environment.

5.2.1 Raw Costs of Cryptographic Primitives

As mentioned in the section introduction, only tests where operations are shifted from the local machine to the cloud machine of the Cloud Environment are repeated. Those tests are the ADD scheme's homomorphic sum operation, and the Content Based Image Retrieval Search operation.

5.2.1.1 Partially Homomorphic Schemes

Since only the ADD scheme's sum operation is repeated in this environment, only the dataset of 1000 ADD ciphertexts is needed. Each consecutive pair of items is added for a total of 500 sums. The average time for this operation is 0.006 milliseconds, exactly the same time as in the Local Environment measurements.

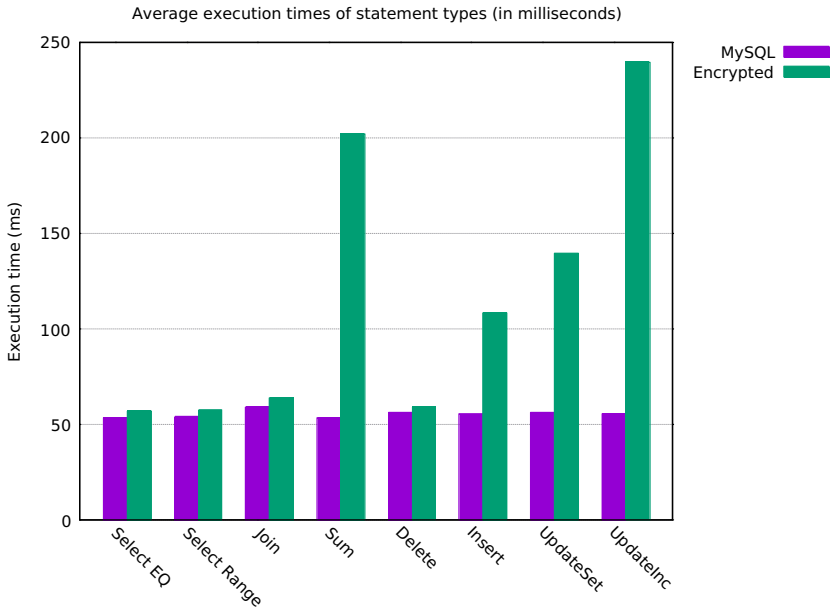
5.2.1.2 Content Based Image Retrieval operations

The image dataset (1000 images previously encrypted, uploaded to the cloud machine and there indexed) is used to perform the Search operation measurements in the cloud machine from the moment a Search request arrives. This test yields an average of 279.127 milliseconds, practically the same as the Local Environment's results.

Despite the cloud machine being considerably more powerful, there seems to be no discernible performance benefit, neither for the ADD scheme's sum nor the Content Based Image Retrieval's Search operations.

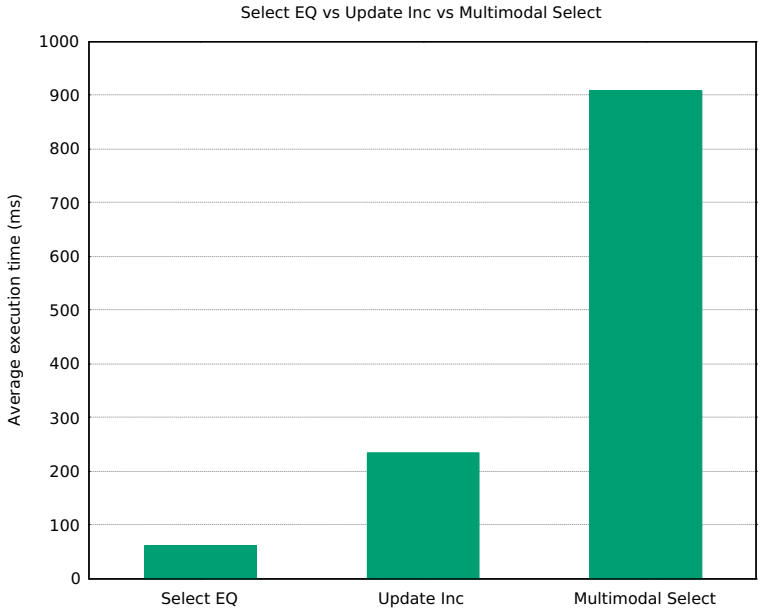
5.2.2 Query Type Performance Comparison

For the comparison of query type performances we use a database schema with 3 tables, populated with 2400 rows distributed across those tables, for both plaintext MySQL and an encrypted database and queries. A set of SQL statements for each type is executed for 1000 runs. We directly compare these results with the Local Environment Deployment's results.



As can be seen in figure 5.2.2, and comparing to figure 54, the average total times for all operations have increased considerably, both for plaintext MySQL and using our Middleware to encrypt data and queries. The perceived performance differences between plaintext and encrypted SQL operations, however, have been amortized by the introduced end-to-end latency. Even for the Sum, UpdateSet and UpdateInc statement types with encryption, which stand out as the most expensive and with higher overheads, the difference to plaintext is not as drastic.

Figure 5.2.2 displays a comparison of average Multimodal Select, UpdateInc and Select EQ statement execution times. Again, the introduced end-to-end latency reduces the performance differences, albeit only slightly. Table 58 shows the increase in average total time for these statement types. As we can see, the latency overhead has a lot more impact on the SelectEQ and UpdateInc (164.9% and 110.0%) statements than the Multimodal Select statements (15.2%).



	SelectEQ	UpdateInc	Multimodal Select
Local	5.911 ms	67.455 ms	780.168 ms
Cloud	61.495 ms	232.322 ms	908.401 ms
Difference (%)	164.9 %	110.0 %	15.2 %

Table 58: Average time increase between Local and Cloud deployment

5.2.3 TPC-C Benchmarks Comparison

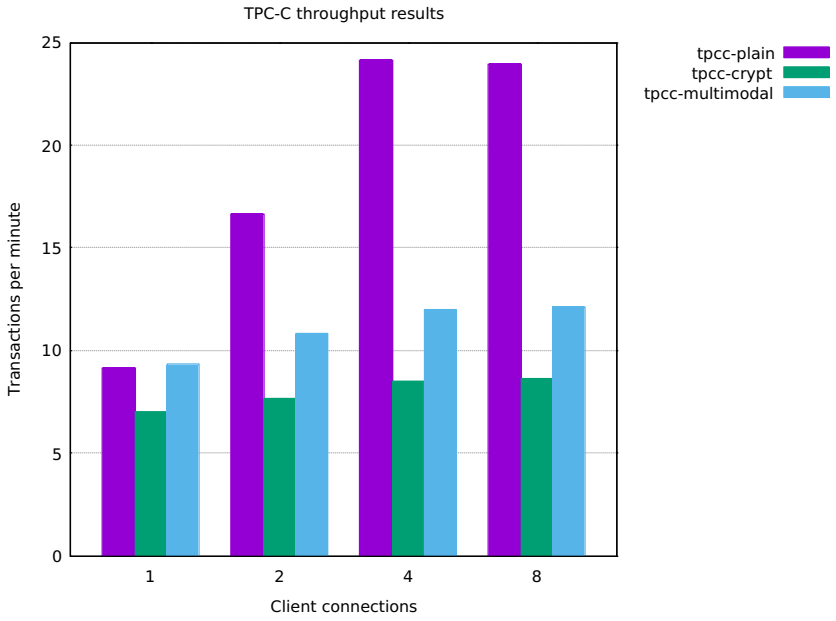
The TPC-C implementations used for this test are the same as those previously introduced for the same purpose in the Local Environment Deployment, namely the *tpcc-plain*, *tpcc-crypt* and *tpcc-multimodal* implementations. We measure the throughput of all three in the Cloud Environment Deployment, and compare the differences in performance in this deployment to the differences in the Local Environment Deployment.

5.2.3.1 TPC-C Benchmark for SQL Data

Figure 5.2.3.1 shows the throughput of all three implementations in transactions per minute (TpmC). While in figure 56, for the local TPC-C results, the TpmC axis is displayed in logarithmic scale, in figure 5.2.3.1 the scale is linear. This aspect alone should already be indicative of the improvement in the perceived performance of the system. For one client connection only, the *tpcc-crypt* implementation reaches throughput performance close to *tpcc-plain*. However, with more simultaneous connections, the performance of *tpcc-plain* significantly increases while *tpcc-crypt* barely improves. Both seem to stabilize at 8 simultaneous client connections.

The improvement and decrease in performance differences between *tpcc-plain* and *tpcc-crypt* implementations in the Local Environment and the Cloud Environment can be

attributed in part to the end-to-end latency introduced in the latter. Additionally, while in the Local Environment all Middleware components ran in a single local machine, in the Cloud Environment Deployment the Middleware components are distributed across two machines.



5.2.3.2 Extended TPC-C Benchmark for Multimodal SQL Text and Image Queries

The throughput results for the *tpcc-multimodal* implementation are also displayed in 5.2.3.1. For one client connection, its throughput actually surpasses *tpcc-plain*. However, as previously mentioned in the Local Environment section, this is only due to having altered existing Select statements from the original SQL transactions, which directly affect posterior queries and their Result Sets. Overall, the total workload of *tpcc-multimodal* can be inferior to that of *tpcc-plain* and *tpcc-crypt*. For multiple concurrent connections however, *tpcc-plain* throughput greatly increases, while the improvement for *tpcc-multimodal* is moderate, but consistently above *tpcc-crypt* throughput.

5.2.4 Image Uploading and Indexing Costs

The complete dataset of 1000 images was uploaded to the cloud side. The total time to upload the image dataset was 10 minutes and 36 seconds, an increase compared to the Local Deployment's 8 minutes and 48 seconds. This increase is both expected and the only possible outcome, due to the introduced latency between the local and the cloud machines, whereas in the Local Deployment there was none.

The training and indexing operations amounted to 23 minutes and 23 seconds, a significant improvement over the Local Environment time of 43 minutes and 36 seconds for the same operations. In the Cloud Environment, training and indexing is performed

at the cloud machine, which is more powerful, rather than in the more modest local machine. In total, the time to setup the image repository is of 33 minutes and 59 seconds, against the Local Environment's total time of 52 minutes and 24 seconds.

Chapter 6

Conclusions

The objective of this dissertation was the design, implementation and experimental evaluation of a security middleware solution, implementing a client/client-proxy/server-appliance software architecture, to support the execution of applications requiring online multimodal queries on "always encrypted" data. With the proposed solution protected data is maintained in outsourced cloud storage solutions, such as DBaaS (Database as a Service) backends. Our solution addressed a practical and usable solution that can be regarded as an innovative approach for future privacy-enhanced cloud DBaaS deployments.

The proposed solution includes the support for SQL based text queries enhanced with searchable encryption image retrieval capabilities. The support combines a multimodal searchable encryption environment supporting extended SQL queries. We implemented a prototype for the designed solution and conducted an extensive experimental benchmark evaluation, in order to observe the effectiveness, latency and performance conditions.

Considering the realization of our objectives, we summarize the following achieved contributions:

- The design, implementation and deployment of the described middleware solution, defining its system model and architecture, as well as, explaining the processing modules materialized in two main components: the Client Proxy (supporting client side operations) and a cloud server remote appliance (supporting and dispatching the remote queries from clients to be executed on encrypted data stored in the server or cloud side).
- Design and implementation of a hybrid support for SQL querying, extending the text based queries combined with content based image search and retrieval functions. This support consists of SQL operations combined with indexing and searching by color similarity, to retrieve multimodal confidential images stored in the server or cloud side data repository.

The implementation prototype is available [25] and ready for use. At the same time it is a base platform for future extensions and future reserach work of other forms of searchable encryption queries over cloud enabled multimodal databases.

6.1 Experimental observations

Our experimental observations show that the solution is valid and a viable approach to address our objectives. In general, our tests show that we achieve interesting resuts and very acceptable indications, in terms of performance and latency benchmarking observations of searchable queries involving encrypted data.

We summarize the following observations:

- There are costs inherent to the use of some cryptographic constructs, namely those using more complex partial homomorphic encryption algorithms or queries combining more heavyweight partial homomorphic constructions and image search and retrieval operations. However, through the ciphertext caching optimizations for such partially homomorphic constructs, it is possible to improve the performance of our solution, always preserving the confidentiality requirements.
- For query type performance comparison, there is a significant performance difference between plaintext MySQL queries and encrypted Middleware searchable encrypted queries on MySQL encrypted databases. This difference is more obvious when comparing local settings (where the clients, the database and our Middleware searchable encrypted queries are all supported in a single machine). However, even considering the global overhead in this setting, the least performant SQL statement type displays an average excution time below 70 ms, within satisfactory bounds for real time queries and interactivity purposes in many applications.
  In the local setting the Multimodal Select operation (involving color similarity queries on encrypted images) has no plaintext counterpart to compare to. These queries have an average execution time around 800 ms, a much higher value compared to queries involving only textual data, but we still consider this acceptable for some applcations.
  When evaluating the performance of our solution in a Cloud Environment setting, we observe that the introduced end to end latency (between client and the cloud enabled database / data repository) amortizes the above performance differences visible in the local setting, to the point that some statement types involving tested queries display an almost negligible difference. Some queries are still significantly more expensive, with the least performant executing in 250 ms on average, which is a satisfactory indication.
  The particular case of Multimodal Select (involving similarity searchable encryption on encrypted images) shows an average execution time that increases to slightly over 900 ms. However, we also consider this value as quite acceptable for many purposes.
- From the implemented TPC-C benchmark analysis, we observed that SQL statements are executed in the context of transactions. For the Local Environment we observe some performance problems of both TPC-C adaptations that use our Middleware, namely *tpcc-crypt* and *tpcc-multimodal*, in comparison to *tpcc-plain* which runs the TPC-C benchmark on a plaintext database. Furthermore, when we increase

from one to multiple concurrent client connections, while *tpcc-plain* sees an increase in performance, the other two display an expected degradation of their transaction throughput result.
In the Cloud Environment, the end to end latency once again amortizes the performance differences to the point of *tpcc-plain* and *tpcc-crypt* being comparable.

6.2 Future work directions

There are some interesting future trends to address in the wake of the results obtained in this dissertation. We emphasize the following challenges as some relevant issues that could be addressed:

- The current implementation of our prototype used in the evaluations can be optimized, to concurrently dispatch the more heavyweight queries using a pre cached ciphertext solution, for example.
- The enhancement of exception handling in the Middleware Proxies (Client and Server) components, specifically in regards to enhance the input validation support.
- It is possible to overcome the current rigidness and limited extensibility of the regular expressions used to parse SQL statements in the Middleware Client Proxy's Query Processor. An alternative should be found to replace and optimize the regular expressions and to improve the parsing processing for extended SQL statements.
- Finally, we suggest the future integration of our Middleware solution in a real application that could explore the hybrid support for SQL queries enhanced with Content Based Image Retrieval operations with confidentiality guarantees. This could allow a new experimental observation of the designed solution in the context of a specific application. An example would be an application managing Electronic Medical Records, which deals with sensitive patient data regarding not only personal information, but encrypted exam results, medical histories, and specific medical images, searchable by similarity queries supported by our extended multimodal queries.

Bibliography

[1] *20 Million People Fall Victim to South Korea Data Leak*. Acc. Jun 5th. URL: http://www.securityweek.com/20-million-people-fall-victim-south-korea-data-leak.

[2] M. Abdalla, M. Bellare, D. Catalano, E. Kiltz, T. Kohno, T. Lange, J. Malone-Lee, G. Neven, P. Paillier, and H. Shi. "Searchable Encryption Revisited: Consistency Properties, Relation to Anonymous IBE, and Extensions". In: *Journal of Cryptology 21.3, pp. 350-391* (2008).

[3] *Accenture latest to breach client data due to misconfigured AWS server*. Acc. Jun 5th. URL: http://www.healthcareitnews.com/news/accenture-latest-breach-client-data-due-misconfigured-aws-server.

[4] *Amazon EC2 Pricing*. Acc. Jun 5th. Amazon Web Services. URL: https://aws.amazon.com/ec2/pricing/.

[5] A. Arasu, K. Eguro, R. Kaushik, and R. Ramamurthy. "Transaction Processing on Confidential Data using Cipherbase". In: *ACM SIGMOD* (2014).

[6] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. "Orthogonal Security With Cipherbase". In: *6th Biennial Conference on Innovative Data Systems Research (CIDR '13)*. 2013.

[7] S. Bajaj and R. Sion. "TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality". In: *IEEE Transactions on Knowledge and Data Engineering, Vol. 26, NO. 3* (2014).

[8] F. Baldimtsi and O. Ohrimenko. "Sorting and Searching behind the curtain". In: *Proceedings of the 9th International Conference on Financial Cryptography and Data Security*. 2015.

[9] H. Bay, T. Tuytelaars, and L. V. Gool. "SURF: Speeded Up Robust Features". In: *Proceedings of the 9th European Conference on Computer Vision - ECCV'06. Springer, pp.404-417*. 2006.

[10] M. Bellare, A. Boldyreva, and A. O. Neill. "Deterministic and Efficiently Searchable Encryption". In: *Proceedings of the 27th International Cryptology Conference - CRYPTO'07, pp. 535-552*. 2007.

[11] *Benefits at a Glance*. Acc. Jun 5th. Amazon Web Services. URL: https://aws.amazon.com/application-hosting/benefits/.

[12] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. "DEPSKY: Dependable and Secure Storage in a Cloud-of-Clouds". In: *EuroSys'11, Salzburg, Austria*. 2011.

[13] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. "Order-Preserving Symmetric Encryption". In: *Proceedings of the 28th Annual International Conference on Advances in Cryptology: the Theory and Applications of Cryptographic Techniques, pp. 224-241*. 2009.

[14] D. Boneh, G. D. Crescenzo, R. Ostrovsy, and G. Persiano. "Public key encryption with keyword search". In: *Proceedings of the 23rd Annual International Conference on the Theory and Applications of Cryptographic Techniques - EUROCRYPT'04. Springer, pp. 506-522*. 2004.

[15] K. D. Bowers, A. Juels, and A. Oprea. "HAIL: A High-Availability and Integrity Layer for Cloud Storage". In: *CCS'09, Chicago, Illinois, USA*. 2009.

[16] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. "Privacy-Preserving Multi-Keyword Ranked Search over Encrypted Cloud Data". In: *IEEE Transactions on Parallel and Distributed Systems 25.1, pp. 222-233* (2014).

[17] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. "Dynamic searchable encryption in very large databases: Data structures and implementations". In: *Proceedings of the 21st Annual Network and Distributed System Security Symposium - NDSS'14. Vol. 14*. 2014.

[18] O. Chowdhury, D. Garg, L. Jia, and A. Datta. "Equivalence-based Security for Querying Encrypted Databases: Theory and Application to Privacy Policy Audits". In: *22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015.

[19] 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 - CCS'06, pp. 79-88*. 2006.

[20] R. Datta, D. Joshi, J. Li, and J. Z. Wang. "Image Retrieval". In: *ACM Computing Surveys 40.2, pp 1-60* (2008).

[21] *Disgruntled Employees and Data: a Bad Combination*. Acc. Jun 5th. URL: http://www.datacenterknowledge.com/archives/2017/05/31/disgruntled-employees-data-bad-combination.

[22] C. Dong, G. Russello, and N. Dulay. "Shared and searchable encrypted data for untrusted servers". In: *Journal of Computer Security 19.3, pp. 367-397* (2011).

[23] T. ElGamal. "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". In: *IEEE Transactions on Information Theory 31 (4): 469–472* (1985).

[24] B. Ferreira, J. Rodrigues, J. Leitão, and H. Domingos. "Privacy-Preserving Content-Based Image Retrieval in the Cloud". In: *SRDS '15*. 2015.

[25] *fm_correia/porksoda - Bitbucket*. Acc. Jun 5th. URL: https://bitbucket.org/fm_correia/porksoda.

[26] C. Gentry. "A fully homomorphic encryption scheme". PhD thesis. Stanford University, 2009.

[27] *Giant Equifax data breach: 143 million people could be affected*. Acc. Jun 5th. URL: http://money.cnn.com/2017/09/07/technology/business/equifax-data-breach/index.html.

[28] *GitHub - AgilData/tpcc: Java implementation of the TPC-C benchmark*. Acc. Jun 5th. URL: https://github.com/AgilData/tpcc.

[29] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. "SiRiUS: Securing remote untrusted storage". In: *Proceedings of the 10th Annual Network and Distributed System Security Symposium - NDSS'03, pp. 131-145*. 2003.

[30] *google/encrypted-bigquery-client · GitHub*. Acc. Jun 5th. Google. URL: https://github.com/google/encrypted-bigquery-client/blob/master/tutorial.md.

[31] H. Hacıgümüş, B. Iyer, C. Li, and S. Mehrotra. "Executing SQL over Encrypted Data in the Database-Service-Provider Model". In: *SIGMOD* (2002).

[32] F. Hahn and F. Kerschbaum. "Searchable Encryption with Secure and Efficient Updates". In: *Proceedings of the 21st ACM Conference on Computer and Communications Security - CCS'14. ACM, pp. 310-320*. 2014.

[33] *Healthcare data breaches haven't slowed down in 2017, and insiders are mostly to blame*. Acc. Jun 5th. URL: http://www.fiercehealthcare.com/privacy-security/healthcare-data-breaches-haven-t-slowed-down-2017-and-insiders-are-mostly-to-blame.

[34] G. R. Hjaltason and H. Samet. "Index-driven similarity search in metric spaces". In: *ACM Transactions on Database Systems 28.4, pp. 517-580* (2003).

[35] C. Y. Hsu, C. S. Lu, and S. C. Pei. "Image Feature Extraction in Encrypted Domain with Privacy-Preserving SIFT". In: *IEEE Transactions on Image Processing 21.11, pp. 4593-4607* (2012).

[36] *IBM Database Encryption Expert for encryption of data at rest*. Acc. Jun 5th. IBM. URL: https://www-01.ibm.com/support/knowledgecenter/SSEPGG_9.7.0/com.ibm.db2.luw.admin.sec.doc/doc/c0053466.html.

[37] *Insider Steals Data of 2 Million Vodafone Germany Customers*. Acc. Jun 5th. URL: http://www.securityweek.com/attacker-steals-data-2-million-vodafone-germany-customers.

[38] J. Jeon, V. Lavrenko, and M. Manmatha. "Automatic image annotation and retrieval using cross-media relevance models". In: *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR'03, pp. 119-126*. 2003.

[39] S. Jeong, C. S. Won, and R. M. Gray. "Image retrieval using color histograms generated by Gauss mixture vector quantization". In: *Computer Vision and Image Understanding 94.1, pp. 44-66* (2004).

[40] K. S. Jones. "A statistical interpretation of term specificity and its application in retrieval". In: *Journal of Documentation 28.1, pp. 11-21* (1972).

[41] K. S. Jones, S. Walker, and S. E. Robertson. "A probabilistic model of information retrieval: development and comparative experiments: Part 2". In: *Information Processing & Management 36.6, pp. 809-840* (2000).

[42] S. Kamara and C. Papamanthou. "Parallel and dynamic searchable symmetric encryption". In: *Proceedings of the 7th International Conference on Financial Cryptography and Data Security - FC'13, pp. 1-15*. 2013.

[43] S. Kamara, C. Papamanthou, and T. Roeder. "Dynamic searchable symmetric encryption". In: *Proceedings of the 19th ACM Conference on Computer and Communications Security - CCS'12. ACM, pp. 965-976*. 2012.

[44] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout. *SLIK: Scalable Low-Latency Indexes for a Key-Value Store*. Tech. rep. Stanford University, 2015.

[45] M. Kuzu, M. S. Islam, and M. Kantarcioglu. "Efficient Similarity Search over encrypted data". In: *Proceedings of the 28th IEEE International Conference on Data Engineering - ICDE'12, pp. 1156-1167*. 2012.

[46] L. Lamport, R. Shostak, and M. Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programing Languages and Systems, 4(3):382–401, July 1982* (1982).

[47] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout. "Implementing linearizability at large scale and low latency". In: *25th ACM Symposium on Operating Systems Principles (SOSP'15)*. 2015.

[48] D. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision 60.2, pp. 91-110* (2004).

[49] W. Lu, A. Swaminathan, A. L. Varna, and M. Wu. "Enabling Search over Encrypted Multimedia Databases". In: *IS&T/SPIE Electronic Imaging 7254, pp. 725418-725418-11* (2009).

[50] C. Mavroforakis, N. Chenette, A. O'Neill, G. Kollios, and R. Canetti. "Modular Order-Preserving Encryption". In: *ACM SIGMOD International Conference on Management of Data*. 2015.

[51] *More than 316,000 patient blood tests exposed in breach linked to home monitoring company*. Acc. Jun 5th. URL: http://www.fiercehealthcare.com/privacy-security/data-breach-medical-records-blood-tests-patient-home-monitoring-kromtech-security.

[52] M. Naveed, M. Prabhakaran, and C. A. Gunter. "Dynamic Searchable Encryption via Blind Storage". In: *Proceedings of the 35th IEEE Symposium on Security and Privacy - S&P'14*. 2014.

[53] M. Naveed, S. Kamara, and C. V. Wright. "Inference Attacks on Property-Preserving Encrypted Databases". In: *CCS'15*. 2015.

[54] D. Nister and H. Stewenius. "Scalable recognition with a vocabulary tree". In: *Proceedings of the 19th IEEE Conference on Computer Vision and Pattern Recognition - CVPR'06. IEEE, pp. 2161-2168*. 2006.

[55] R. Ostrovsky. "Efficient Computation on Oblivious RAM". In: *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*. 1990.

[56] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. "The RAMCloud Storage System". In: *ACM Transactions on Computer Systems (TOCS), Volume 33 Issue 3* (2015).

[57] P. Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *EUROCRYPT'99*. 1999.

[58] R. A. Popa, E. Stark, J. Helfer, and S. Valdez. "Building web applications on top of encrpyted data using Mylar". In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. 2014.

[59] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. "CryptDB: Protecting Confidentiality with Encrypted Query Processing". In: *SOSP '11*. 2011.

[60] *Pricing Overview - How Azure Pricing Works*. Acc. Jun 5th. Microsoft. URL: https://azure.microsoft.com/en-us/pricing/.

[61] R. L. Rivest, A. Shamir, and L. Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM (CACM) 21.2, pp. 120-126* (1978).

[62] R. L. Rivest, A. Shamir, and M. L. Dertouzos. "On data banks and privacy homomorphisms". In: *Foundations of Secure Computation 4.11, pp. 169-180* (1978).

[63] J. Rodrigues and H. Domingos. "TSKY: A Dependable Middleware Solution for Data Privacy using Public Storage Clouds". MA thesis. Universidade Nova de Lisboa, 2013.

[64] J. Rodrigues, B. Ferreira, J. Leitão, and H. Domingos. "TMS - A Trusted Mail Repository Service using Public Storage Clouds". In: *ACM/IFIP/USENIX 14th International Middleware Conference and Workshops*. 2013.

[65] A. Shamir. "How to share a secret". In: *Communications of the ACM, Volume 22 Issue 11* (1979).

[66] D. X. Song, D. Wagner, and A. Perrig. "Practical Techniques for searches on encrypted data". In: *Proceedings of the 21st IEEE Symposium on Security and Privacy - S&P'00, IEEE, PP. 44-55*. 2000.

[67] *SQL Server Encryption*. Acc. Jun 5th. Microsoft. URL: https://technet.microsoft.com/en-us/library/bb510663.aspx.

[68] N. B. of Standards. *Federal Information Processing Standards Publication 46: Data Encryption Standard*. pub-NIST, 1977.

[69] N. I. of Standards and Technology. *Federal Information Processing Standards Publication 197: Announcing the Advanced Encryption Standard (AES)*. pub-NIST, 2001. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[70] E. Stefanov, C. Papamanthou, and E. Shi. "Practical Dynamic Searchable Encryption with small leakage". In: *Proceedings of the 21st Annual Network and Distributed System Security Symposium - NDSS'14*. 2014.

[71] M. J. Swain and D. H. Ballard. "Color Indexing". In: *International Journal of Computer Vision 7.1, pp. 11-32* (1991).

[72] J. Tatemura, O. Po, W.-P. Hsiung, and H. Hacıgümüş. "Partiqle: an elastic SQL engine over key-value stores". In: *SIGMOD '12 Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, Pages 629-632*. 2012.

[73] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. "Processing analytical queries over encrypted data". In: *PVLDB, vol. 6, no. 5* (2013).

[74] *US data leak: 198 million Americans' personal information accidentally released*. Acc. Jun 5th. URL: http://www.independent.co.uk/news/world/americas/us-politics/us-leak-data-americans-personal-information-deep-root-analytics-republican-national-committee-a7798251.html.

[75] C. Wang, N. Cao, K. Ren, and W. Lou. "Enabling Secure and Efficient Ranked Keyword Search over Outsourced Cloud Data". In: *IEEE Transactions on Parallel and Distributed Systems 23.8, pp. 1467-1479* (2012).

[76] *What is BigQuery?* Acc. Jun 5th. Google. URL: https://cloud.google.com/bigquery/what-is-bigquery.

[77] *Why we need to improve cloud computing's security*. Acc. Jun 5th. URL: https://phys.org/news/2017-10-cloud.html.

[78] W. K. Wong, B. Kao, D. W. L. Cheung, R. Li, and S. M. Yiu. "Secure query processing with data interoperability in a cloud database environment". In: *ACM SIGMOD International Conference on Management of Data*. 2015.

[79] C. Wueest, M. B. Barcena, and L. O'Brien. *Mistakes in the IaaS cloud could put your data at risk*. Acc. Jun 5th. URL: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/mistakes-in-the-iaas-cloud-could-put-your-data-at-risk.pdf.

[80] J. Zobel and A. Moffat. "Inverted files for text search engines". In: *ACM Computing Surveys 38.2, p. 6* (2006).