# Encrypted Cassandra

Client-Side Encryption through Query-Rewriting of CQL

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science

im Masterstudium

Wirtschaftsinformatik

Eingereicht von:
Arjol Qeleshi

Angefertigt am:
Institut für Wirtschaftsinformatik - Data & Knowledge Engineering

BeurteilerIn:
o. Univ.Prof. Dr. Michael Schrefl

Mitbetreuung:
Dr. Michael Karlinger

August 2015

# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

## Kurzfassung

Cloud Dienste bieten Unternehmen attraktive Vorteile. Bedenken über die Vertraulichkeit der Daten verhindern aber oftmals ihren Einsatz. Ziel dieser Masterarbeit ist es, eine Cloud-kompatiblen Verschlüsselungsansatz für NoSQL Datenbanken vorzustellen, sowie eine Beispielimplementation für Cassandra bereitzustellen. Die Verschlüsselung erfolgt auf Client-seite, so dass nur verschlüsselte Daten in die Cloud gelangen, wodurch ihre Vertraulichkeit gewahrt wird. Zusätzlich ist die Verschlüsselung so ausgelegt, dass das Durchsickern von Klartextmustern durch die Verschlüsselung vermieden wird. Um diese Ziele zu erreichen ist eine Abstimmung zwischen Performance, Funktionalität und Sicherheit notwendig.

## Abstract

Cloud services offer attractive benefits to businesses, yet their adoption remains hampered by confidentiality concerns. The scope of this thesis is to present a cloud-compatible encryption scheme for NoSQL databases, as well as to provide a sample implementation for Cassandra. The encryption occurs on the client side, thus ensuring that only encrypted data will be shared with the cloud provider in order to maintain its privacy. In addition to this, the encryption is designed with the goal of avoiding plain text data patterns leaking through it. To achieve this, various tradeoffs between performance, functionality and security need to be made.

# Contents

# 1 Introduction

## 1.1 Motivation

The goal of Cloud Computing is to offer computing as a utility[2]. This allows businesses to outsource issues like data center operations and instead focus on their core competencies, while at the same time allowing cloud providers to pick those very same issues up and specialize on them. However, while this may seem like an obvious win-win scenario for the overall business landscape, there are several issues upon closer look, like concerns about platform lock-ins or non-obvious performance implications, which still hamper the adoption of the cloud.

One major concern is maintaining the confidentiality of the data stored in the cloud[2]. This issue has several aspects, as the data needs to be protected not only from third-party attacks on the cloud service provider, but also from direct abuse by provider, as well as possibly from state-level actors, depending on the specific legal circumstances that may apply.

This is a very real issue instead just some theoretical concern, as demonstrated by McCandless' current overview of news articles on major data breaches classified by method of breach and number of records stolen [31], as well as press releases of cloud providers claiming governments' spying activities to negatively impact their business performance [14].

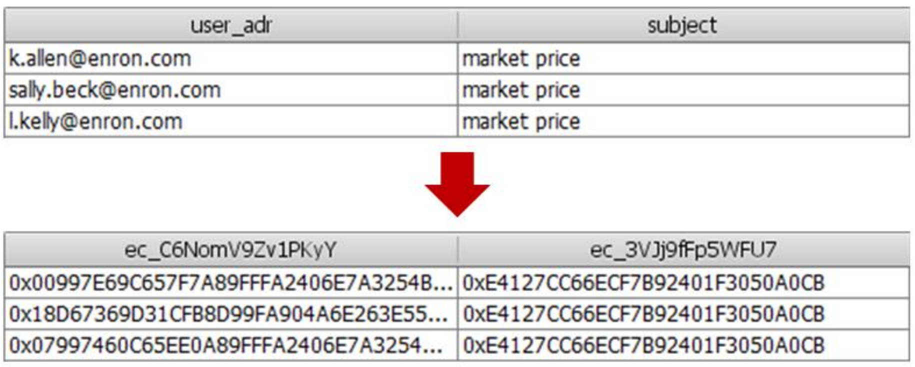
## 1.2 Related Work

Given the popularity of cloud services as well as confidentiality concerns, a lot of research work has focused on creating new encryption primitives that are useful for cloud-hosted databases [7, 20, 21, 9], as well as how to best utilize the currently existing encryption technology to solve specific problems in cloud-hosting scenarios [25, 34, 8]. In addition to the research work focusing on specific issues, there are also efforts on implementing these results in already existing commercial databases [35, 3, 4].

## 1.3 Objective

The goal of this thesis is an encryption scheme for Cassandra databases that ensures the privacy of confidential data. The data is encrypted on the client side, i.e. before being shared with the cloud provider, so that no further entities beyond one's own organization need to be trusted.

Additionally to this, the data is encrypted with the goal of patterns contained in the plain text data to not leak through the encryption. Patterns leaking through the encryption would occur whenever attribute-level plain texts would be encrypted with the same key, resulting in identical ciphertexts after the encryption, if the original plain texts were identical, thus giving off the frequencies of the various pieces of data stored in the database (e.g. see Figure 1).



| user_adr | subject |
|---|---|
| k.allen@enron.com | market price |
| sally.beck@enron.com | market price |
| l.kelly@enron.com | market price |

| ec_C6NomV9Zv1PKyY | ec_3VJj9fFp5WFU7 |
|---|---|
| 0x00997E69C657F7A89FFFA2406E7A3254B... | 0xE4127CC66ECF7B92401F3050A0CB |
| 0x18D67369D31CFB8D99FA904A6E263E55... | 0xE4127CC66ECF7B92401F3050A0CB |
| 0x07997460C65EE0A89FFFA2406E7A3254... | 0xE4127CC66ECF7B92401F3050A0CB |

**Figure 1:** Despite the data being encrypted, it is still obvious that all the values in the second column are identical.

The basic concept for the encryption scheme presented in this thesis was designed by the Department for Business Informatics - Data & Knowledge Engineering at the Johannes Kepler University in Linz. It has already been implemented in several other Bachelor and Master theses for various target databases. Ultimately the scheme is a generalized version of the approach presented by Feigenbaum et al. [19]. A summarized description of the approach can also be found in Subsection 3.8 of Schneier's book Applied Cryptography [38].

The encryption process is as transparent as possible, to application programmers that utilize the database, and is implemented as an additional layer within the client drivers of the database. Thus it is fully compatible with any cloud-hosting scenarios, since only the client-portion of the architecture needs to be modified.

However, due to Cassandra's specifics, some functionality needs to be sacrificed and several tradeoffs between functionality, performance as well as security need to be made. For some parts it is not possible to fully maintain the transparency of the encryption layer – especially since the encryption involves non-trivial performance implications, despite being otherwise API compatible, or exposing various additional configuration options, in order to allow further tuning of aforementioned tradeoffs to specific the user's needs.

The encryption scheme itself is independent of specific encryption algorithms and can utilize any symmetric block cipher. However, for some database queries that allow for more involved searches, a symmetric order preserving encryption algorithm is also needed. One such algorithm currently exists [7], its security specifics are however still an open research issue [42].

Encryption schemes that would allow for multiple database users with different privilege levels are outside the scope of this thesis. The goal is to protect the data against foreign entities and thus the assumption is made that users within one's own organization can be fully trusted. That said, the encryption scheme does of course still support multiple users accessing the database simultaneously – the only limitation is that all of these users share the same privileges at the encryption level.

Since all of these objectives can be achieved with symmetric encryption primitives, asymmetric ones have not been considered, due to generally being more complex to understand and less efficient to execute. Additionally, since providing a practical encryption scheme along with an implementation is a major goal of the thesis, only encryption algorithms that are practically viable have been considered.

Although the presented encryption scheme focuses on Cassandra specifically, the general concept behind it is suitable and can be applied to any other Extensible Record Store (a subset of NoSQL databases [10]).

## 1.4 Idea

The encryption is implemented as an additional layer inside the database client drivers, and works by rewriting queries to the database such that the plain text values in them are replaced with ciphertexts. The returned result sets (if any) are decrypted before being handed over to the user (i.e. the application).

This has the benefit that no modifications need to be made to the cloud-hosted parts of the database architecture, as well as the process being mostly transparent to users by default, since most of the database API remains unchanged.

In order to avoid patterns leaking through the encryption (as shown in Subsection 1.3), instead of static keys the data access path itself (which the database uses for accessing the data) is used. This ensures that every individual attribute value is encrypted with a key that is distinct to the keys of every other attribute value in the database. Additional care needs to be taken for the database's non-scalar data types however.

This does have some implications as far as functionality is concerned though – the main one being that inequality or range searches (e.g. "all records with attribute x being larger than 10") are not possible any more. However, most of them can be alleviated by the use of order preserving encryption, at the cost of runtime performance and somewhat lower security guarantees. Additionally, some database functionality (mainly secondary indices and the data dictionary) needs to be reimplemented on the client-side, since the server-side implementation cannot be used, due to it being impossible to perform the desired operations on the data without being able to decrypt it first.

## 1.5 Document Structure

Section 2 provides an in-depth description of relevant encryption primitives as well as a discussion and theoretical evaluation of alternate approaches to database encryption. In order to contrast the various approaches to one another, the objectives outlined in Subsection 1.3 are expanded upon, creating a criteria catalog.

Section 3 presents the general encryption scheme and how it can be applied to Cassandra specifically, as well as showing what the query rewriting looks like. Additionally it also evaluates the encryption scheme against the criteria catalog presented in Section 2.

Section 4 focuses on the implementation and the details necessary in order to achieve the set goals.

Section 5 showcases the encryption scheme with an example use case and provides a general idea of the performance implications involved, by comparing queries on an encrypted Cassandra database to an unencrypted one.

Section 6 provides the conclusion to the thesis by summarizing the achieved goals, current drawbacks and limitations, as well as providing an outlook to further work in this area.

# 2 Encryption

This section presents a brief overview of approaches to encrypting data within databases as well as the relevant cryptographic primitives, in order to introduce the tools and research results available, as well as the tradeoffs involved.

## 2.1 Cryptographic Primitives

Cryptographic primitives are algorithms that are used to perform the actual encryption and decryption of raw chunks of data. They are utilized in more involved encryption schemes in various ways.

### 2.1.1 Symmetric Block Ciphers

Symmetric block ciphers are algorithms to encrypt and decrypt one fixed-size chunk of data based on an encryption key [38]. The size of the chunk of data that the cipher can operate on is called the block size, whereas the size of encryption keys supported is called the key size.

Conceptually block ciphers can be thought of as large lookup tables that simply translate one chunk of data into another. Their actual implementation however is more involved than this, since tables large enough to be useful for encryption purposes are much too large for current computers to handle.

In order to be able to encrypt data that is larger than the block size of the cipher a mode of operation needs to be used. Modes of operation specify how the data is to be partitioned, how the block cipher needs to be applied to the individual sub-blocks, and how those sub-blocks are then to be merged again. They differ in e.g. how efficiently they can be implemented or parallelized, and ultimately also their security characteristics. Some modes of operation, like the Counter mode ("CTR", [30]), turn block ciphers into stream ciphers thus allowing them to be used on data of arbitrary size, while other modes, like Cipher Block Chaining ("CBC", [38]), require the data to be a multiple of the underlying block cipher's block size.

In order to be able to use modes like CBC on data of arbitrary sizes nonetheless, the data needs to be padded up to the next full block-size boundary first and the padding then later on discarded after decryption. Several standards for paddings exist, like Public-Key Cryptography Standard #7 (PKCS#7, [26]) or ISO 10126-2 [5], which use predefined patterns or random data for padding respectively.

Another issue with block ciphers is that identical pairs of encryption key and plain text result in identical ciphertexts. This is an issue because patterns of the plain text can leak through the encryption, when encrypting larger messages. Most modes of operation try to mitigate this issue by chaining the result of a prior execution of the block cipher to the next one, thus effectively randomizing the next iterations result, while still ensuring that the final ciphertext is decryptable. However this still leaves the first iteration of the block cipher to produce identical ciphertexts for identical plain texts for the very first block of the message. In order to avoid this, an Initialization Vector (IV) is used for the randomization of the first block's ciphertext.

IVs have a similar effect on the ciphertext as the encryption key. They are however still distinctively different in that they do not need to be kept secret and do not need to be randomized or uniformly distributed. The only requirement they need to satisfy is to not be used more than once for one encryption key.

### 2.1.2 Homomorphic Encryption

After encrypting data with a symmetric block cipher no further operations can be performed on it prior to decryption. This is of course a design goal of block ciphers and part of the security they provide. Nevertheless, in certain situations, it might still be useful to be able to e.g. compare two different encrypted values and determine the larger one without needing to first decrypt them.

In contrast to block ciphers, homomorphic encryption schemes allow for certain operations to be performed over ciphertexts, at the cost of security and performance.

Also unlike block ciphers, homomorphic algorithms are still very much research-in-progress currently and not accepted industry-standard solutions.

#### *2.1.2.1 Fully Homomorphic Encrpytion (FHE)*

The first fully homomorphic encryption scheme published was Gentry's [20]. It used weaker homomorphic properties of asymmetric encryption to first bootstrap a circuit for its own decryption and then through recursion allow for arbitrary computations to be executed over the encrypted data. The main issue with it is that it currently is still not practically viable, due to its runtime performance as well as resource-usage in terms of data size. Nevertheless, efforts have been made at optimized implementations of the algorithm [21].

Newer fully homomorphic algorithms, like [9] which is currently implemented by HELib[1], offer much better performance. However, while the implementations can be used to demonstrate the effects of the encryption, they are still not practically viable for usage on real-world data volumes [18].

An additional limitation to all of these approaches is that operations over ciphertexts introduce noise into them, requiring the values to be periodically re-encrypted.

#### *2.1.2.2 Order Preserving Encryption (OPE)*

Unlike fully homomorphic encryption, order preserving encryption only ensures that the ciphertexts maintain the order relations of their original plain texts between them.

One such algorithm currently exists [7]. It encrypts the data by mapping values of a fixed domain into a fixed range based on a probabilistic scheme. An implementation of it is provided in CryptDB [35]. While its runtime performance is several orders of magnitude slower than that of block ciphers, it still is fast enough to be viable for practical purposes, as demonstrated by CryptDB.

Other than the performance, a further concern is that the security awarded by the encryption is still an open research issue [42].

### **2.1.3 Hashes and Message Authentication**

A (cryptographic) hash function is a one-way function that transforms an arbitrary-sized block of data into a fixed-sized one, while trying to distribute the domain of possible inputs over the range of possible outputs as evenly as possible. Being a “one-way” function means that it is difficult to provoke hash collisions – i.e. construct an input to a hash function that results in a specific predetermined value after being hashed.

Hashes are used primarily for generating encryption keys for block ciphers from user-provided passwords and for integrity checks of transmitted or stored data. Additionally, keyed-hashes can be used for the authentication of ciphertexts. This is more commonly referred to as “message authentication” [38].

[1] https://github.com/shaih/HElib - HELib currently only offers direct support for addition and multiplication operations. Other higher-level operations need to be implemented by the users using the toolset provided by the library.

The need for message authentication arises in symmetric cryptosystems since there is no way to ensure that a ciphertext has not been modified by any malicious entities after its creation. Trying to decrypt corrupted ciphertexts in turn can lead to possibly corrupted plain texts or provide an attack surface for side-channel [40] attacks.

In order to avoid these attacks, so-called message authentication codes (MACs) are used on the final ciphertext after encryption, in order to generate an authentication tag (which can vary in length, depending on the specific algorithm) which is then added to the ciphertext. Prior to decryption the MAC is applied to the ciphertext again, and checked against the originally generated authentication tag. If both values match, then the ciphertext is unlikely to have been modified (or suffered data corruption) and is thus ready for decryption.

Other than authenticating ciphertexts after their encryption with a MAC in a separate step, there also exist modes of operation for block ciphers, like EAX[2] [6], which combine these steps, thus reducing the potential for user errors when applying the cryptographic primitives.

## 2.2 Encryption for Databases

### 2.2.1 Page-Level Encryption

Databases can be encrypted using straight-forward data encryption tools, like block ciphers, at the file system or database page level. This option is offered by many current commercial products like SQL Server[3] and Oracle[4], and is ideal in that it provides the best performance possible and does not impact the functionality of the database. It does require the encryption key to be known to the database operator however.

The issue becomes severely more complex if the data also needs to be protected from the database operator itself, since in this case the database server software must not be able to decrypt the values, while still somehow needing to be able to meaningfully process it. This is an issue that has received much attention and several research results have been published so far.

If the database operator cannot be trusted, then the data needs to be encrypted prior to being stored in the database.

### 2.2.2 Buckets

An example of encrypting data on the client side was presented by Feigenbaum et al., where records are encrypted based on one defining attribute in them, which is later used to look them up [19]. This works, since users looking for the record also implicitly provide the decryption key (i.e. the afore-mentioned defining attribute) for it.

However there are two issues with this approach:

- If the defining attribute of the records is not unique, then the distribution of attribute values across the records will leak through the encryption.
- The approach is limited in that it only allows for point-queries in the database. To also allow for range queries more elaborate schemes are needed.

One way to allow for range queries and the indexing of data, despite it being encrypted, is to group similar records into buckets and then encrypt the buckets, while exposing a defining common

[2] EAX is not an abbreviation and instead the algorithm's actual name.
[3] https://msdn.microsoft.com/en-us/library/bb510663.aspx
[4] http://www.oracle.com/technetwork/database/options/advanced-security/index-099011.html

fragment of the data in the bucket as plain text [25]. This allows the database to perform searches and indexing at the coarse-grained level of the buckets. After the relevant buckets to a query have been returned by the database, the client then proceeds on to decrypt them and then perform any remaining fine-grained filtering and sorting operations necessary to satisfy the user query.

Additionally, since the buckets contain multiple individual rows, no patterns of individual identical values across the rows will leak through the encryption, other than the defining parts of the buckets that have been left intentionally unencrypted.

However buckets are not ideal:

- They require some data (or at least fragments of data) to remain unencrypted.
- They incur network overheads due to the entire buckets of data having to be transmitted in response to a query instead of only the actually relevant result set.
- They require post processing of the query on the client side, in order to perform any work necessary that requires access to the plain text data.
- They preclude the usage of any database functionality that directly modifies data on the server side, e.g. incrementing a value stored in an integer field.
- If foreign keys are used in the database, then the unencrypted defining parts of the buckets will leak those relationships through the encryption, at the coarse-grained level of the buckets.

### 2.2.3 CryptDB

An alternative to buckets, in order to avoid their issues, which is used by CryptDB [35], is to encrypt each attribute individually and maintain a database of encryption keys (or initialization vectors), so that each individual value will be encrypted with its very own key. This way, identical plain texts would still result in different ciphertexts after the encryption, with overwhelming probability.

Additionally, an order preserving algorithm (see Subsection 2.1.2.2) is used for sorting and indexing operations, allowing them to operate on attribute-level granularity rather than coarse-grained buckets. The use of OPE however precludes the use of random IVs, and thus causes identical plain text values in OPE columns to have identical ciphertexts across the various rows in the table.

Although CryptDB uses OPE, it is not strictly necessary to actually encrypt the data here – any transformation on the plain texts that preserves both their confidentiality as well as their order relation to one another would be a valid choice. As such, instead of actually encrypting the data based on a fixed mathematical function (like the OPE algorithm in CryptDB does), an adaptive stateful scheme can be used, which assigns ciphertexts to plain texts upon request, and keeps track of all assignments so far, so that it can ensure that the ciphertexts maintain the order relation of the original plain texts [34]. Alternatively, if all plain text values that will need to be protected are known beforehand, it is possible to construct a perfect minimal hash function that provides mappings of plain text values into ciphertexts only for those known plain texts, thus ensuring any ordering requirements [8].

CryptDB also allows for performing some operations (like addition) directly on the server side on the ciphertexts, without needing to first decrypt them. They use the basic homomorphic properties of asymmetric encryption, as discussed by Paillier [32], in order to achieve this.

Furthermore, CryptDB's implementation is designed such that it requires an actual proxy server to intercept the user queries and rewrite them. It leverages this to improve the performance of some algorithms, like OPE, by batching up and optimizing simultaneous requests from multiple users, as well as allowing to provision high-performance servers for the encryption proxy, so that any computationally intensive cryptographic tasks can be completed quickly.

The downsides of the proxy server of course are increased network overhead for simple queries that are not computationally intensive to encrypt, as well as requiring some trusted party to maintain the server.

#### 2.2.4 Trusted Hardware

The other alternative to these various algorithms is the use of trusted hardware. In this scenario the database itself, and thus the database operator, are not able to decrypt the data as such, however they can delegate any tasks that require the decryption of data, such as comparing two values or adding up multiple values, to a trusted component that is capable of decryption, performing the requested operations, and then returning a (possibly encrypted) result.

The core assumption here is that this trusted component is resistant to tampering. In this case the database would never need to touch unencrypted data, while still being able to offer the full breadth of its functionality to users.

Examples of solutions that make use of trusted hardware are TrustedDB [4] and Cipherbase [3]. They are configurable in how the data is encrypted on a per-column level and allow for avoiding the leakage of patterns through the encryption by using randomized IVs. The encryption details and keys of the specific schemas are stored within the trusted modules.

In order to achieve all of this, the database needs to be integrated with the encrypted module.

## 2.3 Comparison of Database Encryption Approaches

The goal of this subsection is to compare the approaches and technologies presented in Subsection 2.2 to each other along some common criteria, and thus to provide an overview of the different tradeoffs they offer.

These criteria will be used to compare the database encryption approaches:

- Target platform:
  If the approach has already been implemented, then the platform is the one it was implemented on, otherwise it is the platform discussed in the papers where the approach was presented.

- Trust model:
  Which parties need to be trusted for the scheme to remain secure and whether the encryption occurs on the server side or the client side.

- Completeness of encryption:
  Whether all data stored is being fully encrypted or if some parts are being left intentionally unencrypted.

- Patterns across ciphertexts:
  Whether any details about the patterns and frequency of the plain text data leaks through the ciphertexts.

- Performance:
  The actual performance characteristics of the approaches are difficult to directly compare as some of them are only published research results without implementations or implementations not available to the public, while others are available and implemented but prohibit the publication of any benchmark data in their license agreements. As such, this will be a theoretical evaluation of their performance characteristics based on the process used for the encryption.

- Functionality (and transparency to the developer):
  How much of the database functionality is still available (or which features will no longer be available) after encryption and how this impacts developers using the database.

- Cloud Compatibility:
  Whether the approach requires the software running on the server side to be modified and if any components that are not traditionally part of cloud architectures are necessary.

The results of the comparison are as follows:

| | Page-Level Encryption | CryptDB | Trusted Hardware | Buckets |
|---|---|---|---|---|
| **Target Platform** | Relational Databases | Relational Databases | Relational Databases | Relational Databases |
| **Trust Model** | Data encrypted on the server side.<br><br>Database operator must be trustworthy. | Data encrypted on the client side.<br><br>No third parties need to be trusted. | Data encryption occurs on both the client and the server side.<br><br>The trusted hardware components need be resistant to tampering by the database operator. | Data encrypted on the client side.<br><br>No third parties need to be trusted. |
| **Completeness of Encryption** | Complete | Partial<br><br>OPE leaks order relations. | Complete | Partial<br><br>Identifying part of the buckets is left unencrypted. |
| **Patterns Across Ciphertexts** | No patterns leak through. | Some patterns leak through due to OPE. (configurable) | No patterns leak through. | Foreign key relations leak through. |
| **Performance** | Best possible | Significant network overheads and optionally slow encryption primitives. | Depends on the performance of custom-made hardware. | Network overheads and query post-processing on client side. |
| **Functionality** | Encryption does not impact functionality. | Supports 99.5% of all queries used across a wide range of tested software. | Encryption does not impact functionality. | Supports point and range queries, as well as joins and aggregations.<br><br>Does not support data being modified by server-side operations. |
| **Cloud Compatibility** | Fully compatible<br><br>However requires the database to support encryption out of the box. | Partially compatible<br><br>Requires a trusted party to maintain a server for encryption. | Partially compatible<br><br>Requires the database to be integrated with the trusted hardware. | Fully compatible<br><br>Some modifications to optimize the execution of queries over the buckets can be applied on the server side. |

**Table 1:** Comparison of database encryption approaches.

As Table 1 shows, there exists a wide variety of options for encrypting relational databases, though none of them are truly ideal, as they all require users to accept some tradeoff. The main issue is that functionality comes either at the cost of security (Page-Level Encryption), performance (CryptDB and Buckets) or cloud compatibility (Trusted Hardware).

# 3 Cassandra

Cassandra's official documentation can be found in [1] and the documentation of the Cassandra Query Language (CQL) in [12]. Most of the specifics of the database presented in this section are based on those manuals.

However Cassandra is still under active development and many aspects of it are only informally documented in internal developer resources like the issue tracker[5] and the mailing lists[6]. As such, many of the specifics of the database are subject to change between versions. The details presented in this thesis are based on Cassandra version 2.1 with CQL version 3.1.

## 3.1 Introduction to Cassandra

Cassandra is an extensible record store, as classified by [10]. It does have a straight-forward (internal) data model, however the situation is complicated by the fact that it tries to present a simplified view of that (internal) data model to its users (i.e. application developers), resulting in it effectively having two different data models, that interact in somewhat non-obvious ways. The data model presented to the users will be referred to as the "CQL data model" here. Note though that the internal data model is still fully accessible to users through the Thrift API, which has been deprecated but is still being maintained for compatibility reasons.

For this thesis only the CQL data model is relevant. The internal data model is briefly mentioned nevertheless, as it is the reason for the various query capabilities that Cassandra can or cannot offer.

### 3.1.1 Internal Data Model

Cassandra's internal data model consists of tables (a.k.a. "column families") in schemas (a.k.a. "keyspaces"). Tables contain tuples (a.k.a. "rows") which in turn contain attributes (a.k.a. "columns"). Generally tuples may contain an arbitrary amount of attributes, independent of the number of attributes in the other tuples of the same table in the internal model.

Attributes are only blob-values in the internal model and the attributes of one row are always stored sorted by the name of the attribute, when written to the internal storage (SSTables [11]).

Every tuple also contains one special attribute – the primary key – which is used for distributing and replicating the tuples across the nodes of the database cluster, in addition to its traditional use for indexing.

### 3.1.2 CQL Data model

Starting with version 1.2 and CQL 3 Cassandra presents a different data model to its users than the one it uses internally. This was done since Cassandra, unlike other NoSQL databases, introduced an SQL-like query language (called the Cassandra Query Language – CQL) to offer its users an easy interface to interact with the database, because the Thrift API, that was being used prior to CQL, was found to be too cumbersome. SQL however, being a language designed primarily with the focus of operating on sets of tuples and not so much sets of attributes, was not a great fit for Cassandra's internal data model and several paradigm-breaking changes were necessary in order for CQL to be useful. Ultimately, with CQL 3 in Cassandra 1.2, the decision was made to map the internal data model into a more traditionally relational one for the users, instead of drifting away further from SQL in the design of CQL.

The internal model was still kept after CQL 3 though, since it is a major part of Cassandra's performance characteristics and its competitive advantage. The goal of the CQL data model was to

[5] https://issues.apache.org/jira/browse/cassandra

[6] http://planetcassandra.org/apache-cassandra-mailing-lists/

allow users that already had prior experience with traditional relational databases to be able to efficiently use Cassandra's data model without any additional effort.

Sadly, the specifics of how exactly the internal model is mapped into the CQL one, as well as the motives for doing so, are not published, and the details are spread across various threads in Cassandra's issue tracker. Nevertheless, a series of informal blog posts [16, 29] has been published, outlining the general details.

While the taxonomy of keyspaces, column families, tuples, and attributes was kept for the CQL data model, the major difference between the internal model and the CQL one is that the attributes of one internal tuple are transposed into multiple different tuples in the CQL model. Additionally, while tuples of the same column family can have an arbitrary number of columns in the internal data model, they have a fixed schema in the CQL data model. This allows for CQL to be much closer to SQL, since the querying criteria can all be written within a where clause that is identical to SQL – at least on a syntactic level at first sight – as well as allowing for a straight forward projection of the result set using specific column names in the SELECT-part of the query.

Further differences between the internal and the CQL data model are that columns have fixed types in the CQL data model, i.e. they are no longer just blob values, and the primary key can be a compound key, i.e. consist of multiple columns. Additionally, the primary key is further divided into two different parts: the partitioning primary key and the clustering primary key, both of which can consist of an arbitrary amount of actual columns. The partitioning primary key is the one that will be used as the primary key in the internal data model, and thus to shard and replicate the tuples, while the clustering primary key will be used when transposing CQL tuples into internal tuples to define their ordering, i.e. multiple CQL tuples with a common partitioning primary key will be sorted in one internal tuple based on their clustering primary key, when written to the internal storage.

The clustering primary key may be left empty, i.e. consist of no columns, though all subkeys, i.e. any individual column that is part of the primary key, must not be null.

Additionally to this, Cassandra offers also 4 non-scalar data types with the CQL model: maps, sets, lists and documents. Maps, sets and lists are broken down into multiple individual attributes in the internal model for performance reasons, whereas documents are only desugared into one single attribute currently (and thus operations on them have read-before-write semantics).

A column family "email" in a keyspace "mailsystem", containing the emails of the system's users, and using the email address as the partitioning primary key and the message ID as the clustering primary key, would look as shown in Figure 2 in the internal data model, and as shown in Figure 3 in the CQL data model.

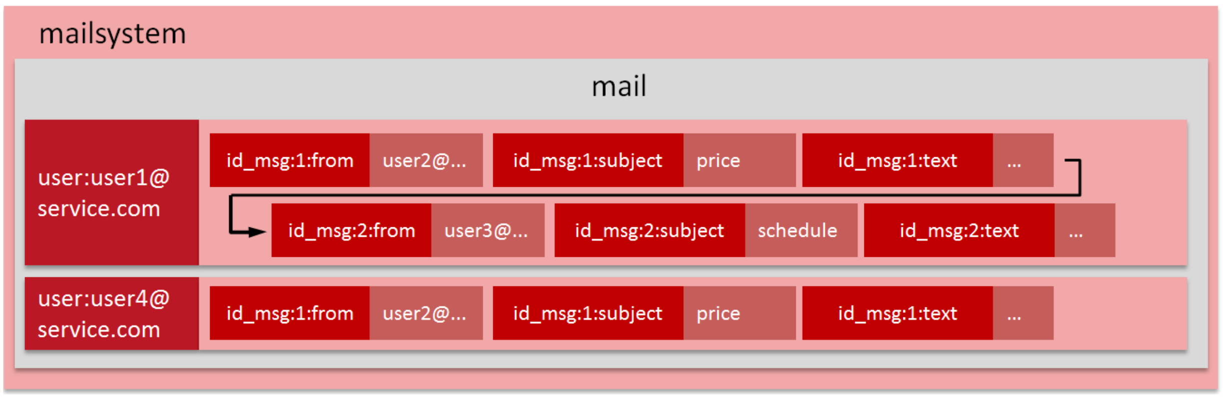


**Figure 2:** Example of a column family in the internal model.

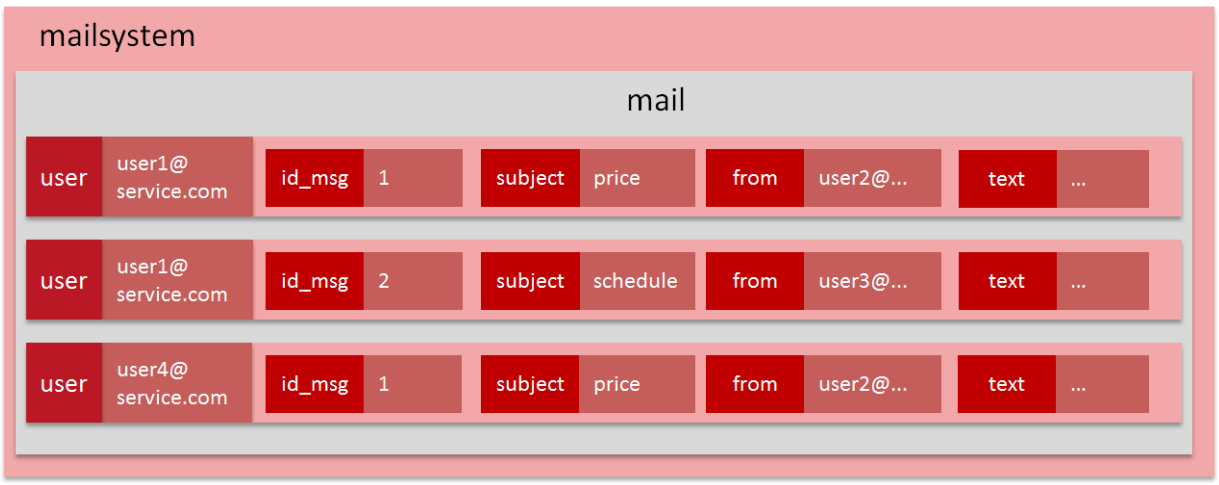


**Figure 3**: Example of the column family from Figure 2 in the CQL data model.

### 3.1.3 Introduction to CQL

Though CQL does look very similar to SQL at first sight, Cassandra being an extensible record store does mean that it only allows for much more limited queries to be executed on it. This subsection presents the queries Cassandra allows for, in order to later discuss how much of the functionality can be maintained, despite the data being encrypted on the client side.

Cassandra does provide a CQL reference to its users in [12], however that document is not complete and many specifics on CQL are spread across the issue tracker and the mailing lists, especially since sometimes it is unclear if certain behaviors are intentional or not. As such, the details presented in this section are based on Cassandra 2.1 and CQL 3.1 specifically.

#### *3.1.3.1 Inserts, Updates, Deletes, Lightweight Transactions and Batches*

Inserts, updates and deletes are indistinguishable from one another to Cassandra, despite all three statements being valid syntax. An insert is an update with a primary-key value in the where clause which does not yet exist in the database and a delete is an update that sets all fields to null for that specific tuple. The consequence of this is that once all attributes of a tuple have been set to null, the tuple itself is considered as deleted too.

Internally null values are either not actually stored in the database, when inserting a new row, or a tombstone is written instead, in the case of delete statements or whenever setting an attribute to null causes its prior value to be deleted. Tombstones are kept for a duration of ten days (by default) and are automatically deleted afterwards. They are needed by Cassandra in order to communicate the deletion of values to database nodes that did not participate in the original execution of the query that deleted the values.

A deviation from delete Statements in SQL is that CQL also allows for deleting individual columns – i.e. a statement of the form

```
DELETE col FROM tbl WHERE key=1;
```

is valid CQL and has the semantics of

```
UPDATE tbl SET col=null WHERE key=1;
```

Additionally to this, Cassandra also allows the deletion of individual collection elements for maps, using a statement of the form

```
DELETE col['map-key'] FROM tbl WHERE key=1;
```

Starting with version 2.0 Cassandra also introduced support for so-called “Lightweight Transactions” (LWT) [17], which allow for linearizing operations on individual tuples (only for I/U/D queries). Note that the name is somewhat misleading here, as these are not actual ACID-like transactions [24] but rather an implementation of Paxos-based linearizability [28] over attributes of a single tuple only [37, 27].
At the CQL level LWTs can be used by providing an additional IF clause to the query:

```
UPDATE tbl SET col=1 WHERE pk=1 IF col=2;
```

The semantics of this query are that “col” is to be set to 1, if its current value is 2, across all nodes responsible for maintaining this tuple. If the value is not 2, e.g. because another query being executed on another node already modified the value prior to this statement taking effect, then the query will have no effects on the database and return an error to the user.

Cassandra also allows for batching multiple individual I/U/D statements together into one batch statement, e.g.:

```
BEGIN BATCH
       UPDATE tbl SET col=val 1 WHERE pk=1
       UPDATE tbl SET col=val 2 WHERE pk=2
       UPDATE tbl SET col=val 3 WHERE pk=3
       ...
APPLY BATCH;
```

The individual statements within the batch statement will all be executed independently of one another. However batches are “atomic” by default, which means that Cassandra guarantees that any statements that fail will be re-executed until all the statements within the batch statement have been successfully completed [15]. Non-atomic batch statements can be used by starting the batch statement with BEGIN UNLOGGED BATCH instead.

I/U/D statements all face the same limitations (with minor differences):

- The where clause needs to contain the full primary key, unless it is a delete, in which case only the partitioning primary key needs to be provided. The effect of this is that multiple tuples of a column family can be deleted by using a less restrictive WHERE clause, but not updated. Nevertheless, if specific collection elements are being deleted then the full primary key must be provided for deletes too. This is due to how collections are represented in the internal data model.

- The where clause must not contain any non-primary key columns (i.e. secondary indices can only be used for selects).

- Primary key columns and counter columns cannot be used in the IF clause of a Ligtweight Transaction.

- No IN() conditions may be used in the where clause on clustering primary keys. This is an implementation artifact of Cassandra, i.e. not necessitated by the data model itself, which may be lifted in future versions.

- No IN() conditions may be used for any part of the primary key if the affected tuples are participating in a Lightweight Transaction.

- Delete statements for individual collection elements are only allowed for maps. Set and List elements can still be individually deleted, but only through the appropriate collection operators in update statements.

### *3.1.3.2 Selects*

The limitations that Cassandra imposes on select queries may seem arbitrary at first, but they stem from the underlying internal data model and the design goal of allowing only efficiently executable queries to also be valid ones.

Generally, tuples can only be queried for by their primary key. Every part of the partitioning primary key must be present in the query. The clustering primary key may be fully or partially omitted. However, all leading attributes of the clustering primary key must be provided, if any part of it is used in the query. Additionally, the last parts of both the partitioning as well as the clustering primary key can be queried using an IN() condition. Furthermore, the last occurring part of the clustering primary key (in the specific query) may be queried for with an inequality condition (i.e.  >, >=, <, <= but not !=).

To illustrate these limitations the following sample table will be used:

```
CREATE COLUMNFAMILY test (
       ppk1 int, ppk2 int, ppk3 int,
       cpk1 int, cpk2 int, cpk3 int,
       nk int,
       PRIMARY KEY((ppk1, ppk2, ppk3), cpk1, cpk2, cpk3)
);
```

Columns ppk1, ppk2 and ppk3 are part of the partitioning primary key. Columns cpk1, cpk2 and cpk3 are part of the clustering primary key.

These sample queries are valid ones:

```
SELECT * FROM test WHERE ppk1=1 AND ppk2=1 AND ppk3 IN (1,2);
SELECT * FROM test WHERE ppk1=1 AND ppk2=1 AND ppk3 IN (1,2) AND cpk1>=1;
SELECT * FROM test WHERE ppk1=1 AND ppk2=1 AND ppk3 IN (1,2) AND cpk1=1
                                    AND cpk2>=1;
SELECT * FROM test WHERE ppk1=1 AND ppk2=1 AND ppk3 IN (1,2) AND cpk1=1
                                    AND cpk2=1 AND cpk3 IN (1,2);
```

These sample queries on the other hand are not:

```
SELECT * FROM test WHERE ppk1=1;
```
(A part of the partitioning primary key is missing.)

```
SELECT * FROM test WHERE ppk1 IN (1, 2) AND ppk2=1 AND ppk3=2;
```
(IN(..) may only be used for the last key fragments.)

```
SELECT * FROM test WHERE ppk1=1 AND ppk2=1 AND ppk3=2 AND cpk1 IN (1,2);
```
(IN(..) may only be used for the last key fragments.)

```
SELECT * FROM test WHERE ppk1=1 AND ppk2=1 AND ppk3=2 AND cpk1>=1 AND cpk2=1;
```
(Inequality conditions may only be used on the last clustering key fragment.)

```
SELECT * FROM test WHERE ppk1=1 AND ppk2=1 AND ppk3>=2;
```
(Inequality conditions may only be used on the last clustering key fragment.)

Using secondary indices allows an indexed column to be used instead of the primary key in the where clause, in which case no further columns may be contained in it. Multiple indexed columns cannot be used in the where clause of the same query, since Cassandra implements no mechanism for merging the results of multiple secondary index lookups.

Additionally Cassandra also allows querying for only distinct attribute values (i.e. SELECT DISTINCT .. FROM ..), in which case only columns of the partitioning primary key may be contained in the select part of the query. All other constraints regarding the where clause of select queries remain unaffected in this case.

For range queries Cassandra also provides a mode with more lenient restrictions by using the ALLOW FILTERING option, which searches all tuples on all nodes for the ones matching the queried-for conditions, and thus does not require the presence of any partitioning primary key fragments in the where clause. This is mainly a convenience feature for application developers, which allows for easier debugging.

#### *3.1.3.3 Counters*

Counters are a feature of Cassandra that allows for distributed counting.

At the CQL level they exist as a data type (much like "int"). Counters are the only data type that may be incremented and decremented in update queries in Cassandra – though they can only be incremented and decremented by static amounts, but not the value of any other columns. E.g. this query to update a counter column is a valid one, since it increments the counter using only a static value:

```
UPDATE tbl SET col=col+1 WHERE pk=1;
```

This query on the other hand is not valid, since the counter is being incremented using the value of another column:

```
UPDATE tbl SET col_1=col_1+col_2 WHERE pk=1;
```

Additionally tables containing counters may only contain counter columns, aside from the primary key columns. After their creation, at which point they are initialized with 0, counters can only be modified by increment and decrement operations, i.e. they cannot be set to some specific value. Both of these are inherent limitations of the way Cassandra's counters are implemented.

More specifically, Cassandra uses a Conflict-Free Replicated Data Type (CRDT, [39]) to store individual counter updates internally, and calculates the sum of all stored updates whenever the user queries the value of the counter. The actual implementation in Cassandra is more involved, since various optimization attempts are made, e.g. trying to converge the values per node. Nevertheless, the overall limitations still remain, since:

— Cassandra needs to store multiple rows per counter internally, which precludes the use of any further columns that are not a CRDT of the same kind.

— Resetting or setting the counter to some specific value would require all currently stored counter updates across all nodes to be deleted and the new counter value to be inserted, all in one atomic step, in order to avoid any interference by any other ongoing counter updates of other clients.

### *3.1.3.4 Static Columns*

Cassandra offers the option to define a column of a column family (in CQL) as being only dependent on the partitioning primary key. The effect of this is that the column is shared across all tuples of the same partition. This kind of column is called a static column, and can be used by adding the keyword STATIC to the column definition, after the data type, in the CREATE statement. E.g. in the table created by the following statement the col_static column will be a static one:

```
CREATE COLUMNFAMILY tbl (
        ppk int,
        cpk int,
        col_static int STATIC,
        col_regular int,
        PRIMARY KEY(ppk, cpk)
);
```

Examples of what static columns can be used for are one-to-many relationships as well as aggregates of the data stored within one partition.

### *3.1.3.5 Server-Side Functions*

A variety of functions that are evaluated on the server-side is provided to database users – e.g. functions to get the current date, generate random values or convert data types.

Additionally Cassandra also offers a COUNT() function, however only in the form of COUNT(*) which returns a count of the rows that would have been in the result set. No other arguments are valid for the function. Also other traditional aggregate functions like MIN() or SUM() do not exist.

Finally there are also two functions that operate on internal metadata only:

- WRITETIME() returns when the attribute was last written.
- TTL() sets when the value of the attribute is to be automatically deleted. This is a feature offered by Cassandra and can be used to e.g. automatically invalidate cached values that are being stored in a column family.

### *3.1.3.6 Secondary indices*

Cassandra's secondary indices serve a somewhat different purpose than secondary indices in traditional relational databases and have vastly different performance characteristics.

In Cassandra the data contained in one table is spread across the individual nodes of the database cluster, in order to improve the database's performance. This partitioning of the data occurs based on the primary key (actually only the partitioning primary key, see subsections 3.1.1 and 3.1.2 on the details of the data model), so queries that use the primary key in the where clause can be immediately delegated to the nodes responsible for the queried-for data partitions, and thus avoid tying up unnecessary resources. This is one reason why Cassandra generally requires the primary key to be provided in select queries.

Secondary indices allow for omitting the primary key from a select query, and querying an indexed column instead. The index data is always stored locally on the same node that also contains the primary data. The consequence of this is that queries using secondary indices have to run on every node of the database cluster, since it is not clear beforehand which nodes actually contain the queried-for data to delegate the query to them, unlike queries that use primary keys.

From a performance perspective this has two implications:

- Querying high-cardinality indexed columns will result in poor performance, since the entire database cluster will be tied up to process the query, while relevant (i.e. queried-for) data will only be present on a few nodes.

- Querying very low-cardinality columns will result in poor performance as the individual nodes will have to perform many disk seeks to read all relevant (i.e. queried-for) tuples from storage.

Ideally, the cardinality of an indexed column should be such that every node will contain at least a few relevant tuples (so that processing the query on all of them will not waste resources) – but not too many, so as to be able to retrieve all of them in as few disk seeks per node as possible. Further details on this, as well as the motivation for why Cassandra's secondary indices have been designed the way they are, can be found in [36].

Considering these details, it can be difficult to determine when and when not to use secondary indices for their performance in practice, and thus the general advice is to "manually" index the data (i.e. create additional tables that serve as lookup lists), if performance is critical.

Nevertheless, secondary indices can still serve as a convenience feature for queries that are not often executed, as they are the only way of querying a table using non-primary key columns. (See Subsection 3.1.3.2 for the details on the actual query limitations.)

Secondary indices can be created and deleted through DDL statements. To create an index on the test table of Subsection 3.1.3.2 the following statement can be used:

```
CREATE INDEX idx_test_nk ON test(nk);
```

Note that the index name in the statement ("idx_test_nk") is optional. Cassandra will assign a name of its own if no one was provided by the user. One column family can have arbitrarily many indexed columns, though one column may only be indexed once, and only one indexed column may be used in any one select query.

#### *3.1.3.7 Privilege Management*

Cassandra offers the option to define users and set their privileges (e.g. create, drop and modify) for keyspaces and column families.

However it has no mechanism of its own to authenticate users and thus requires a third-party service for authentications.

## 3.2 Encryption for Cassandra

A detailed description of extensible record stores can be found in [10]. Subsection 3.2.1 will focus on their data model at a high level, in order to explain how the encryption scheme outlined in Subsection 1.4 can be applied to them. Subsection 3.2.2 will then present the specific details necessary to encrypt Cassandra databases.

### 3.2.1 Encryption for Extensible Record Stores

Extensible Record Stores, much like traditional relational databases, consist of tables containing tuples which in turn contain attributes. The tuples are indexed by a primary key. Depending on the actual implementation, that primary key may be a compound key (i.e. consist of multiple attributes) and there may exist the option to also index further attributes. Depending on the actual implementation, the tables containing the tuples may further be nested in schemas and the attributes themselves may be of a non-scalar type.

Tuples will generally be partitioned (and replicated) across the nodes of the database cluster depending on their primary key (or a subset of the attributes comprising the primary key) and a major reason for using these kind of databases is the performance and fault-tolerance they can deliver, under the assumption that most data operations will be performed based on a primary key, so that the operations can immediately be delegated to the affected nodes, rather than tie up cluster-wide resources.

The idea for the encryption scheme presented in this thesis is to encrypt every data value using all identifying data values that are necessary to reach its current nesting level in the data model, in addition to some user-specific base key. E.g. an attribute value within a column, within a tuple, within a table, within a schema will be encrypted based on the user key, the schema name, the table name, the column name, and finally the tuple's primary key value.

This ensures that every encrypted piece of data stored in the database will have its very own encryption key, that is unique for that one user, based on the user key, since duplicates of every value used to construct the encryption key would be necessary in order for one encryption key to be a duplicate of another, which is precluded by the data model.

If the database allows for non-scalar attribute types then the scheme can be generalized further, in order to be applicable to them too:

- For map-like types the map-key can further be added after the primary key in order to form the encryption key for the map-value, while the map-keys themselves are treated as regular attribute values.

- Document-like types can be treated as nested maps.

- Set-like types (that disallow duplicates) need no special treatment, since they preclude the possibility of duplicate plain text values and hence there will be no patterns that could leak in the first place.

The only real issue that remains are list-like types (i.e. sets that allow for duplicates), since they may contain duplicate values that are not indexed by any unique values themselves. There exists no ideal solution for their encryption, and if the data in the list needs to be protected against identical list elements being identifiable after the encryption, then additional (unique) data needs to be added to the list elements prior to the encryption. (The data needs to be added to the list elements and cannot be added to the encryption key, since otherwise a decryption of the values would no longer be possible.)

Two alternatives exist here:

- Adding random data.
  This allows for the operations on the individual list elements to still be able to be delegated to the database, but precludes the use of the attribute in a primary key or from being indexable in general, as well as being individually deletable (i.e. without reading the full collection first and then writing it back without the element to be deleted). (Note though that non-scalar attributes being part of a primary key may not be possible in the first place, depending on the actual database implementation.)

Additionally this also may reduce the security of the encryption by allowing for side-channel attacks on the random number generator (RNG) [40]. The feasibility of such attacks however is beyond the scope of this thesis and may be a non-issue if the RNG is not vulnerable to them.

- Adding the position of the element in the list to the list element data.
  This would allow for the attribute to be used as part of a primary key (if the database supports the feature), though it would still require for the user to provide the list elements all in the correct order for the lookup query. Furthermore it also still practically precludes the attribute from being generally indexable, since lookup queries would need to query for all the possible places that the element could occur in the list.
  This also precludes the possibility of the database performing operations on the list elements on the server side, as insertions and deletions of elements into and from the list would inevitably change the position of the other elements on the list. Thus the encryption layer would need to first read all elements of the list, then perform the operation, and then write all of them back again.

### 3.2.2 Application of the Encryption Scheme

This is a detailed description of how the general encryption scheme outlined in Subsection 3.2.1 is applied to Cassandra specifically, based on the CQL data model. Note that the actual query rewriting details are in Subsection 3.2.3 and the implementation details in Section 4.

As discussed in Subsection 2.2.1, encryption keys and initialization vectors (IVs) have a similar effect on the resulting ciphertexts. As such, it would be suitable to incorporate the data access path into either one of them, for the goal of randomizing the ciphertexts as discussed in Subsection 3.2.1. Nevertheless, for this specific scenario the IV is the better choice, since for IVs (unlike for keys) no care needs to be taken to ensure that they are uniformly distributed. This allows EncryptedCassandra to construct a matching IV, which is to be used in the encryption, by simply applying a fast hash function, which merely satisfies the block cipher's size requirements, on the final value of the constructed data access path.

The actual data access paths constructed by EncryptedCassandra are as follows:

- Table (column family) names will be encrypted using the
  base IV
  + the schema (keyspace) name.

- Attribute names will be encrypted using the
  base IV
  + the schema (keyspace)
  + the table (column family) name

— Primary key attribute values will be encrypted using the
base IV
+ the schema (keyspace)
+ the table (column family) name
+ the name of the primary key attribute

If the primary key is a compound key, then the column name of the specific primary key sub-key's attribute will be used in that attribute's encryption. In addition to this all values of all preceding primary key sub-key attributes will also be added to the encryption key. This is done in order to avoid identical primary key fragments having identical ciphertexts.

— Non-primary-key attribute values will be encrypted using the
base IV
+ the schema (keyspace)
+ the table (column family) name
+ the column name of the attribute
+ the primary key value of the tuple (row)

Using the example given in Figure 3 of Subsection 3.1.2, Figure 4 shows which parts of the data model are used in the construction of the IV for the encryption of the individual pieces of data. An arrow indicates that the value at the arrow's shaft will be used in the encryption key of the value at the arrow's head.

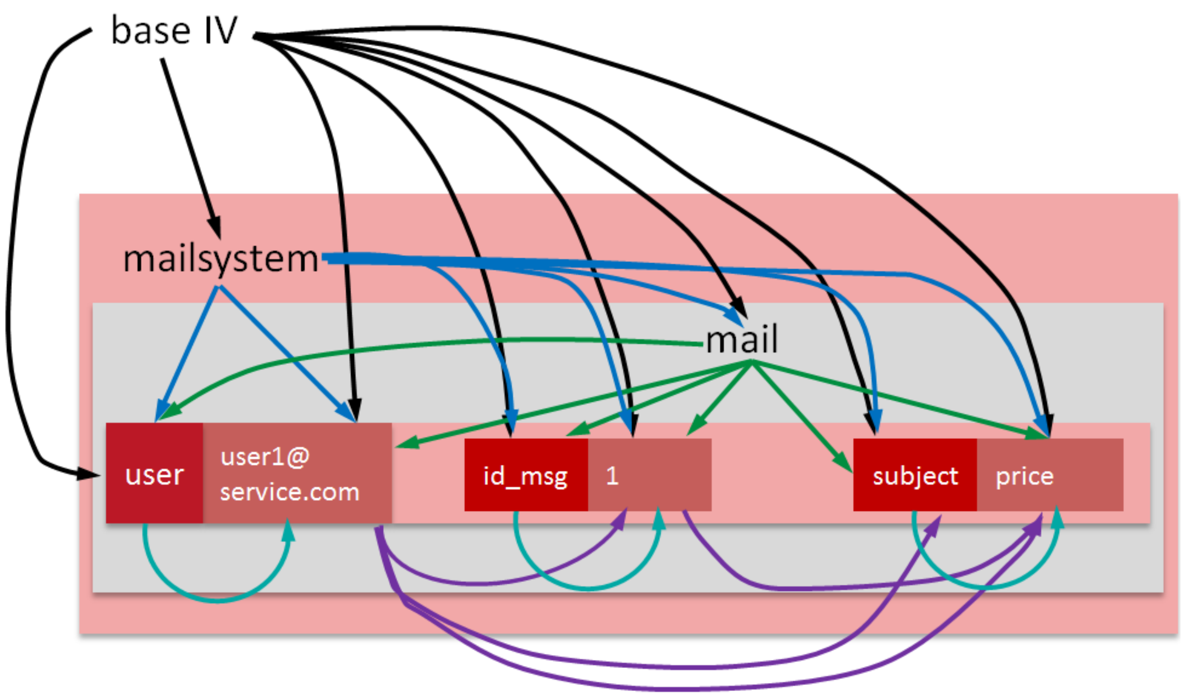


**Figure 4:** The fragments used for creating the encryption keys for the individual pieces of data.

Collections are processed based on the actual type of collection:

- For list elements additional random bytes are added to the elements. The amount of bytes can be set in the DDL statement when creating the table. E.g. in the following table 2 bytes of random data will be used in the elements of the col_lst lists:

  ```
  CREATE COLUMNFAMILY tbl (
          pk int,
          col_lst list<int> RAND(2),
          PRIMARY KEY(pk)
  );
  ```

  The statement would still be perfectly valid without the code fragment in orange, in which case no random bytes are used for the encrypted list elements.

- For maps the map key is treated as the attribute value and the map value is encrypted the map key, in addition to the data access path that was used for the map key itself.

- For set elements no further action is taken and every set element is treated like an ordinary attribute value.

Document types on the other hand, while straight-forward at the conceptual level, are more complex since they are still under development[7] in Cassandra currently. Thus it is unclear how to best support them, and as such they are not implemented in EncryptedCassandra.

Generally two options exist to support document types in EncryptedCassandra, neither of which are ideal however, given Cassandra's current state of the feature:

- Manual desugaring: since document types currently are only CQL-level syntactic sugar, it would be possible to manually desugar them into blob values. However this would require for them to be fully re-implemented inside the encryption layer in order to do so (and maintain API compatibility to the user), and would preclude the implementation of benefiting from any performance improvements if/when document types receive direct server-side support.

- Direct support: by encrypting the individual values inside the document. This would avoid the issues of manual desugarring, however it is complicated by the fact that document type attributes can be part of the primary key, while still being able to change some of their structure over time, as well as contain list-like elements which would require the addition of random data into them to be securely encrypted, which would effectively result in tuples being not queryable (due to the primary key containing random data after the encryption).

If/when document types receive full database support by Cassandra (i.e. stop being just syntactic sugar), Cassandra's developers are likely to face similar issues and impose certain restrictions onto

[7] See these issue tracker entries for the current changes planned for document types:
https://issues.apache.org/jira/browse/CASSANDRA-7423 - full database support instead of just desugarring documents into blobs;
https://issues.apache.org/jira/browse/CASSANDRA-7826 - specifics of the interactions with collections;
https://issues.apache.org/jira/browse/CASSANDRA-6382 - indexing of document types in general;
https://issues.apache.org/jira/browse/CASSANDRA-7771 - indexing of sub-documents;
https://issues.apache.org/jira/browse/CASSANDRA-7791 - namespace/scope of document types across keyspaces.

them to alleviate these issues. This in turn will allow for a more straight-forward way of supporting their encryption in EncryptedCassandra.

For counters a Positive/Negative Conflict-Free Replicated Data Type (PN-CRDT, [39]) is implemented atop conventional CQL tables. It stores the individual encrypted increments/decrements that the counter has been manipulated by, and adds the individual entries together whenever the user queries the counter's value. This is similar to Cassandra's internal implementation of counters. Further details are provided in Subsection 4.2.4.

Unless otherwise specified, all values are encrypted using a symmetric block cipher. The exception to this are null values, which always remain unencrypted as they signify the deletion of values in Cassandra. In order to encrypt null values properly, manual management of tombstones would be necessary, which is not practical to implement at the client-level of a database driver.

In addition to symmetric block ciphers, attribute values of all data types can alternatively be encrypted by an order preserving algorithm or left entirely unencrypted. The exception to this are collections, which cannot use OPE.

When using order preserving encryption (OPE), the primary key fragments of the tuple are not added to the encryption key, in order to allow the value to be indexed by Cassandra's secondary indices. If indexing of the values is not necessary, then a so called Cluster-OPE mode can be used instead, which does still include all preceding primary key fragments into the encryption key, while also using OPE. The purpose of Cluster-OPE is to allow the use of inequality conditions on encrypted clustering primary key columns in the WHERE clause of select queries.

Whether a column is encrypted by a symmetric block cipher (default), OPE, Cluster-OPE, or left unencrypted is set in the DDL statement when creating the column family (see Subsection 3.2.3.1 for the details on the syntax).

### 3.2.3 Query Rewriting in Detail

CQL has grown substantially in version 3 and is now a language with many specific details. As such, instead of presenting the entirety of the individual grammar rules and how they affect the query rewriting, this subsection will instead provide a general description of the process and focus on the details only in non-obvious cases or for CQL features that are not supported by the rewriting process. Additionally examples are provided to illustrate the effects of the query rewriting.

#### *3.2.3.1 DDL*

In DDL statements only the plain text names of the individual database objects (like tables and columns) are replaced by their ciphertexts and column types are changed to either blob (default) or varint[8] (for OPE columns). For unencrypted columns the column data type remains unchanged. For counter columns the data type is long (see Subsection 4.2.4 for the specifics of the counter implementation).

Since the data types of the columns are different after the encryption, EncryptedCassandra needs to maintain a data dictionary of its own, in order to keep track of the original data types, to perform type checks as well as convert the decrypted data back into the appropriate type, prior to returning it to the user. Necessary data dictionary operations are always performed implicitly. This includes setting up the data dictionary tables when creating a new keyspace (schema), creating new data dictionary entries when new database objects are added to the schema and deleting them when objects are removed. Since these operations are performed transparently to the database user,

[8] Arbitrary-precision integer

executing one unencrypted DDL statement by the user will in fact result in multiple encrypted statements being issued to Cassandra.

EncryptedCassandra however also extends CQL's syntax for creating columns – it allows users to (optionally) specify the encryption details of the column by adding the keywords OPE, CLUSTER-OPE or PLAIN after the column type, to have the column be encrypted by the OPE scheme or remain fully unencrypted. The differences between OPE and Cluster-OPE were discussed in Subsection 3.2.2.

Additionally, EncryptedCassandra also allows users to specify the amount of random bytes to be used for list elements in their encryption, by using the RAND keyword and providing the number of bytes in parentheses (an example of this was shown in Subsection 3.2.2).

When specifying columns to be OPE-encrypted, the domain and range to be used for the encryption can optionally be provided in parentheses. This CQL DDL statement creates a table where the last_name column uses OPE with a domain of 20 bytes and a range of 60 bytes, while the first_name column remains unencrypted:

```
CREATE COLUMNFAMILY emp (
  empID int,
  deptID int,
  first_name text PLAIN,
  last_name  text OPE(20,60),
  PRIMARY KEY (empID, deptID)
);
```

The syntax parts in purple, black and blue are default CQL syntax. The parts in orange are extensions by EncryptedCassandra. The statement would still be perfectly valid if any of the orange parts were to be omitted.

After the encryption the statement would be rewritten to this:

```
CREATE COLUMNFAMILY "ec_4dUsSG"."ec_Lw41GVe" (
  "ec_88X3B7PMoKugS8QqaefyiVFiL" blob,
  "ec_7WrxWstKgq" blob,
  "ec_VaoF9dkHxhu" blob,
  first_name text,
  "ec_hPvZZPPeJvhPcYY" varint,
  PRIMARY KEY ("ec_7WrxWstKgq", "ec_VaoF9dkHxhu")
);
```

The first_name column, which was specified to remain unencrypted, has not been changed while all other columns have had their name and data type rewritten. The column names have been encrypted and the resulting ciphertexts have been encoded such that they can be used as identifiers within Cassandra. Additionally to being encrypted, the table name has also been prefixed by the (encrypted) keyspace name.

Note also how the encrypted table has one more column – this is due to last_name being OPE encrypted which requires the column to be duplicated internally. The column is duplicated in order to also store an encrypted value of last_name that does not use OPE. This allows to avoid the need to decrypt the OPE ciphertext later on, if the user selects it in a query, which is very computationally intensive, and thus improve performance.  Further details on this are provided in Subsection 4.2.2.

To delete the created table a drop statement can be used:

```
DROP COLUMNFAMILY emp;
```

After encryption the statement will be rewritten to this:

```
DROP COLUMNFAMILY "ec_4dUsSG"."ec_Lw41GVe";
```

Additionally to these create and drop statements, various insert and delete statements to populate and clean up the data dictionary are created during the encryption. When creating the aforementioned emp table, these insert statements (one per column in the emp table) will be used to populate the data dictionary:

```
INSERT INTO ec_catalog_col (cf, col, col_order, col_role,
       col_type_original, col_type_encrypted,
       ope_type, ope_domain_size, ope_range_size,
       col_encd, col_rand, col_index, index_name)
VALUES ('emp', 'empid', 1, 'partition',
       'int', 'blob',
       'none', 0, 0,
       1, 0, 'none', '');

INSERT INTO ec_catalog_col (cf, col, col_order, col_role,
       col_type_original, col_type_encrypted,
       ope_type, ope_domain_size, ope_range_size,
       col_encd, col_rand, col_index, index_name)
VALUES ('emp', 'deptid', 2, 'cluster',
       'int', 'blob',
       'none', 0, 0,
       1, 0, 'none', '');


INSERT INTO ec_catalog_col (cf, col, col_order, col_role,
       col_type_original, col_type_encrypted,
       ope_type, ope_domain_size, ope_range_size,
       col_encd, col_rand, col_index, index_name)
VALUES ('emp', 'first_name', 3, 'none',
       'text', 'text',
       'none', 0, 0,
       0, 0, 'none', '');

INSERT INTO ec_catalog_col (cf, col, col_order, col_role,
       col_type_original, col_type_encrypted,
       ope_type, ope_domain_size, ope_range_size,
       col_encd, col_rand, col_index, index_name)
VALUES ('emp', 'last_name', 4, 'none',
       'text', 'text',
       'ope', 20, 60,
       1, 0, 'none', '');
```

Since the ec_catalog_col table is encrypted, these insert statements will be encrypted prior to execution too. How inserts are encrypted is shown in Subsection 3.2.3.2. The semantics of the individual columns of the ec_catalog_col table are explained in Subsection 4.2.1.

When deleting (droping) the emp table, the data dictionary will be cleaned up using this delete statement:

```
DELETE FROM ec_catalog_col WHERE cf='emp';
```

Since the ec_catalog_col table encrypted, this delete statement will be encrypted prior to execution too. How deletes are encrypted is shown in Subsection 3.2.3.5.

Another noteworthy case of DDL are secondary indices. EncryptedCassandra does two distinctively different things, based on how the column is being encrypted:

- If it is an OPE-encrypted or an unencrypted column, then only the object names in the statement will be rewritten (and the data dictionary will be updated to reflect the changes). This will result in Cassandra's native secondary index facility to be used for indexing the column.

  As such, the following statement:

  ```
  CREATE INDEX ON emp(last_name);
  ```

  would be rewritten to:

  ```
  CREATE INDEX ON "ec_4dUsSG"."ec_Lw41GVe"("ec_hPvZZPPeJvhPcYY");
  ```

- If it is an encrypted non-OPE column then the CREATE INDEX statement will be completely dropped and EncryptedCassandras's will use a secondary index facility of its own to handle the indexing of the column.

  How EncryptedCassandra's secondary indices are built is shown in Subsection 4.2.3. Additionally to creating the index table itself though, rewriting this query also involves updating the data dictionary, such that it lists the last_name column as an indexed one from now on. This is done by setting the col_index column to ec:

  ```
  UPDATE ec_catalog_col SET col_index='ec'
  WHERE cf='emp' AND col='last_name';
  ```

  EncryptedCassandra also extends CQL here to allow users to indicate if they want operations on indexed columns to be ticketed – this can be done by using the optional keyword TICKET after CREATE – i.e. CREATE TICKET INDEX .. ON ..

  Tickets are an optional feature that can be used in order to completely avoid any derangement of operations on index and data tables. They come at a significant performance overhead though. Subsection 4.2.3 provides further details on this.

#### *3.2.3.2 Inserts*

Inserts are fairly straight forward in that generally only data and identifiers need to be rewritten in the statements.

Noteworthy exceptions to this are:

- OPE columns, since the OPE column need to be duplicated in the statement, in order to avoid having to decrypt OPE values later on and thus improve performance. Subsection 4.2.2 provides for further details on this.

- Indexed and encrypted columns that do not use OPE. For these columns EncryptedCassandra implements a secondary index facility of its own. Thus statements writing to indexed columns require certain index maintenance operations to be performed, in order to keep the index updated and in sync with the primary data. Subsection 4.2.3 provides further details on how EncryptedCassandra maintains its secondary indices.

Any server-side functions that generate or manipulate data (like NOW() or UUID()) and would be valid to use in an insert query can be used, provided that the column they are being used on is an unencrypted one. Using those functions on encrypted columns is not supported, as their functionality would need to be re-implemented on the client side within the encryption layer.

An insert into the emp table of Subsection 3.2.2.1 could be as follows:

```
INSERT INTO emp (empid, deptid, first_name, last_name)
VALUES (1, 2, 'first', 'last');
```

This statement would be rewritten to:

```
INSERT INTO "ec_4dUsSG"."ec_Lw41GVe" ("ec_7WrxWstKgq", "ec_VaoF9dkHxhu", first_name,
       "ec_hPvZZPPeJvhPcYY", "ec_88X3B7PMoKugS8QqaefyiVFiL")
VALUES (0xC97C91042660, 0xA7F519583DB5, 'first', 13216..., 0xF70D992D251E);
```

The OPE-encrypted value for the last name has been truncated here for brevity. Note also that there is one more column in the encrypted query compared to the unencrypted one – this is due to the OPE value being duplicated.

### *3.2.3.3 Selects*

Before rewriting a select statement EncryptedCassandra first checks:

- if all data necessary to perform the encryption of the data in the query statement (i.e. all relevant primary key fragments) are present
- if all data necessary to decrypt the result set is being selected (i.e. if the result set will contain all key fragments, if any encrypted columns are being queried for)

If the first check fails then the query is rejected. In most cases queries that fail the first check would also have been rejected by Cassandra itself too, since the database requires the primary key values to perform the actual lookup of the data. The one exception to this are queries that use the ALLOW FILTERING option (see subsection 3.1.3.2 for details), which allows to omit the partitioning primary key in the query. As such, EncryptedCassandra does not support the use of ALLOW FILTERING.

If the second check fails then EncryptedCassandra adds the columns missing to decrypt the result set to the query itself, and then later on removes them from the result set prior to returning it to the user after decryption. These "missing" columns are primary key columns that the user did not want included in the final result set. As such, EncryptedCassandra determines if any columns are missing by checking if any primary key columns are not present in the select part of the query that the user provided.

Other than that, the actual encryption of identifiers and data in the statement is similar to inserts.

A select from the emp table of Subsection 3.2.3.1 could be as follows:

```
SELECT first_name, last_name
FROM emp
WHERE empid = 1;
```

This statement would be rewritten to:

```
SELECT first_name, "ec_88X3B7PMoKugS8QqaefyiVFiL" as "ec_hPvZZPPeJvhPcYY",
    /* ec: */ "ec_7WrxWstKgq",
    /* ec: */ "ec_VaoF9dkHxhu"
FROM "ec_4dUsSG"."ec_Lw41GVe"
WHERE "ec_7WrxWstKgq" = 0xC97C91042660
```

The reason for more columns being selected in the encrypted statement is that EncryptedCassandra adds the empid and deptid columns (the primary key) to the query, both of which it needs to decrypt the last_name column. It also aliases the selected last_name column to “last_name” since this is an OPE column and thus its non-OPE-encrypted value is stored in a different column internally, in order to avoid having to decrypt the OPE value itself and thus speed up the decryption process (see Subsection 4.2.2 for the details on OPE).

Additionally to this, depending on the where clause, the statement may also need to be split up into multiple statements internally. This is the case if both an IN() operation is used on the partitioning primary key as well as querying for specific clustering primary key values. This is due to the clustering primary key values having different ciphertexts depending on the actual partitioning primary key value.

In the statement

```
SELECT last_name
FROM emp
WHERE empid IN(1, 2) AND deptid = 5;
```

the deptid of 5 will have different ciphertexts depending on whether the empid is actually 1 or 2. Since Cassandra does not allow for any OR conditions in the where clause, two different statements need to be issued in order to query the database for all the rows that could be affected. As such, after encryption the query will be split up into the following two queries:

```
SELECT "ec_88X3B7PMoKugS8QqaefyiVFiL" AS "ec_hPvZZPPeJvhPcYY",
    /* ec: */"ec_7WrxWstKgq" ,
    /* ec: */"ec_VaoF9dkHxhu"
FROM "ec_4dUsSG"."ec_Lw41GVe"
WHERE "ec_7WrxWstKgq" IN (0xC97C91042660) AND "ec_VaoF9dkHxhu" = 0xA7F5195F3AFA;


SELECT "ec_88X3B7PMoKugS8QqaefyiVFiL" AS "ec_hPvZZPPeJvhPcYY",
    /* ec: */"ec_7WrxWstKgq" ,
    /* ec: */"ec_VaoF9dkHxhu"
FROM "ec_4dUsSG"."ec_Lw41GVe"
WHERE "ec_7WrxWstKgq" IN (0xC97C91074603) AND "ec_VaoF9dkHxhu" = 0x53B679C00CA0
```

After the queries are executed the individual resultsets are merged (and sorted if need be) before returning the final resultset to the user.

Cassandra also offers so called slicing conditions that allows for multiple columns to be compared to multiple values – e.g. SELECT ... WHERE (col1, col2) < (1, 2), which has the semantics of ((col1 < 1) OR ((col1 <= 1) AND (col2 < 2))). The motivation for slicing conditions was to offer a way of formulating some disjunctive conditions while still being able to satisfy queries with only sequential reads from storage – which would not have been possible with a general OR operator. Slicing conditions are not supported by EncryptedCassandra because of the complexity involved in rewriting (and splitting up) the statements as the number of columns in the conditions grows.

Other features that Cassandra offers for select statements like static columns, aliasing of column names, the COUNT(*) function or the TTL() and WRITETIME() functions are fully supported by EncryptedCassandra.

### *3.2.3.4 Updates*

Updates are similar to selects, but have an additional restriction in that they need to contain the full primary key, i.e. they also contain the full clustering primary key. Furthermore, no inequality operators may be used in the where clause.

As such, updates are fairly straight-forward to encrypt. Noteworthy however is that queries still may need to be split up (similar to selects), if multiple rows are being updated by the unencrypted statement through the use of an IN() condition on the partitioning primary key.

Updates do not allow for any data to be modified depending on its current value – i.e. statements of the form UPDATE col = col + 1 are not allowed. However the syntax for incrementing and decrementing counters as well as adding and removing elements from collections does make use of the plus and minus operators. When rewriting queries like these it is sufficient to simply only encrypt the actual data in the query without modifying the operation itself, since no actual additions or subtractions are performed.

Considering that, this statement to append an element to a list

```
UPDATE lst SET fruits = fruits + ['apple'] WHERE id = 1
```

can simply be encrypted to

```
UPDATE "ec_4dUsSG"."ec_MxCb6YK"
SET "ec_9bBCrfrsAk1" = "ec_9bBCrfrsAk1" + [0x5F6E2B530EB38F]
WHERE id = 1
```

### *3.2.3.5 Deletes*

Deletes face similar restrictions as updates for their where clause, but they can also fully omit the clustering primary key if no individual collection elements are being deleted and no Lightweight Transactions are being used. This means that EncryptedCassandra can fully support deletes, since all data necessary to encrypt the statement is also always required by Cassandra for the statement to be a valid one. Note though that splitting up statements (like it is done for selects and updates) may still be necessary, if the IN() operator is used on the partitioning primary key when not omitting the clustering primary key.

A delete on the emp table of Subsection 3.2.3.1 could be as follows:

```
DELETE FROM emp WHERE empid IN (1,2) AND deptid = 5
```

After encryption the statement would be split up into these two:

```
DELETE FROM "ec_4dUsSG"."ec_Lw41GVe"
WHERE "ec_7WrxWstKgq" IN (0xC97C91042660) AND "ec_VaoF9dkHxhu" = 0xA7F5195B7C23

DELETE FROM "ec_4dUsSG"."ec_Lw41GVe"
WHERE "ec_7WrxWstKgq" IN (0xC97C91074603) AND "ec_VaoF9dkHxhu" = 0x53B679C44F99
```

### *3.2.3.6 User Management and Authorization*

Cassandra also offers a variety of statements to create users and manage privileges. EncryptedCassandra only encrypts identifier names in these statements and does not interfere with Cassandra's privilege system (or make use of it) in any other way.

A grant statement to give user u1 I/U/D privileges on the keyspace t1 looks as follows:

```
        GRANT MODIFY ON KEYSPACE t1 TO u1;
```

This would be rewritten by EncryptedCassandra to:

```
        GRANT MODIFY ON KEYSPACE "ec_4dUsSG" TO u1;
```

### 3.2.4 Limitations Imposed by the Encryption

As was shown in subsections 3.1.2 and 3.2.3, the limitations imposed by the encryption scheme and the ones imposed by Cassandra/CQL themselves are fairly similar, since by design the data necessary to query the tuples is often times identical to the data necessary to decrypt it. This subsection briefly summarizes those limitations, and presents additional ones that apply across multiple types of queries.

Point-queries are fully supported. When using nested IN() conditions the query is split up into multiple individual queries internally and the result sets are then merged prior to being returned to the user (an example of this is provided in Subsection 3.2.3.3).

Range queries on the other hand require that the attribute that is being queried for with an inequality condition be encrypted using either OPE or cluster-OPE. The use of ALLOW FILTERING however is not supported, as there is no practical way to implement it, since the data necessary to encrypt the values in the query itself may not be present.

Non-trivial server side functions (e.g. functions to get the current date, generate random values or convert data types, as mentioned in Subsection 3.1.3.5) are not applicable to encrypted columns, since the functions would need to be fully reimplemented within the encryption layer. They can however still be used on unencrypted columns without any restrictions. The other functions (i.e. COUNT(*), WRITETIME() and TTL()) can however be used on encrypted columns without any restrictions, as they are not affected by the encryption.

The use of inequality conditions on collections is not supported, since OPE for collections is not supported. This only affects the use of collections in the IF clause (i.e. Lightweight Transactions), since that is the only place Cassandra itself allows for the use of inequality conditions on collections.

Counters are almost fully supported – the only hard limitation for them is that they may not be used if the table is defined as using COMPACT STORAGE[9]. Additionally, since the counters are being fully reimplemented atop CQL tables, the performance is likely to be significantly worse than for Cassandra's native counters – especially since Cassandra's own implementation tries to converge per-node counter states, which cannot be done when only having access to the CQL API of the database.

EncryptedCassandra supports the indexing of data through secondary indices by

- providing the options for it to remain unencrypted or be encrypted with OPE, thus allowing it to be indexed through Cassandra's native secondary indices
- implementing a secondary index facility of its own for encrypted data that does not use OPE

[9] COMPATC STORAGE instructs Cassandra to use an alternate internal table layout that precludes the use of more than one non-key column in a CQL table, thus making the existence of multiple counters in a counter table impossible. However COMPACT STORAGE is mainly a legacy option created for CQL tables to be more easily accessible over the Thrift API. Its usage has no further API-level implications.

Note that data already existing in a table prior to the creation of the index will not be indexed by EncryptedCassandra and only data that has been added after index' creation will be considered by queries using the index. Indexing data already stored in the table prior to the index' creation would require fetching all of the table's data onto the client first, in order to be able to index it, which is not practically viable.

### 3.2.5 Characteristics of the Encryption Approach

This subsection discusses the properties of the presented encryption scheme, based on the criteria used in Subsection 2.3 to compare the various database encryption approaches.

**Target Platform:**
The general encryption scheme presented can be applied to any extensible record store, as discussed in Subsection 3.2.1. The specific implementation and the presented details are for Cassandra.

**Trust Model:**
The encryption occurs on the client side and no unencrypted data needs to be exposed to the database, as such, no third parties need to be trusted.

**Completeness of Encryption:**
The data stored is fully encrypted. However there also exists the option to leave certain parts of the data unencrypted, in order to be able to perform some server-side operations on it, like data type conversions.

**Patterns across Ciphertexts:**
As discussed in Subsection 3.2.2 duplicates of encryption keys should be impossible based on the data model, however this is only true if no compound primary keys are used. Earlier versions of Cassandra did not provide direct support for compound primary keys and users often manually simulated the feature by serializing multiple attributes into just one blob that was then used as the primary key. Starting with CQL3 however Cassandra offers direct support for compound primary keys at the CQL level.

The issue with compound primary keys is that all leading key fragments (i.e. up to but excluding the very last one) may be identical across different tuples while also being encrypted by identical encryption keys since no other parts of the access path are available for inclusion into the encryption key of those leading subkeys.

While this may seem like a massive compromise of the goal to not have data patterns leaking through the encryption at first, it is important to note that:

- A similar issue would occur if a database wouldn't directly support compound primary keys and users continued to manually serialize data into primary key blobs, if those blobs grew large enough to exceed the block size of the used block cipher (usually 16 bytes).

- Taking a closer look at how attributes are transposed into tuples in Cassandra (see subsections 3.1.1 and 3.1.2) being able to identify identical subkeys across multiple. tuples in the CQL data model is technically the same issue as being able to identify the number of attributes of a tuple in the internal data model.

Considering these two points, there is no way to avoid this issue short of maintaining a second database of only encryption keys (or initialization vectors) much like CryptDB [35] does.

Note also that this only affects attribute values of primary key columns. All other attribute values still have fully unique encryption keys.

This issue is especially relevant for EncryptedCassandra’s secondary indices though, as the indexed attribute will always be the leading part of the primary key in the index table, thus revealing the distribution of that attribute in the data, if it were possible for an adversary to somehow identify the data table and the indexed attribute that an index table corresponds to.

Additionally to leading sub-keys, since CQL 3 forces its users to commit to a fixed schema unlike earlier versions of Cassandra that were mostly “schema-free”, it is possible to identify even after the encryption which parts of the schema have been left blank (i.e. which values are missing), since querying the encrypted data returns nulls for values that are not present.

Encrypting null values however is not practical since, as mentioned in Subsection 3.1.3.1, Cassandra uses null values to indicate the deletion of data. Thus the encryption of null values would also require manual management of tombstones in the database, which is something that cannot be implemented on the client-side without significant effort as well as compromises in terms of performance.

**Performance:**
Detailed benchmarks on how the encryption impacts the database performance are provided in Subsection 5.2. The presented encryption approach is practically viable, though care needs to be taken when using OPE.

Other than when using OPE, the time spent actually encrypting and decrypting the data is negligible. Most of the incurred overheads are in the code used to rewrite the queries and setup the encryption, which is a mostly static amount per CQL query. However significant performance penalties occur, when database features that need to be reimplemented by EncryptedCassandra atop CQL are used, i.e. counters and secondary indices.

Using OPE requires on the order of 20 to 100 milliseconds per value that is to be encrypted, depending on the actual domain and range of the values. Thus systems that utilize OPE need to be structured such that the encryption of the values can be spread out across a wide range of clients, so that the overhead remains insignificant for the individual user.

The alternative to this would be concentrating OPE operations on high-performance servers that can batch process and optimize them, like CryptDB [35] does. However EncryptedCassandra does not implement CryptDB’s optimization that allows for the batch processing of OPE operations.

**Functionality:**
The functionality of the database after encryption was discussed at length in subsections 3.2.3 and 3.2.4. Generally all point queries as well as most range queries are supported. Limitations apply when using various features that exist for reasons of backwards compatibility or for developer convenience.

**Cloud Compatibility:**
The presented approach can be fully implemented within the database client drivers, as such, no modifications to cloud-hosted components need to be made and no further third-party services are required, making EncryptedCassandra fully cloud compatible.

**Summary**

EncryptedCassandra tries to support as much of Cassandra's original functionality as possible while still remaining fully cloud compatible. This does come at the cost of some performance.

Compared to the other approaches presented in Subsection 2.2, EncryptedCassandra is closest to CryptDB (see Subsection 2.2.3), on a conceptual level. Both systems try to expose as little details about the plain text data to the cloud service as possible, while also trying to avoid post-processing of the queries on the client side, and aiming for transparency towards users through query rewriting. Nevertheless, the actual implementations differ significantly. Since CrypDB targets a vastly more complex database (at the API level), their implementation decisions differ from EncryptedCassandra's – e.g. CryptDB uses generated unique IVs to avoid patterns leaking across ciphertexts and requires an actual proxy server to perform the rewriting/encryption and data maintenance tasks.

# 4 Implementation

This section presents the overall architecture of EncryptedCassandra's implementation, as well as the general structure of its source code. Additionally it provides the details on the various database features that have been reimplemented within the encryption layer, so as to maintain the database's functionality after encryption, as well as offer API compatibility and transparency to users.

## 4.1 Architecture

Starting with the introduction of CQL 3 Cassandra no longer provided any official database client drivers any more, and instead only exposed a binary protocol to communicate with the database over the network (alongside the pre-existing Thrift-API that continues to be maintained for legacy reasons). After this change a variety of client drivers for various programming languages emerged, with the new Datastax Java driver[10] effectively becoming the de-facto official driver, since Datastax is also the entity leading the development effort on Cassandra, despite the project itself being open source and part of the Apache Software Foundation.

EncryptedCassandra is implemented as a wrapper around this Datastax Java driver and encrypts queries prior to handing them over to the actual driver as well as decrypting any returned result sets prior to returning them to the calling application. Additionally it automatically performs any other operations necessary like updating the data dictionary tables or keeping secondary indices synchronized.

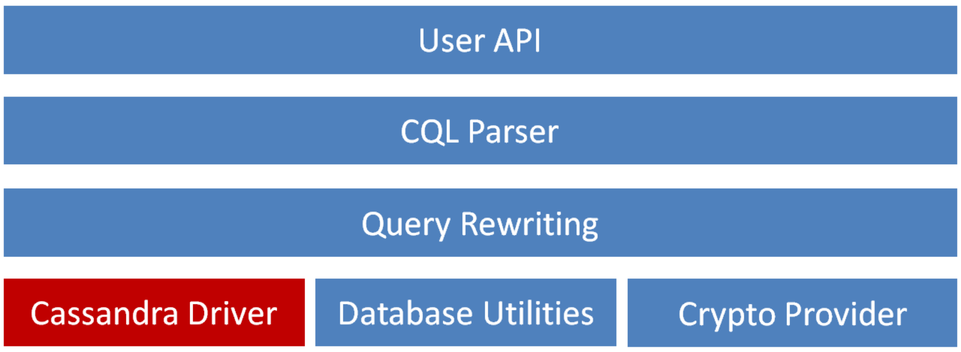


**Figure 5:** Architecture of EncryptedCassandra

Figure 5 shows EncryptedCassandra's overall architecture.

The User API serves two purposes:

- It provides users with an API to access the database which is compatible to Cassandra's, so that no client code needs to be changed to interface with the database when using EncryptedCassandra.

- It provides users an API to interact with EncryptedCassandra itself in order to e.g. request specific services (like rewriting queries) or to configure the specific details used for the encryption.

The CQL Parser is used to identify the individual statements and extract the relevant data out of them. These parsed statements are then processed in the Query Rewriting module, which coordinates the encryption process and performs any other steps necessary to successfully rewrite the query such that it satisfies the user requirements.

[10] https://github.com/datastax/java-driver

The Crypto Provider is responsible for encrypting individual fragments of data as well as encoding any ciphertexts into valid values, if they need to be used as identifiers. It also implements the actual process that is used to chain the keys together for the encryption (see Subsection 3.2.2). It does however not itself implement any cryptographic primitives – that step is delegated to a third-party encryption library (see Subsection 4.3). The Query Rewriting module makes use of the Crypto Provider in order to obtain the ciphertexts to be used in the rewritten queries.

The Database Utilities module contains the various bits of functionality that are necessary for EncryptedCassandra to be as compatible with Cassandra and as transparent to the user as possible. Specifically these are:

- A data dictionary and a serializer to perform type checks and conversions that Cassandra would otherwise provide. This is necessary since after the encryption all data stored in the database is of either blob (default) or varint (for OPE) type.

- A secondary index facility, if the user wants to use secondary indices but does not want to use OPE or to leave data unencrypted. This is necessary as encrypted data cannot be meaningfully indexed by Cassandra's native indices, unless it is OPE encrypted.

- A reimplementation of Cassandra's Counters atop the CQL interface. This is necessary as otherwise it would not be possible to encrypt counters, since they would require encrypted data to be incremented or decremented on the server side.

Finally the Cassandra Driver, while not strictly part of EncryptedCassandra itself (thus the different color for it in Fig. 2), is still part of the architecture as it provides the actual Java classes and interfaces necessary to provide users with a Cassandra-compatible API, as well as be able to actually issue any queries to Cassandra instances.

## 4.2 EncryptedCassandra's Features in Detail

This subsection presents the database features that are implemented by EncryptedCassandra, in order to not limit the database's functionality after encryption.

### 4.2.1 Data Dictionary and Serialization

Cassandra provides various data types that can be used for the columns of a CQL table. However the encryption primitives used can only encrypt binary blobs (symmetric block ciphers) and integers (OPE). In order for the encryption to remain API compatible to Cassandra, as well as to provide the same data integrity checks to users that Cassandra does, the data needs to be manually serialized prior to the encryption (and deserialized later when returned in the result sets). Additionally a separate data dictionary needs to be maintained to keep track of the data types as well as encryption specifics of the columns of a table.

Upon creation of a new schema in Cassandra EncryptedCassandra implicitly creates a new (fully encrypted) table called ec_catalog_col that contains all the encryption details of all the columns of all tables in the current schema.

ec_catalog_col has the following columns:

- Name of column family (table)
- Name of column
- Position of the column in the table definition, if the column is a primary key column

- Role of the column
  (partitioning/clustering primary key, non-key column or static column)
- Column type prior to encryption
- OPE Type of encryption (OPE, Cluster-OPE or none)
- OPE domain size
- OPE range size
- Whether the column is encrypted or not
- How many bytes of random data list elements of the column will contain, if the data type is a list collection.
- Whether the column is indexed and how specifically
  (Indexed by EncryptedCassandra's secondary indices with or without ticketing, indexed by Cassandra's native secondary indices, or not indexed)
- Name of the index for the column, if EncryptedCassandra's secondary indices will be used for this column

Additional data dictionary tables are used for EncryptedCassandra's secondary indices, these are however covered in Subsection 4.2.3.

The parsing of the data values out of the CQL queries, their serialization into binary blobs, and deserialization into java objects later on is delegated to Cassandra's own serialization code, so as to remain fully compatible with it.

After the encryption, other than unencrypted columns and null values, all data is either an integer of arbitrary length (for OPE and cluster-OPE columns) or a blob (for non-OPE columns). The exception to this are unencrypted columns, which remain fully untouched and thus retain their data type, so that any server-side database function (e.g. type conversions) can be directly applied to them.

### 4.2.2 Order Preserving Encryption (OPE)

For OPE columns the data needs to be treated differently, as the OPE implementation used can only encrypt integers into other integers and hence a binary representation for the various data types needs to be used that ensures that the lexicographical order of the serialized values preserves the original order of the values prior to serialization.

This is done as follows:

- For **integer**-like data types (Integer/Int, Long/BigInt, BigInteger/Varint) a straight-forward 2's-complement representation is used. It is however preceded by an additional byte to indicate the sign of the actual number in order to "flatten out" the discontinuity of the representation that would otherwise cause negative numbers to be lexicographically larger then positive ones. Default values for the OPE domain of integers are 5 bytes for Int, 9 Bytes for Long and 128 Bytes for BigInteger.

- For **float**-like data types (Float, Double, BigDecimal/Decimal) a completely different serialization approach is used. The first 2 bytes of the serialized value indicate whether the number itself and its exponent respectively are positive or negative. The next 4 bytes contain the value of the exponent, serialized using a conventional 2's-complement representation. Finally an unscaled arbitrary-length/full-precision 2's complement representation of the mantissa (padded with 0s to up the next full byte boundary) is written. Default values for the OPE domain of floats are 9 bytes for float, 13 bytes for doubles and 128 bytes for BigDecimal.

- **Date and time** values are treated as timestamps, i.e. 8 byte Longs.

- Universally Unique Identifier (**UUID**) types are treated differently depending on the type of UUID. If they contain date and time value then that value will first be extracted and serialized as a timestamp, before being prepended it to the serialized UUID itself. The serialization of the actual UUID on the other hand occurs identically to Cassandra itself. This does mean than non-date UUIDs will be sorted differently by Cassandra compared to EncryptedCassandra, the difference should however be insignificant, since these will be random values to begin with. Default values for the OPE domain of UUIDs are 25 bytes for UUIDs with dates and 16 bytes for those without.

- For **string**-like types (Text, ASCII) Cassandra's own serialization methods are used – i.e. ISO-8859-1 for the ASCII type and UTF8 for the text type. This works out mostly fine for ordering most English texts as both serialization formats assign larger byte-values to letters that occur later in the English alphabet, it is however not fully compatible to Cassandra's sort order, which uses proper collations when sorting strings. Default value for the OPE domain of strings is 128 bytes.

- For **booleans**[11] Cassandra's own serialization methods are used. The default OPE domain size of booleans is 1 byte.

- The **IP** (inet) data type is serialized by using the 2's-complement representation of each individual octet, preceded by one additional byte to indicate whether the octet's value is positive or negative. This is compatible with both IPv4 as well as IPv6 addresses. The default OPE domain size for IPs is 8 bytes (i.e. IPv4 addresses are assumed by default).

While these representations are somewhat wasteful as far as space-efficiency is concerned, they were nevertheless chosen due to being intuitively understandable as well as debugable.

Columns that are to be OPE encrypted are duplicated in the actual tables stored in the database. One of them contains the actual value encrypted using a traditional block cipher (that is read and used for decryption if the user queries its value) and another one that contains the OPE encrypted value which is exclusively used for querying the database and sorting the result sets. This is done as the runtime cost of decrypting an OPE value is several orders of magnitude higher than the cost of encrypting and then decrypting an additional value using symmetric block ciphers.

Additionally to these transformations, datatypes that can vary in length (like strings and arbitrary-size integers) need to be padded or cropped off in order to perfectly fit the OPE domain. Otherwise e.g. strings of different lengths would be serialized into integers that will not necessarily preserve the lexicographical order of the strings. The padding is done by filling up the least significant bits of the domain with zero-bytes.

These padding and cropping operations have 2 interesting effects:

- For string types it allows to simulate a startsWith()-type functionality. E.g. to query for strings that start with "a" this query could be used:

```
SELECT .. FROM tbl WHERE pk=.. AND ck_ope >= 'a' AND ck_ope < 'b'
```

[11] Note that OPE support for booleans is being provided since booleans are simply another datatype in Cassandra. Whether using OPE on booleans in practice is sensible or viable is left up to the user to decide.

This will return tuples having e.g. “a” or “abc” in ck_ope but not “b” or “bcd”. (This behavior of EncryptedCassandra is identical to Cassandra itself).

- Since the actual value used for decryption later on will be fully stored in another column, the cropping of values to make them fit the OPE domain sizes can introduce a binning effect if values frequently exceed their domain size. A drawback of this is that OPE columns used in the primary key effectively reduce the possible keyspace available to tuples, if the domain size configured for the column is insufficient.

### 4.2.3 Secondary Indices

EncryptedCassandra already allows for values to be indexed through Cassandra’s native secondary indices by either not encrypting them or encrypting them using OPE. However if the security implications of these alternatives are unsatisfactory, it also provides the option to use secondary indices that are reimplemented atop (separately encrypted) CQL tables and transparently maintained by EncryptedCassandra itself.

Specifically, an EncryptedCassandra secondary index is simply a CQL table consisting of the indexed value as the partitioning primary key and all primary key columns of the indexed table as clustering primary keys.

For example creating an index with

```
CREATE INDEX idx ON emp(last_name);
```

on this table

```
CREATE COLUMNFAMILY emp (
  empID int,
  deptID int,
  first_name text,
  last_name  text,
  PRIMARY KEY (empID, deptID)
);
```

would result in the following table being created to store the index:

```
CREATE COLUMNFAMILY ei_idx (
  index_value text,
  empID int,
  deptID int,
  PRIMARY KEY (index_value, empID, deptID)
);
```

The index_value column of the ei_idx table contains the values of the last_name column of the emp table.

The actual index table will still be fully encrypted. The statement has been left unencrypted here for clarity. The index name in the create index statement may be omitted – in this case EncryptedCassandra will assign a random valid name to the index that is not already in use.

The main challenge for these self-maintained secondary indices is how to keep them synchronized with the primary data, despite Cassandra’s limited ability to provide ACID-like consistency across multiple independent queries, while also not using any third-party synchronization services.

To illustrate the issue consider the following scenario: two independent clients update the same indexed attribute of a tuple – both clients need to issue two separate queries to the database to do so. One to update the attribute in the table and one to update the index table for that attribute. Since the order in which statements sent to the database are applied to the data is not guaranteed in any way by Cassandra (since failures may occur in between or during the execution of statements), there is a very real chance that the data-operation of the one client could be interleaved with the index operation of the other client, thus resulting in index data and primary data becoming out of sync.

These scenarios cannot be avoided in Cassandra without the use of some other synchronization service to lock rows prior to updates. They can however be resolved after the fact, by one of these alternatives:

- Manually tagging attributes in the data and index tables with writetime tickets, as well as manually issuing queries for cleanup operations (much like Amazon's Dynamo [13]) when queries by clients return conflicting results.
- Using triggers [43], since they are guaranteed to execute whenever the triggering statement is executed.
- Exploiting the algorithm that Cassandra uses to converge conflicting updates, in order to effectively achieve an implicit rollback if primary data and index data ever get out of sync.

For EncryptedCassandra the third alternative was chosen, since manually issuing cleanup queries would be significantly more complex, as well as likely have significantly worse performance, compared to relying on Cassandra's internal cleanup mechanism. As for triggers, they are still very much in their infancy – they only exist as instead-of triggers and they require the trigger code to be loaded as a compiled library within the database server process. Additionally, while the triggers are guaranteed to be executed, no guarantees are being made for the effects of the data operations performed by those triggers. Thus it is ultimately unclear how feasible a practical implementation of secondary indices through triggers currently is in Cassandra, but it remains an interesting alternative nonetheless, as the functionality provided by triggers is expected to expand in future versions.

In the case of conflicting updates Cassandra resolves the situation by simply declaring the update with the higher timestamp the winner of the conflict, when converging the data. Up until the point of convergence the data may still be out of sync. This is however expected behavior of an eventually consistent database like Cassandra. Thus the only thing left to be done in order to ensure that index data and primary data remain in sync is to ensure that both queries will (eventually) get executed and that both queries will be executed with the same writetime timestamps. Both of these requirements can be met by using atomic batches[15] in Cassandra. Note that atomic batches in Cassandra are not truly atomic in the ACID-sense[24] that the effects of the queries will be rolled back if errors occur. Instead they only guarantee that eventually all statements within the batch will be executed. This is however sufficient to ensure a rollback-like functionality through the execution of other batches (eventually) overwriting conflicting data by either updating the index table or the data table.

Cassandra uses millisecond granularity for its writetime timestamps. If 2 writes on the same data field occur within the same millisecond, i.e. "simultaneously", then the write with the higher value is considered the winner (with null, i.e. deletes, being considered the highest value). Writes on the same data field within the same millisecond are not realistic, since they would need to occur on the same physical node and the workload involved in one write will usually exceed one millisecond of wall-clock time. Nevertheless EncryptedCassandra still addresses this issue by providing an optional ticketing feature for updates that involve index updates.

This ticketing feature provides each query that writes indexed data with a unique number (the "ticket"). These tickets are in ascending order and are assigned to queries prior to their execution, thus ensuring that a total chronological order across all queries that affect one indexed attribute exists, and thus avoiding the issue of multiple queries having identical timestamps, when being considered by Cassandra's convergence mechanism.

The implementation of the ticketing feature is essentially just an additional data dictionary table (ec_index_ticket) consisting of the three columns: column family, column and last_ticket_nr. In order for a client to receive a ticket number to perform an update on an indexed attribute it needs to perform a Lightweight Transaction on this table to increase the last_ticket_nr for the particular column in the particular column family. If the Lightweight Transaction succeeds, then the client has successfully obtained the ticket. If not, it needs to retry with a new ticket number. If multiple indexed attributes in one tuple are being updated then first the highest last_ticket_nr across all attributes needs to be determined, then a ticket number with the next-highest integer needs to be obtained for all those attributes.

After all necessary tickets have been obtained, an atomic batch including the data as well as the index updates is performed, with the writetime of the batch being set to the ticket number, thus precluding the possibility of independent batches having identical writetime timestamps.

While this works, it should be noted however that it does have a relatively high impact on performance, since Lightweight Transactions require a Paxos run across all nodes responsible for the affected tuple, involving 4 round trips [17].

#### 4.2.4 Counters

Counters are implemented using a straight-forward Positive/Negative Conflict-Free Replicated Data Type (PN-CRDT, [39]) and thus relatively similar to Cassandra's own internal implementation.

A counter table consists of the specified primary key columns, one additional primary key column (a UUID) that is required to identify individual increments and decrements and finally one non-key column per counter in the table definition. Each time a counter is incremented or decremented a new row is inserted into the table with the amount the counter is being increment or decremented by. When counter values are read by clients all these individual increments and decrements are selected and added up, resulting in the final value that is then returned to the client.

The main difference to Cassandra's counters is that Cassandra tries to converge counter values at least within individual nodes. Converging of counter values is however not something that can be easily performed automatically by EncryptedCassandra, since care must be taken to not interfere with other ongoing counter operations, as well as not corrupt the overall counter value with failed convergence attempts.

### 4.3 Structure of the Code

EncryptedCassandra is structured into 5 main packages:

- The api package contains classes and interfaces that are expected to be touched by client code. This includes the EncryptedCassandra class that is used to wrap the session object, as shown in Subsection 5.1.9.1, but also classes that wrap rows and resultsets returned by Cassandra.

- The catalog package is concerned with the maintenance of the data dictionary, the secondary indices, as well as the serialization and deserialization of data.

- The cql package contains the CQL parser and is also responsible for performing the actual rewriting of the queries.

- The crypto package contains the implementations of the cryptographic services needed to encrypt and decrypt individual values.

- The cqlpad package contains the GUI shown in Subsection 5.1.9.2.

These packages are mostly self-contained and access each other through the API package that provides references to the classes in the other packages, depending on the specific configuration.

Several packages utilize third-party code in one way or the other in order to perform their work:

- The catalog package uses Cassandra's own code for most actual serialization and deserialization operations. As a consequence of this a full copy of Cassandra's codebase needs to be loaded in the client process that wants to use Encrypted-Cassandra.

- The crypto package uses the OPE implementation of CryptDB [35] as well as the AES and EAX implementations of the BouncyCastle[12] library.

Additionally, the cql package uses classes generated by ANTLR [33] based on the CQL grammar.

CryptDB's OPE implementation is written in C++. Porting the codebase to Java, while practically feasible, would nonetheless require a major effort and is outside the scope of this thesis. As such, the code was instead compiled and called from the Java process in order to perform the actual OPE operations. Generally two options were available to call the C++ code from Java: either by loading it into the same JVM process and calling into it using JNI[13] or JNA[14] (i.e. loading compiled native code as a shared library into the JVM and calling it using one of Java's foreign-function interfaces), or by running it in an individual process and exchanging the data through standard inter-process communication (IPC) facilities. Ultimately the latter approach was chosen since the code makes use of NTL[15] (a C++ number theory library), which is not thread-safe, which in turn would mean that at most only one OPE query could be encrypted at any one time within one JVM process, if the code were to be loaded into it. In addition to this, the OPE code is implemented using recursion and thus requires large stack sizes (over 2MB), depending on the size of the data to be encrypted, whereas JVM threads traditionally have relatively small stacks (0.5 MB by default). While spinning up an entire process for the encryption of only one value may seem wasteful, the actual runtime of the OPE encryption overshadows the process creation time by far.
For the non-OPE encryption primitives the BouncyCastle library was used instead of the JDK-provided primitives, since JDK installations generally need to be reconfigured with the "Unlimited Strength Jurisdiction Policy Files"[16] in order for some encryption primitives to be available, whereas the BouncyCastle library can be bundled with the rest of EncryptedCassandra and run on client machines without further concerns.

[12] https://www.bouncycastle.org/
[13] Java Native Interface: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/
[14] Java Native Access: https://github.com/twall/jna
[15] NTL is not an abbreviation and instead the library's actual name: http://www.shoup.net/ntl/
[16] http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html

## 4.4 Transparency to the User

EncryptedCassandra is fully implemented within the client drivers for Cassandra and automatically rewrites and splits up queries as needed for the encryption, while also later on decrypting and merging result sets as needed. Additionally it performs any data management tasks (e.g. for the data dictionary or the secondary indices) automatically without the need of further user intervention.

The implementation is transparent to users (i.e. developers), except in these scenarios:

- Setup:
  EncryptedCassandra needs to first be set up and configured at runtime – i.e. it is not as simple to use as just using another driver library and having the encryption immediately work. Thus using it does require some (if only minimal) integration effort.

- DDL:
  Some of the encryption features offered require configuration details to be provided on-use, i.e. after the initial setup of library itself has been completed (e.g. to indicate which columns of a table should be OPE encrypted or remain unencrypted). To allow the use of these features as unobtrusively as possible, EncryptedCassandra minimally expands the CQL language where necessary, giving users the option to provide additional configuration details. Note though that these features are fully optional – i.e. CQL DDL can be used without breaking transparency, if these features are not used.

- Functionality:
  Some functionality of Cassandra is simply not supported (see Subsection 3.2.4 for further details on this), thus breaking transparency whenever users try to run queries that use those features.

Some functionality, despite being otherwise API compatible, may nevertheless have significantly different performance characteristics that could possibly preclude its usage practically. Examples for this are EncryptedCassandra's secondary indices (Subsection 4.2.3) and counters (Subsection 4.2.4).

Additionally to this, EncryptedCassandra also alters the semantics of the Java driver API for these features:

- Streaming of result sets (i.e. the continuous retrieval of additional rows from the database for a query, without having to fully buffer the query in memory) is not supported, and result sets are instead fully buffered in memory, even if instructed otherwise over the API. This is necessary since some queries may need to be split up into multiple queries after the encryption resulting in multiple result sets that need to be merged before being returned to the user (and thus need to first be fully buffered on the client).

- Prepared statements are supported on the API level, however they are always fully expanded on the client side prior to being sent to the database, thus EncryptedCassandra does not truly allow for the use of actual prepared Statements. This is necessary for being able to correctly rewrite the queries.

# 5 Evaluation

This section uses a simple example to showcase how EncryptedCassandra can be used and what its effects are.

## 5.1 Sample Use Case

The example presented in this subsection is that of a database used to store emails. The focus of the example is on presenting the effects of the query rewriting and how the encryption affects the database. Thus only the database aspects themselves are covered. Another example, that focuses more on operational aspects and motivations instead of data modeling and queries, of how Cassandra is being used in the industry for a mail-like messaging platform can be found in [41].

### 5.1.1 Data Model and DDL

The database consists of 3 tables:

- Mailbox: contains the data that is presented to users when browsing the various mailboxes (e.g. sender, subject and date).
- Mailcontent: contains the actual e-mail messages.
- Keyword: contains a word count (per word) of the emails in one folder.

The purpose of the keywords table is to allow the mail service provider to show contextual ads to their users.

This is the CQL DDL statement to create the mailbox table:

```
CREATE COLUMNFAMILY mail.mailbox (
       id_user    text,
       folder     text,
       msg_date   timestamp OPE,
       id_msg     text,
       msg_to     set<text>,
       cc         set<text>,
       bcc        set<text>,
       msg_from   text,
       subject    text,
       is_unread  text,
       PRIMARY KEY ((id_user, folder), msg_date, id_msg)
);
```

The msg_to, cc and bcc fields are declared as collections (sets without duplicates), as an email can be addressed to multiple recipients. The date field are encrypted using OPE in order to allow users to query for mails in a specific time frame. The id_user is the username that a user is known by within the mail system, and the folder is the place that the user has stored the mail within their account. Additionally, the partitioning primary key is being explicitly defined as (id_user, folder) by the extra set of parentheses.

After encryption, the DDL statement looks as follows (the additional DML statements to populate EncryptedCassandra's data dictionary with the new table's metadata have been omitted here for brevity):

```
CREATE COLUMNFAMILY "ec_2H2cQFqXA"."ec_2xtFR7kck5rYo" (
       "ec_PFN3g1o5gnSj47QESdEt7Cr"       blob,
       "ec_UCPmhC9bSYzF"                  blob,
       "ec_8LZAX7GTQ8R"                   blob,
       "ec_2qjpdKjMfjUBFq"                varint ,
       "ec_7ANvaayjA7u"                   blob,
       "ec_6Un9PZEADKi"                   set<blob>,
       "ec_2CQpLg"                        set<blob>,
       "ec_6PumWDi"                       set<blob>,
       "ec_2qjpdKkP8Wwz7q"                blob,
       "ec_oMBuDZGbVh5X"                  blob,
       "ec_A7eewUi23ZHWA8u"               blob,
       PRIMARY KEY (("ec_UCPmhC9bSYzF", "ec_8LZAX7GTQ8R"),
              "ec_2qjpdKjMfjUBFq", "ec_7ANvaayjA7u")
);
```

The OPE-encrypted column is of varint type after the encryption and an additional column was added to store a symmetrically encrypted ciphertext of the date, so as to avoid having to decrypt it later on (for performance reasons). For collections only the "collected" data type was changed to blob, the collection itself remains unchanged.

Additionally, the msg_date and msg_from columns are indexed in order to allow searching for emails by specific senders as well as for searching within certain timeframes.

```
CREATE INDEX mbx_date ON mail.mailbox(msg_date);
CREATE INDEX mbx_from ON mail.mailbox(msg_from);
```

Since the msg_date column is OPE encrypted, a Cassandra index is used for indexing it, thus the first statement will be encrypted as follows:

```
CREATE INDEX  ON "ec_2H2cQFqXA"."ec_2xtFR7kck5rYo"("ec_2qjpdKjMfjUBFq");
```

The msg_from column on the other hand is symmetrically encrypted, and thus a secondary index maintained by EncryptedCassandra needs to be used for it. A new table is implicitly created to hold the data for this index:

```
CREATE COLUMNFAMILY mail.ei_mbx_from (
       index_value  text,
       id_user      text,
       folder       text,
       msg_date     timestamp,
       id_msg       text,
       PRIMARY KEY (index_value, id_user, folder, msg_date, id_msg)
);
```

The table contains one column for the indexed value as well as one column for every primary key column in the indexed table. Every column of the index table is part of the primary key, with the index_value column being the partitioning primary key. However the actual query executed will be the following one, as index tables are fully encrypted:

```
CREATE COLUMNFAMILY "ec_2H2cQFqXA"."ec_DBiRQqQZg9EonwLn6R" (
       "ec_4JBj7iUsVSNRc9pPxM" blob,
       "ec_WEYh2uYhteJG"       blob,
       "ec_7nbEMwwGp7f"        blob,
       "ec_2zrrDUaPd8JXJf"     blob,
       "ec_7d9YHjxX82s"        blob,
       PRIMARY KEY ("ec_4JBj7iUsVSNRc9pPxM", "ec_WEYh2uYhteJG",
              "ec_7nbEMwwGp7f", "ec_2zrrDUaPd8JXJf", "ec_7d9YHjxX82s")
);
```

The index for the msg_from column can be created as a ticketed index, using:

```
CREATE TICKET INDEX mbx_from ON mail.mailbox(msg_from);
```

In this case the ei_mbx_date table remains unchanged, however an entry for ec_index_ticket is generated, to setup the ticket management:

```
INSERT INTO mail.ec_index_ticket
(cf, col, last_ticket_nr)
VALUES
('mailbox', 'msg_from', 0);
```

Since the ec_index_ticket table is fully encrypted, the actually executed statement looks as follows:

```
INSERT INTO "ec_2H2cQFqXA"."ec_2NiFhZMRh6Zu1nHR3MhfVWmo"
("ec_4Hmogz", "ec_FWo6cY7", "ec_Jjx2bJjAKPg69o3JWzsLfj")
VALUES
(0x4E82AFC1726CD13AC2, 0x13E60A4F44FD582A4006,0xAC47FC02186E);
```

For the mailcontent table this DDL statement is used:

```
CREATE COLUMNFAMILY mail.mailcontent (
       id_msg      text,
       msg_date    timestamp,
       msg_to      set<text>,
       cc          set<text>,
       bcc         set<text>,
       msg_from    set<text>,
       subject     text,
       msg_txt     text,
       attachments map<text, blob>,
       PRIMARY KEY (id_msg)
);
```

A map is used for the attachments column in order to store both the name as well as the contents of the attachments. Its encryption is similar to the mailbox table:

```
CREATE COLUMNFAMILY "ec_2H2cQFqXA"."ec_DrDREi2KpqHQQS7Thn" (
       "ec_bjLfsNaK2UR"        blob,
       "ec_CSn8FjNEmYo91p"     blob,
       "ec_b5BYu5mBNks"        set<blob>,
       "ec_641fgm"             set<blob>,
       "ec_PBkbZY2"            set<blob>,
       "ec_CSn8FjMCrzubgW"     blob,
       "ec_3imKV8NX6Sebe"      blob,
       "ec_3bPTZnt1hGkwe"      blob,
       "ec_HdjsfVoe4K5khTprGo" map<blob, blob>,
       primary key ("ec_bjLfsNaK2UR")
);
```

To track the word counts in the folders with the keyword table, a counter is used. Since the keywords will only be relevant within the context of a folder for one user, both columns are used for the partitioning primary key:

```
CREATE COLUMNFAMILY mail.keyword (
       id_user    text,
       folder     text,
       keyword    text,
       cnt        counter,
       PRIMARY KEY ((id_user, folder), keyword)
);
```

The encryption of the DDL statement is similar to the mailbox and mail tables. The main difference is that an additional column (named id_mutation) is added, in order to allow for multiple rows per counter value to exist. This is necessary for the implementation of the PN-CRDT [39], as discussed in Subsection 4.2.4.

```
CREATE COLUMNFAMILY "ec_2H2cQFqXA"."ec_2zNYf5kBh9GUv" (
       "ec_2CHPPqnwLkMR8"        blob,
       "ec_ErWsxYW9VXf"          blob,
       "ec_2DmDGsKaARDL8"        blob,
       "ec_AugGbnu"              blob,
       "ec_8UkpdgasveZKp5F2kf"   blob,
       PRIMARY KEY (("ec_2CHPPqnwLkMR8", "ec_ErWsxYW9VXf"),
              "ec_2DmDGsKaARDL8", "ec_8UkpdgasveZKp5F2kf")
);
```

### 5.1.2 Inserts

Receiving an email results in inserts into the mailcontent and mailbox tables. The insert for the mailcontent table looks as follows:

```
INSERT INTO mail.mailcontent
(id_msg, msg_date, msg_to, msg_from, subject, msg_txt, attachments)
VALUES
('282214..',
       '2002-01-25 07:51:31-0800',
       {'sally.beck@enron.com'},
       'peggy.hedstrom@enron.com',
       'Trip to Houston',
       'I am currently scheduled...',
       {'schedule.pdf':0x3F..}
);
```

Encrypting this insert is straight-forward and results in the following query:

```
INSERT INTO "ec_2H2cQFqXA"."ec_DrDREi2KpqHQQS7Thn"
("ec_bjLfsNaK2UR", "ec_CSn8FjNEmYo91p", "ec_b5BYu5mBNks", "ec_CSn8FjMCrzubgW",
       "ec_3imKV8NX6Sebe", "ec_3bPTZnt1hGkwe", "ec_HdjsfVoe4K5khTprGo")
VALUES
(0x9DFAF50A5CBA9C7435FC,
       0x9D1390B65A40DE50314F,
       {0xDA2AB871C2964B71B61061855A00FDE9228C1E8807E8},
       0x505A29F39B28C98FB955F0547787FA823D86AD93C173556CD993,
       0xF77356302D926D3AB366896E9CC0CF29A5,
       0xC2CC8CB2435DA5A07DCAE8BB2817C076FBF9AF814C8044BFFC67354658,
       {0x422C1A3C7A4A001B49203295D355:0xCE..}
);
```

The insert into the mailbox table is more involved however, since the index for the msg_from column, which uses EncryptedCassandra's secondary indices, needs to be updated. Because of that, the encryption of the insert into the mailbox table is being skipped here and will instead be presented in Subsection 5.1.7, which focuses on secondary indices.

Updating the word counts in the keyword table involves counters, which will be presented in Subsection 5.1.8. As such, similarly to the mailbox table, inserting data into the keyword table is being skipped here.

### 5.1.3 Updates

Users can save drafts of emails they are writing in a draft folder and edit them at later points in time. These edits result in update statements on the mailcontent table. E.g. user hedstrom-p could have been editing the email to user beck-s (shown in Subsection 5.1.2), which would have resulted in this update statement for the final edit:

```
UPDATE mail.mailcontent SET
       msg_date='2002-01-24 17:44:21-0800',
       msg_to={'sally.beck@enron.com'},
       msg_txt='I am currently scheduled...',
       subject='Trip to Houston',
       attachments={'schedule.pdf':0x3F..}
WHERE id_msg = '253864..';
```

After encryption, the statement looks as follows:

```
UPDATE "ec_2H2cQFqXA"."ec_DrDREi2KpqHQQS7Thn" SET
       "ec_CSn8FjNEmYo91p"=0x5672FEA7A7987DEA2241,
       "ec_b5BYu5mBNks"={0x34C3278E1F8942D202468C5F5BD620C5493924700242},
       "ec_3bPTZnt1hGkwe"=0x45CDB..,
       "ec_3imKV8NX6Sebe"=0x90F66FAADFD0E236A121787068770C48F7,
       "ec_HdjsfVoe4K5khTprGo"={0x878B5D4AE3FF3167DFD6EF1969A9:0x381479}
WHERE "ec_bjLfsNaK2UR" = 0x9DF7F4005BBA9C742329;
```

### 5.1.4 Batches

Cassandra does not allow inserting or updating multiple rows within only one insert or update statement. It does however allow wrapping multiple I/U/D queries within one batch statement. If a user were to mark multiple emails as read, a batch like this could be used to issue all updates within one CQL statement:

```
BEGIN BATCH
        UPDATE mail.mailbox SET is_unread='N'
        WHERE id_user='beck-s'
                AND folder='inbox' AND msg_date='2002-...' AND id_msg='82214..'

        UPDATE mail.mailbox SET is_unread='N'
        WHERE id_user='beck-s'
                AND folder='inbox' AND msg_date='2002-...' AND id_msg='82215..'

        ...
APPLY BATCH;
```

After encryption the statement looks as follows:

```
BEGIN BATCH
        UPDATE "ec_2H2cQFqXA"."ec_2xtFR7kck5rYo"
        SET "ec_A7eewUi23ZHWA8u"=0x12F8B9
        WHERE "ec_UCPmhC9bSYzF"=0x59BB7D30CEC938FA
                AND "ec_8LZAX7GTQ8R"=0x4E0F4832D82E14
                AND "ec_2qjpdKjMfjUBFq"=8711..
                AND "ec_7ANvaayjA7u"=0xA83139F6106EEF69E2

        UPDATE "ec_2H2cQFqXA"."ec_2xtFR7kck5rYo"
        SET "ec_A7eewUi23ZHWA8u"=0x0C6BC5
        WHERE "ec_UCPmhC9bSYzF"=0x59BB7D30CEC938FA
                AND "ec_8LZAX7GTQ8R"=0x4E0F4832D82E14
                AND "ec_2qjpdKjMfjUBFq"=8722..
                AND "ec_7ANvaayjA7u"=0xBE0DA5B41A8F2DE778

        ...
APPLY BATCH;
```

Note that no indexed columns are being written in the updates within the batch, so no additional statements to maintain any index data are necessary.

### 5.1.5 Selects

To load a user's ten most recent emails in their inbox, for display, a query like this can be used:

```
SELECT msg_from, subject, msg_date, is_unread
FROM mail.mailbox
WHERE id_user='beck-s' AND folder='inbox'
ORDER BY msg_date DESC
LIMIT 10;
```

After encryption the query looks as follows:

```
SELECT "ec_2qjpdKkP8Wwz7q",
        "ec_oMBuDZGbVh5X",
        "ec_PFN3g1o5gnSj47QESdEt7Cr" AS "ec_2qjpdKjMfjUBFq",
        "ec_A7eewUi23ZHWA8u",
        /* ec: */"ec_7ANvaayjA7u",
        /* ec: */"ec_8LZAX7GTQ8R",
        /* ec: */"ec_UCPmhC9bSYzF"
FROM "ec_2H2cQFqXA"."ec_2xtFR7kck5rYo"
WHERE "ec_UCPmhC9bSYzF"=0x59BB7D30CEC938FA AND "ec_8LZAX7GTQ8R"=0x4E0F4832D82E14
ORDER BY "ec_2qjpdKjMfjUBFq" DESC
LIMIT 10;
```

EncryptedCassandra adds the missing id_msg, folder and id_user columns to the select part of the query in order to be able to decrypt every row in the resultset. It also adds the column that contains the symmetrically encrypted value for msg_date, in order to avoid having to decrypt an OPE value and the performance penalties involved with that. Those three columns have been prefixed by the /* ec: */ comment in the encrypted query.

Sorting by the msg_date column in this query is only made possible by these two prerequisites:

- The msg_date column is OPE-encrypted.
  If the column were encrypted using a block cipher then a sorting of the resulting rows would not be possible, since it would require first loading all relevant rows onto the client, sorting them, and then discarding all rows that were not the ten most recent ones. While theoretically possible, this would not be practically viable for larger inboxes.

- The msg_date column is the leading part of the clustering primary key.
  If another column were to be the leading part of the clustering primary key, then the resultset would have to first be sorted by that other column and could only then be sorted by msg_date additionally to that. It would also require that other column to be OPE encrypted (and thus sortable).

An example that utilizes secondary indices (that are maintained by Cassandra) would be selecting all emails prior to some date or within some specific timeframe for e.g. backup or archiving, by the mail service provider:

```
SELECT id_user, folder, msg_date, id_msg
FROM mail.mailbox
WHERE msg_date <= '2000-01-03' AND msg_date >= '2000-01-09';
```

Since Cassandra's native indices are being used, the query's encryption is straight forward:

```
SELECT "ec_UCPmhC9bSYzF", "ec_8LZAX7GTQ8R",
       "ec_PFN3g1o5gnSj47QESdEt7Cr" AS "ec_2qjpdKjMfjUBFq", "ec_7ANvaayjA7u"
FROM "ec_2H2cQFqXA"."ec_2xtFR7kck5rYo"
WHERE "ec_2qjpdKjMfjUBFq" <= 871122906905674365134722849780987955898 97
       AND "ec_2qjpdKjMfjUBFq" >= 871122906930154927878373362305811025 02063;
```

### 5.1.6 Deletes

Continuing with the example used in Subsection 5.1.5, if the selected emails need to be deleted from the inbox after archiving, doing so is not be immediately possible (i.e. using the same WHERE clause), since Cassandra only permits the use of secondary indices (in WHERE clauses) in select queries. However once the relevant messages have been determined, they can be deleted individually:

```
DELETE FROM mail.mailcontent WHERE id_msg='82214..';
```

After encryption the statement looks as follows:

```
DELETE FROM "ec_2H2cQFqXA"."ec_DrDREi2KpqHQQS7Thn"
WHERE "ec_bjLfsNaK2UR"=0x97F0F50959A09C2782;
```

### 5.1.7 Secondary Indices

The query to insert the email of Subsection 5.1.2 into the mailbox table is the following one:

```
INSERT INTO mail.mailbox
(id_user, folder, msg_date, msg_to, msg_from, subject, is_unread, id_msg)
values
('beck-s', 'inbox', '2002-01-25 07:51:31-0800', {'sally.beck@enron.com'},
       'peggy.hedstrom@enron.com', 'Trip to Houston', 'Y', '282214..');
```

However to encrypt the query EncryptedCassandra needs to first check if there already exists a record with an identical primary key in the mailbox table, in order to determine if the index for the mailbox table needs to be cleaned up when this query is executed. To do that, the following query will be used:

```
SELECT msg_from FROM mail.mailbox
WHERE id_user='beck-s' AND folder='inbox'  AND msg_date='2002-01-25 07:51:31-0800';
```

Afterwards, the original insert will be encrypted, and the queries necessary for the index maintenance will be generated. All of them will be wrapped together in one batch statement:

```
BEGIN BATCH

       INSERT INTO "ec_2H2cQFqXA"."ec_2xtFR7kck5rYo"
       ("ec_UCPmhC9bSYzF", "ec_8LZAX7GTQ8R", "ec_2qjpdKjMfjUBFq", "ec_6Un9PZEADKi",
              "ec_2qjpdKkP8Wwz7q", "ec_oMBuDZGbVh5X", "ec_A7eewUi23ZHWA8u",
              "ec_7ANvaayjA7u", "ec_PFN3g1o5gnSj47QESdEt7Cr" )
       VALUES
       (0x59BB7D30CEC938FA, 0x4E0F4832D82E14,
              87112290699830355801243595007212888859812,
              {0xA77455FF4439DA2046F94271DBFACF676058D7F241B0},
              0xFD5DE765FD6412D0A7138B31334D89FEEB4968778E9190F87981,
              0x557FA95D7081267E0AE6215E3BB4B65D56, 0xDE1CA4, 0xB73D3F3D1B49F0845909,
              0x929C3D198C2013D2C24F)

       INSERT INTO "ec_2H2cQFqXA"."ec_DBiRQqQZg9EonwLn6R"
       ("ec_4JBj7iUsVSNRc9pPxM", "ec_WEYh2uYhteJG", "ec_7nbEMwwGp7f",
              "ec_2zrrDUaPd8JXJf", "ec_7d9YHjxX82s" )
       VALUES
       (0x5358F0CFE7AC0BBE11260793AD7FD8CEB516A37E9C380612CF66, 0x1082971A35C1492B,
              0x5F7E0BB2EA900C, 0xC16BB7EFE1F6146729C2, 0x8A499C1A49FF37C3B1E8 )

APPLY BATCH;
```

The first insert query in the batch statement is the encrypted version of the original insert query. The second insert is an insert into the ei_mbx_from index table. Unencrypted the query looks as follows:

```
INSERT INTO mail.ei_mbx_from
(index_value, id_user, folder, msg_date, id_msg)
VALUES
('peggy.hedstrom@enron.com', 'beck-s', 'inbox', '2002-01-25 07:51:31-0800',
       '282214..');
```

If there already existed a row with that key in the mailbox table, then there would also be an additional delete query in the batch statement to first remove the old index entry. Unencrypted that query would be as follows (the encrypted variant is being omitted for brevity):

```
DELETE FROM mail.ei_mbx_from
WHERE index_value='peggy.hedstrom@enron.com'
       AND id_user='beck-s'
       AND folder='inbox'
       AND msg_date='2002-01-25 07:51:31-0800'
       AND id_msg='282214..';
```

If the index were a ticketed one, then EncryptedCassandra would also need to obtain a ticket number before generating the batch statement. It would first determine the current ticket number using this query (encrypted variant again omitted for brevity):

```
SELECT last_ticket_nr
FROM mail.ec_index_ticket
WHERE cf='mailbox' AND col='msg_from';
```

And then try to obtain the next ticket number for the batch statement, using this update:

```
UPDATE mail.ec_index_ticket
SET last_ticket_nr = 1
WHERE cf='mailbox' AND col='msg_from'
IF last_ticket_nr = 0;
```

The if clause causes a Lightweight Transaction to be used for the update, ensuring that either no other client obtains the same ticket number in the meantime or that this update query fails. If the query fails, then EncryptedCassandra repeats the process by trying to obtain another ticket number. After encryption, the update looks as follows:

```
UPDATE "ec_2H2cQFqXA"."ec_2NiFhZMRh6Zu1nHR3MhfVWmo"
SET "ec_Jjx2bJjAKPg69o3JWzsLfj" = 0xAC47FC03D42C
WHERE "ec_4Hmogz"=0x4E82AFC1726CD13AC2 AND "ec_FWo6cY7"=0x13E60A4F44FD582A4006
IF "ec_Jjx2bJjAKPg69o3JWzsLfj" = 0xAC47FC02186E;
```

The obtained ticket number is then used in the batch statement for setting the statement's writetime attribute. Syntactically, the timestamp is added to the statement right after BEGIN BATCH:

```
BEGIN BATCH USING TIMESTAMP 1
```

For deletes the process is similar, except that there will only be deletes performed on the index table and no inserts. Everything else (including the ticket details) remains unchanged.

When querying the indexed attribute, EncryptedCassandra implicitly performs the index lookup and then rewrites the select query, such that it only uses the primary key in the where clause. It also splits the select up into multiple queries, if multiple rows match the queried-for index value.
E.g. for this select query using the indexed msg_from column:

```
SELECT * FROM mail.mailbox WHERE msg_from = 'peggy.hedstrom@enron.com'
```

EncryptedCassandra performs this index lookup:

```
SELECT id_user, folder, msg_date, id_msg
FROM mail.ei_mbx_from
WHERE msg_from = 'peggy.hedstrom@enron.com';
```

Then the original query is rewritten into these queries, based on the results of the index lookup:

```
SELECT * FROM mail.mailbox
WHERE id_user = 'beck-s'
       AND folder='inbox'
       AND msg_date='2002-01-25 07:51:31-0800'
       AND id_msg='282214..';

SELECT * FROM mail.mailbox
WHERE id_user = 'beck-s'
       AND folder='inbox'
       AND msg_date='2001-12-07 10:06:42-0800'
       AND id_msg='161598..';
...
```

Finally, the select queries are encrypted prior to being executed:

```
SELECT * FROM "ec_2H2cQFqXA"."ec_2xtFR7kck5rYo"
WHERE "ec_UCPmhC9bSYzF" = 0x59BB7D30CEC938FA
      AND "ec_8LZAX7GTQ8R"=0x4E0F4832D82E14
      AND "ec_2qjpdKjMfjUBFq"=87112290699830355801243595007212888859812
      AND "ec_7ANvaayjA7u"=0xB73D3F3D1B49F0845909;

SELECT * FROM "ec_2H2cQFqXA"."ec_2xtFR7kck5rYo"
WHERE "ec_UCPmhC9bSYzF" = 0x59BB7D30CEC938FA
      AND "ec_8LZAX7GTQ8R"=0x4E0F4832D82E14
      AND "ec_2qjpdKjMfjUBFq"=87112290679875860301366127491595754549089
      AND "ec_7ANvaayjA7u"=0xC7D00A6A50EAD98223BE;
...
```

### 5.1.8 Counters

Counter columns cannot be directly encrypted. Instead EncryptedCassandra implements a counter of its own atop CQL. The specifics of the counter implementation were presented in Subsection 4.2.4 and the DDL statement for the creation of the counter table used here was shown in Subsecion 5.1.1.

Internally (i.e. unencrypted) a counter column of EncryptedCassandra is simply another 64-bit integer column. When an update like the following one increases (or decreases) a counter value:

```
UPDATE mail.keyword SET cnt = cnt + 1
WHERE id_user='beck-s' AND folder='inbox' AND keyword='trip';
```

EncryptedCassandra first rewrites the query into an insert:

```
INSERT INTO mail.keyword (id_user, folder, keyword, id_mutation, cnt)
VALUES ('beck-s', 'inbox', 'trip', a7ce.., 1);
```

The id_mutation column was implicitly added during the encryption of the DDL statement that created the table, in order to allow for multiple rows per counter to exist. A generated random value (a Universally Unique Identifier) will be used for it when new rows are inserted. The value for the counter column is simply the amount by which the counter is being increased or decreased by, in the user's original query. Afterwards the insert is encrypted:

```
INSERT INTO "ec_2H2cQFqXA"."ec_2zNYf5kBh9GUv"
("ec_2CHPPqnwLkMR8", "ec_ErWsxYW9VXf", "ec_2DmDGsKaARDL8",
      "ec_8UkpdgasveZKp5F2kf", "ec_AugGbnu")
VALUES
(0x4A5FDA12E6E6D878, 0xA2E36BD9098DBB, 0xACAF221C0677,
      0x341921572F6A229E9BD1E63F5E45C0190896, 0xFC518A95DF5AF7786357);
```

To retrieve counter values a conventional select query can be used:

```
SELECT cnt FROM mail.keyword
WHERE id_user='beck-s' AND folder='inbox' AND keyword='trip';
```

This statement will be encrypted into the following one:

```
SELECT "ec_AugGbnu",
      /* ec: */"ec_ErWsxYW9VXf" ,
      /* ec: */"ec_2CHPPqnwLkMR8" ,
      /* ec: */"ec_8UkpdgasveZKp5F2kf",
      /* ec: */"ec_2DmDGsKaARDL8"
FROM "ec_2H2cQFqXA"."ec_2zNYf5kBh9GUv"
WHERE "ec_2CHPPqnwLkMR8"=0x4A5FDA12E6E6D878
```

```
            AND "ec_ErWsxYW9VXf"=0xA2E36BD9098DBB
            AND "ec_2DmDGsKaARDL8"=0xACAF221C0677
```

This means that selects on counters return multiple individual values that need to first be added together, in order to calculate the actual value of the counter. EncryptedCassandra does this internally and then restructures the resultset accordingly, so that the process remains transparent to the user. The selection of the other 4 primary key columns is necessary in order to be able to decrypt the counter's individual values.

### 5.1.9 Accessing Cassandra and Using EncryptedCassandra

#### *5.1.9.1 API*

Accessing Cassandra through the Datastax Java Driver can be done as follows:

```
session = cluster.connect();
List<Row> rows = session.execute("SELECT * FROM ..").all();
for(Row r:rows) {
     int i = r.getInt(1);
}
```

In order to use EncryptedCassandra, the aforementioned Code would need to be adapted as follows:

```
session = cluster.connect();
session = EncryptedCassandra.encryptSession(session,
                    EncryptedCassandra.getDefaultCryptoProvider(),
                    "config.txt");

List<Row> rows = session.execute("SELECT * FROM ..").all();
for(Row r:rows) {
     int i = r.getInt(1);
}
```

The API of EncryptedCassandra is implemented around Cassandra's own API by using the decorator Pattern. The EncryptedCassandra class itself provides further functions in addition to encryptSession (like encryptQuery, decryptResultSet and decryptRow) to allow for more fine-grained access to the encryption. However the use of these individual functions is not necessary, since once the session object is wrapped, users can fully use it to perform all their operations on the database while encryption and decryption steps transparently occur wherever necessary.

While the session object is fully wrapped by EncryptedCassandra, some limitations apply:

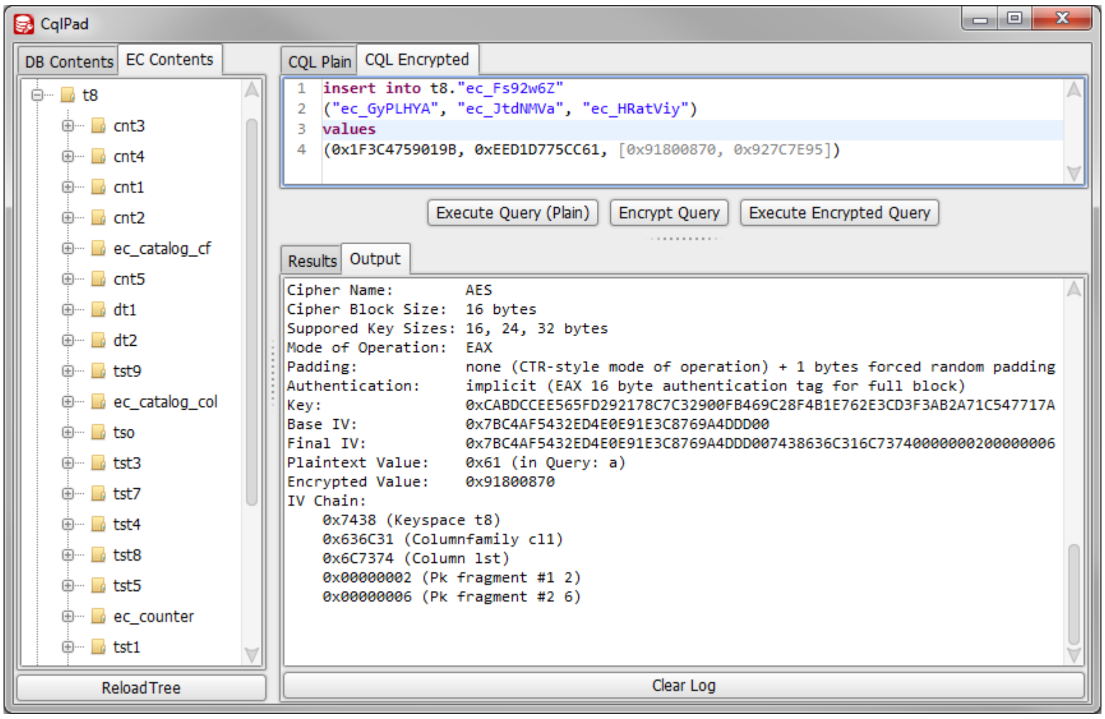
In addition to using the EncryptedCassandra class to wrap the session, users only need to provide a config file containing

- the base key for the encryption
- the base inititalization vector for the encryption
- the tag length to be used for authentication codes, if the scheme used allows for variable-sized ones

Users that want to change the encryption primitives used will also need to provide an implementation of the CryptoProvider API. This interface contains a total of 24 methods that implement details like encrypting and decrypting individual blobs of data using symmetric block ciphers, encrypting and decrypting individual numbers using an OPE algorithm, encrypting and decrypting identifiers and chaining keys.

### 5.1.9.2 GUI: CQLPad

CQLPad is a simple GUI application that allows for browsing Cassandra databases, executing queries, as well as illustrating the functionality of EncryptedCassandra. It is not strictly part of Encrypted-Cassandra, however it is one of few graphical tools currently providing access to Cassandra instances while also being a standalone application.



```
insert into t8."ec_Fs92w6Z"
("ec_GyPLHYA", "ec_JtdNMVa", "ec_HRatViy")
values
(0x1F3C4759019B, 0xEED1D775CC61, [0x91800870, 0x927C7E95])
```

```
Cipher Name:         AES
Cipher Block Size:   16 bytes
Suppored Key Sizes:  16, 24, 32 bytes
Mode of Operation:   EAX
Padding:             none (CTR-style mode of operation) + 1 bytes forced random padding
Authentication:      implicit (EAX 16 byte authentication tag for full block)
Key:                 0xCABDCCEE565FD292178C7C32900FB469C28F4B1E762E3CD3F3AB2A71C547717A
Base IV:             0x7BC4AF5432ED4E0E91E3C8769A4DDD00
Final IV:            0x7BC4AF5432ED4E0E91E3C8769A4DDD007438636C316C7374000000020000006
Plaintext Value:     0x61 (in Query: a)
Encrypted Value:     0x91800870
IV Chain:
    0x7438 (Keyspace t8)
    0x636C31 (Columnfamily cl1)
    0x6C7374 (Column lst)
    0x00000002 (Pk fragment #1 2)
    0x00000006 (Pk fragment #2 6)
```

**Figure 6:** CQLPad

As shown in the screenshot in Figure 6, in addition to providing a preview of how CQL queries look like after encryption, CQLPad also has an explain feature that provides further details on how the individual ciphertexts of the encrypted values were constructed.

CQLPad was created in an attempt to make EncryptedCassandra more tangible as a product as well as to aid in its development and debugging. The functionality as well as the name for the tool were inspired by Microsoft's XAMLPad[17].

## 5.2 Benchmarks

The benchmarks are based on the use case presented in Subsection 5.1 and use the Enron email dataset[18] (517.424 emails in 3499 folders, with a total of approximately 1,3 GB) as their test data. However for OPE benchmarks a reduced dataset of only 10.000 emails was used, in order to ensure reasonable benchmark runtimes, since only one database client was used, so that the OPE operations could not be spread out to multiple machines.

The benchmarks were run on two "CloudLevel 3[19]" virtual servers provided by DomainFactory[20]. The servers are both hosted in the same data center, located in Ismaning Germany (near Munich). The operating system used was the 64 bit version of Ubuntu 12.04. The JVM used was the 64 bit HotSpot server VM in JDK 1.7u79.

[17] https://msdn.microsoft.com/en-us/library/vstudio/ms742398%28v=vs.90%29.aspx
[18] https://www.cs.cmu.edu/~./enron/
[19] Each virtual server has 6 Intel Xeon E5-2620 (2 GHz) cores, 8 GB of RAM and uses conventional hard drives.
[20] https://www.df.eu/de/cloud-hosting/cloud-server/

Every benchmark was run four times. The purpose of the first run was to get the JVM into a steady state, where the hot code paths have been optimized[22, 23], in order to ensure that both the encrypted as well as the unencrypted code would benefit from any applicable optimizations, without the optimization process itself skewing the results. Of the remaining three runs the average was calculated and used as the result, in order to reduce the impact of factors that can't be fully controlled (like garbage collection and VM host system utilization).

Furthermore, every benchmark was run twice – once with client and server on the same machine (these benchmarks are labeled with "Local" in the charts), and once with client and server on different machines connected through a 100 Mb/s network link (these benchmarks are labeled with "Remote" in the charts). This was done to show the relative impact of the encryption when factoring in usual network overheads, as well as to show how network overheads impact EncryptedCassandra, when more involved features are used that issue multiple queries internally for one unencrypted user query.

The benchmark results were converted to milliseconds (of execution time) per query. As such, lower numbers mean better results. Additionally, the reported number of milliseconds is divided into client side and server side execution time (except for Benchmark 17[21]). Server side execution time is the time that passed between calling Cassandra's library to execute a query and the call returning. Everything else is part of the client side execution time.

Note that for queries that were batched together into one statement, the benchmark results report the execution time per query in the batch statement. E.g. a batch statement like the following one would count as 2 queries:

```
BEGIN BATCH
       UPDATE mail.mailbox SET is_unread='N' WHERE id_user='beck-s' AND ...
       UPDATE mail.mailbox SET is_unread='N' WHERE id_user='beck-s' AND ...
APPLY BATCH;
```

This, along with the conversion of all results into the time per query form, was done in order to be able to relate results of different benchmarks to one another. All benchmarks that use batched statements provide the exact number of batched queries they contain in their description.

[21] Benchmark 17 uses total time instead of client side time, since there was no proper way to clearly show the total time in it otherwise, due to it being represented in a line chart, instead of a bar chart like all other benchmarks.

### 5.2.1 Inserts

**Benchmark 1**: Populating the mailbox table using one insert statement at a time per email, without the msg_date and msg_from columns being indexed and without the msg_date column using OPE.

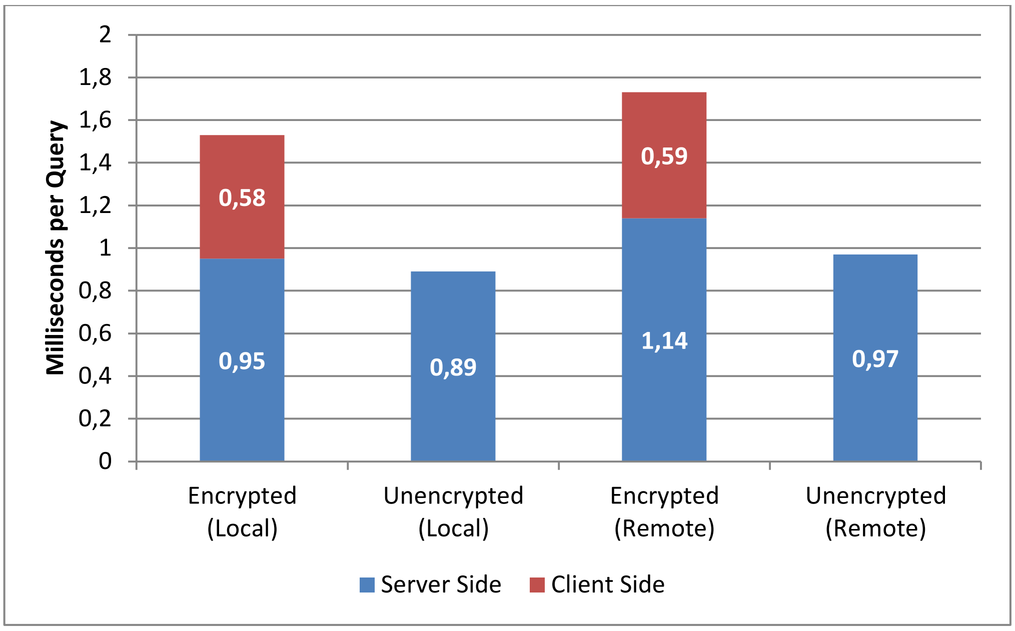


**Figure 7:** Results of Benchmark 1

Benchmark 1 shows the relative impact of the encryption to be between 60% to 65% of the original query runtime. Most of the added execution time (90%) is however spent within the client rewriting and enrypting the query, and only a small part of it (10%) is spent inside the database. The time spent inside the database is larger, since the encrypted ciphertexts are larger and incompressible compared to their plain text counterparts.

**Benchmark 2**: Populating the mailbox table using one insert statement at a time per email, without the msg_date and msg_from columns being indexed and with the msg_date column using OPE.

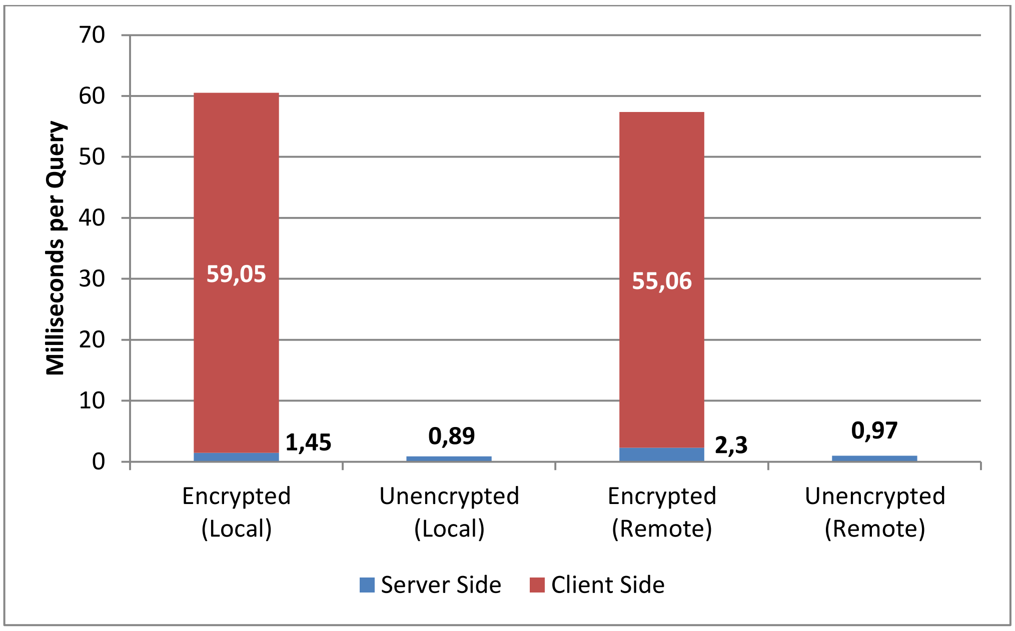


**Figure 8:** Results of Benchmark 2

Benchmark 2 shows the impact of using OPE - basically, the OPE operations completely dominate the result. Nevertheless, the encryption of one query is still fast enough for OPE to be viable for use in interactive scenarios, where users need to wait for queries to return from the database, as individual OPE operations complete in less than 100 milliseconds.

Additionally, Benchmark 2 also shows the time spent inside the database to increase by a much larger amount, compared to Benchmark 1. This is partly due to OPE ciphertexts being very large, relative to the plain text, and partly due to OPE columns being duplicated internally.

**Benchmark 3**: Populating the mailbox table using batched insert statements (100 per batch), without the msg_date and msg_from columns being indexed and without the msg_date column using OPE.

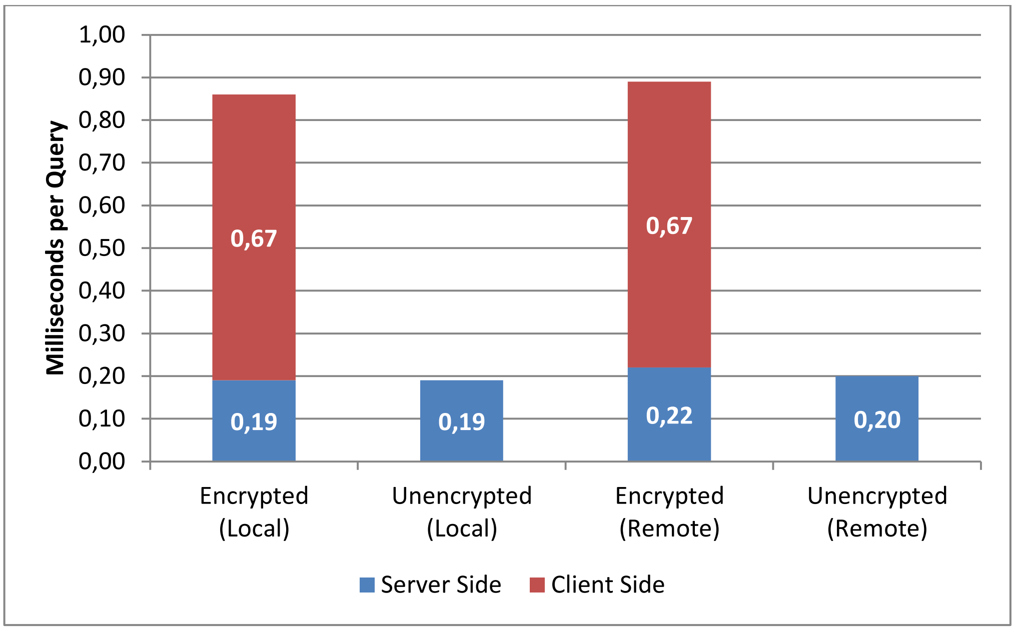


**Figure 9:** Results of Benchmark 3

Benchmark 3 shows how EncryptedCassandra's relative impact increases when using batched statements. However the actual time spent encrypting and rewriting one individual query still remains relatively similar to Benchmark 1, which uses non-batched inserts. The increase in encryption time per query (compared to Benchmark 1) is the added overhead necessary to split up the individual inserts (for rewriting and encryption) and then later on reassembling them into a batch statement again.

**Benchmark 4**: Populating the mailbox table using batched insert statements (100 per batch), without the msg_date and msg_from columns being indexed and with the msg_date column using OPE.

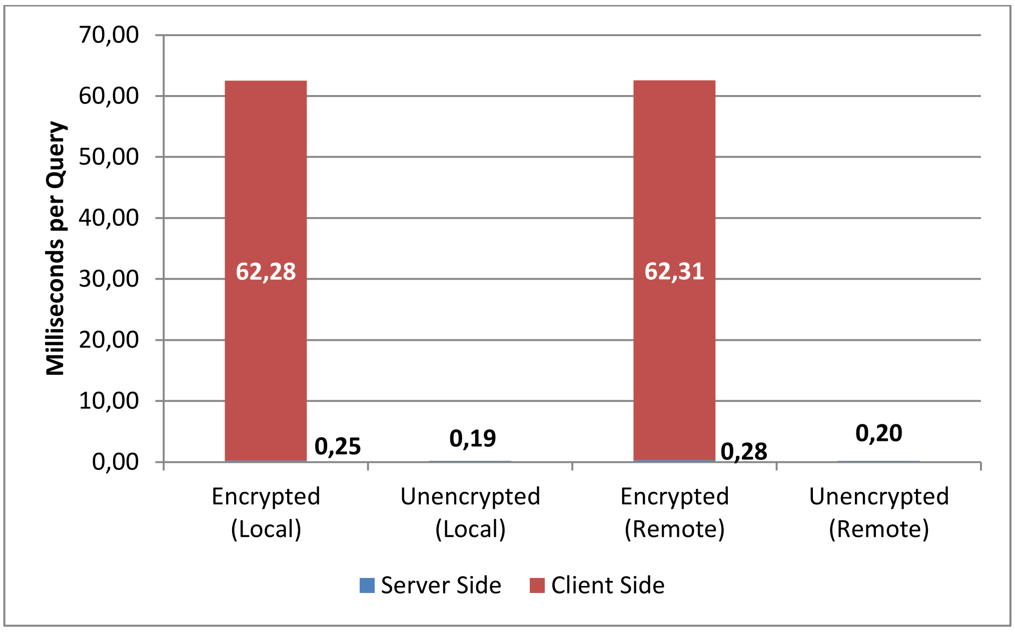


**Figure 10:** Results of Benchmark 4

Benchmark 4 shows a similar result to Benchmark 2, except more pronounced, since the time per query spent inside the database is even smaller, due to using batched statements.

### 5.2.2 Selects

**Benchmark 5**: Looking up rows in the mailbox table by primary key (at random), without the msg_date column using OPE.

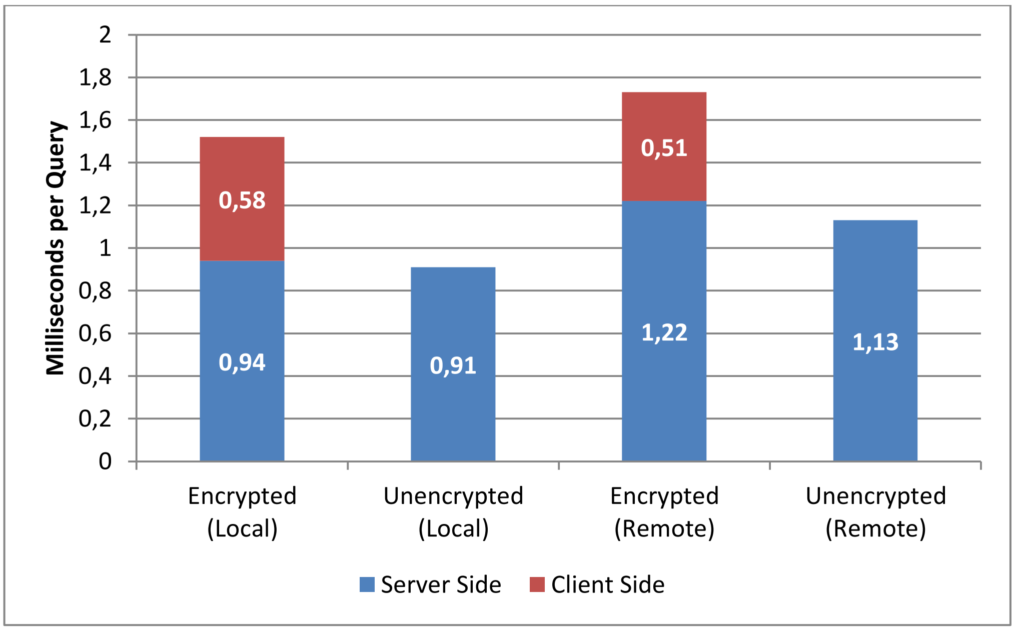


**Figure 11:** Results of Benchmark 5

Benchmark 5 shows the impact of EncryptedCassandra on select queries. The results are similar to Benchmark 1, except somewhat lower (relative to the original query) since Cassandra executes the selects of Benchmark 5 slower compared to the inserts of Benchmark 1 and decrypting the single-row resultsets barely impacts the execution time, compared to rewriting the query.

**Benchmark 6**: Looking up rows in the mailbox table by primary key (at random), with the msg_date column using OPE.

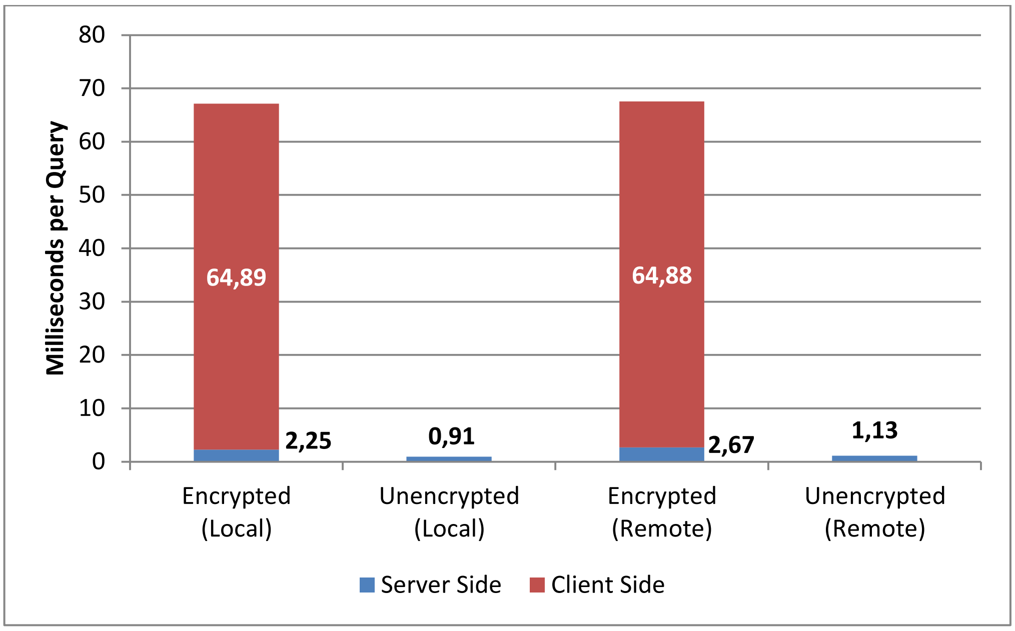


**Figure 12:** Results of Benchmark 6

Benchmark 6 shows a result similar to the ones of Benchmarks 2 and 4. The OPE operations completely dominate the runtime. OPE columns are not decrypted when the user selects them in a query, however they still impact the query in Benchmark 6 since the msg_date column is being queried in the WHERE clause, and thus its value needs to be OPE encrypted to be able to execute the query.

**Benchmark 7**: Selecting any 100 rows in the mailbox table by user and mailbox (at random).

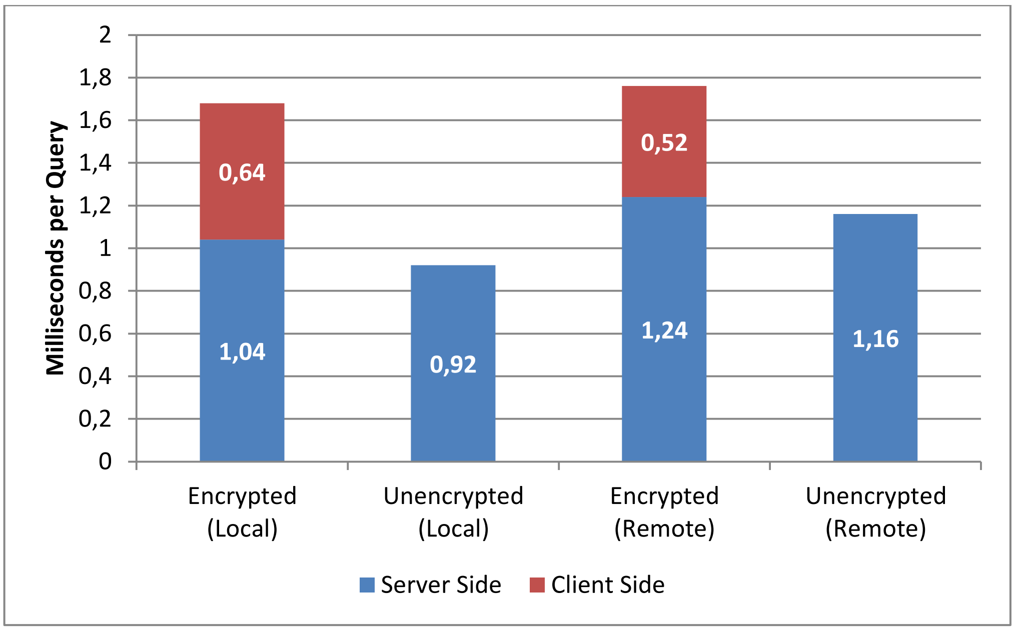


**Figure 13:** Results of Benchmark 7

Benchmark 7 shows the impact of EncryptedCassandra when needing to decrypt larger resultsets. The results are very similar to Benchmark 5, showing that decrypting larger resultsets barely impacts performance.

**Benchmark 8**: Selecting the 100 most recent rows in the mailbox table by user and mailbox, since the year 2000 using OPE.

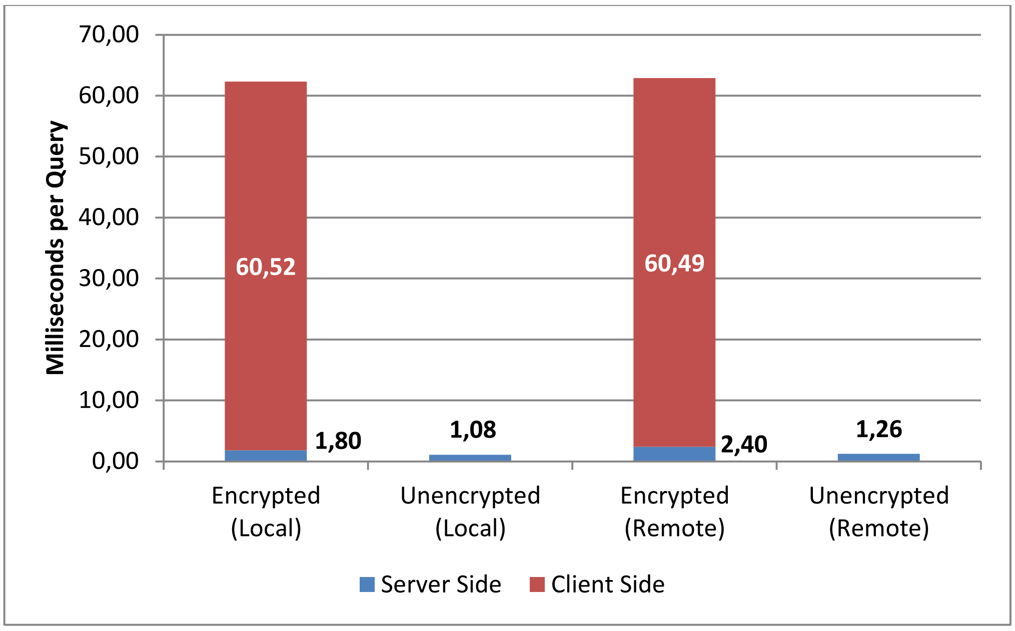


**Figure 14:** Results of Benchmark 8

Benchmark 8 shows a result similar to the other OPE benchmarks so far (Benchmark 2, 4 and 6), in that the OPE operations dominate the runtime. As such, the size of the selected resultsets that need to be decrypted barely impacts the result. Instead encrypting the OPE values used in the WHERE clause of the select queries is what requires the most time.

**Benchmark 9**: Selecting emails by user, folder and id_msg, using IN() on folder and id_msg.

The purpose of this benchmark is to show the performance impact of splitting up one query into multiple queries, as shown in Subsection 3.2.3.2. In order for the query to be valid the mailbox table was modified such that msg_date is no longer part of the primary key.

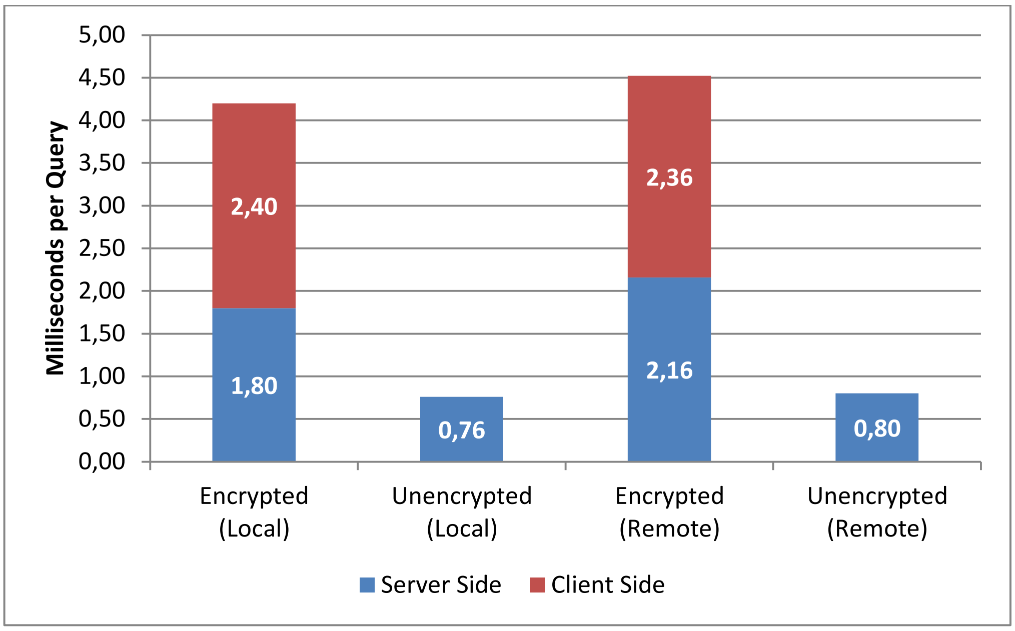


**Figure 15:** Results of Benchmark 9

Benchmark 9 shows a larger relative impact of EncryptedCassandra compared to the other non-OPE benchmarks. This is partly due to the rewriting itself being more involved (since the query needs to be split up) and partly due to multiple queries being issued to Cassandra in the background, in order to satisfy the one original user query.

### 5.2.3 Updates

**Benchmark 10**: Updating the is_unread column in the mailbox table, using one query per email.

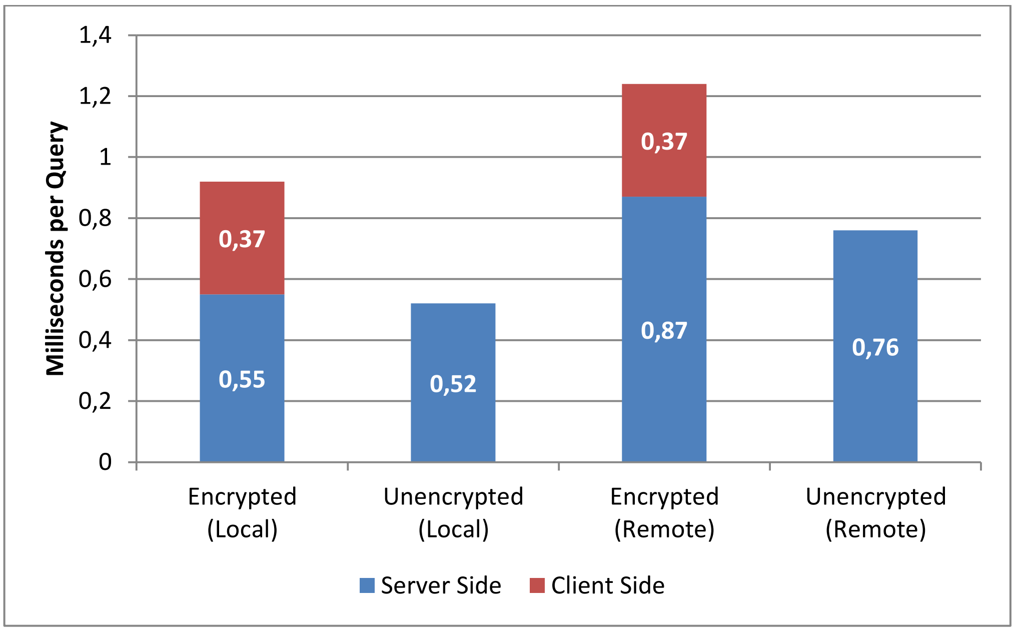


**Figure 16:** Results of Benchmark 10

Benchmark 10 shows a result similar to that of Benchmark 1. However EncryptedCassandra's relative impact is smaller when compared to Benchmark 1, since there are fewer values to rewrite in the query.

**Benchmark 11**: Updating the is_unread column in the mailbox table, using batches of 100 emails.

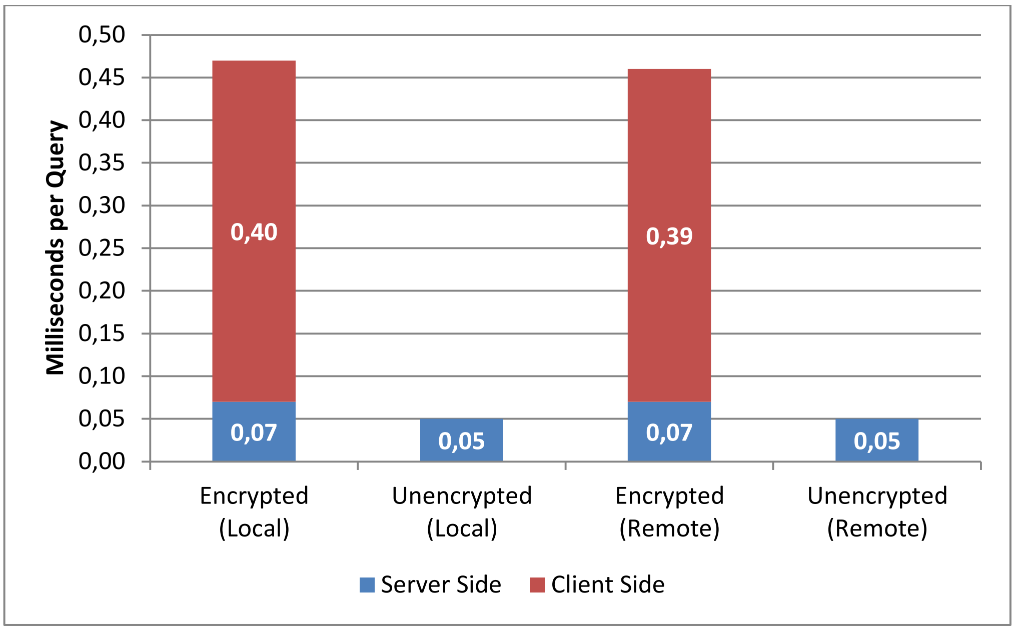


**Figure 17:** Results of Benchmark 11

Benchmark 11 shows a result similar to that of Benchmark 3. The time EncryptedCassandra spent rewriting and encrypting the query is less than in Benchmark 3, since there are fewer values to rewrite in the query, similar to Benchmark 10. However the effect is less pronounced than in Benchmark 10, due to the overhead of splitting up and reassembling the batch statement itself.

### 5.2.4 Deletes

**Benchmark 12**: Deleting emails by primary key (one email per query).

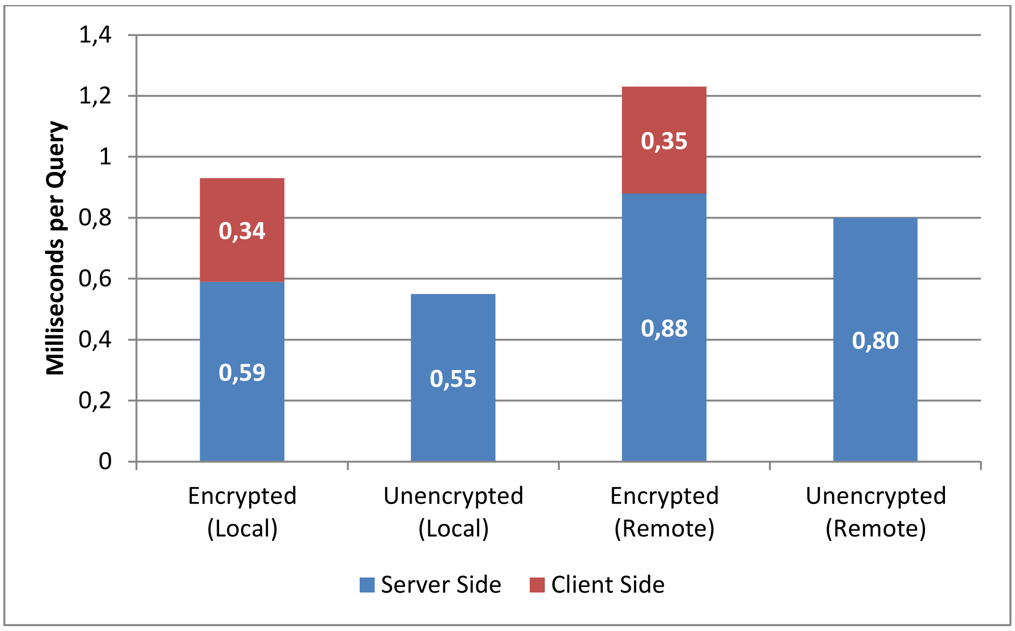


**Figure 18:** Results of Benchmark 12

Benchmark 12 shows the impact of the encryption for deletes by primary key. Since only the primary key columns and the table name need to be rewritten, the relative impact compared to the original query is fairly small.

**Benchmark 13**: Deleting emails by user and folder.

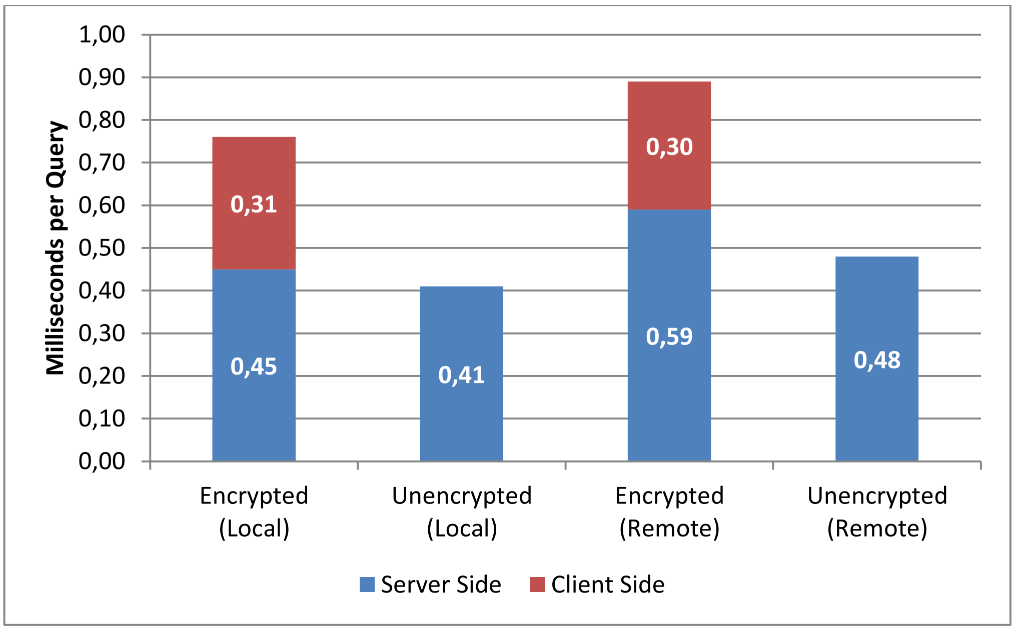


**Figure 19:** Results of Benchmark 13

Benchmark 13 shows the impact of the encryption for deletes by partitionioning primary key only (i.e. multiple rows being deleted by one query). Since only the partitioning primary key columns and the table name need to be rewritten, EncryptedCassandra can process the queries relatively fast compared to the other benchmarks so far, however the relative impact remains similar to Benchmark 12, since Cassandra itself is faster at processing the deletes too.

### 5.2.5 Secondary Indices

**Benchmark 13**: Populating the mailbox table using one insert statement at a time per email, with the msg_from column being indexed by Cassandra's indices in the unencrypted benchmark and EncryptedCassandra's indices (with and without the use of tickets) in the encrypted benchmark. The msg_date column remains unindexed and does not use OPE.

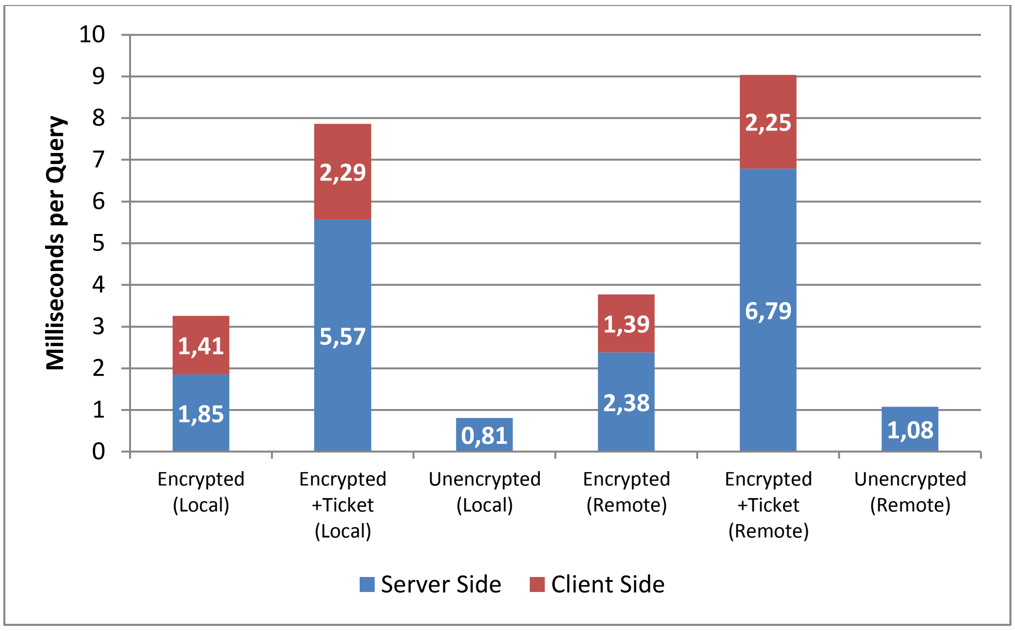


**Figure 20:** Results of Benchmark 13

Benchmark 13 shows the insert performance of EncryptedCassandra's indices compared to the native ones of Cassandra itself. Since multiple queries need to be issued when using EncryptedCassandra's indices, the time spend inside the database also significantly increases. This is especially true for ticketed indices, where the time required to obtain the ticket is in fact the largest part of the query execution time.

**Benchmark 14**: Looking up all emails by one specific sender (at random) in the mailbox table using Cassandra's indices in the unencrypted benchmark and EncryptedCassandra's indices in the encrypted benchmark.

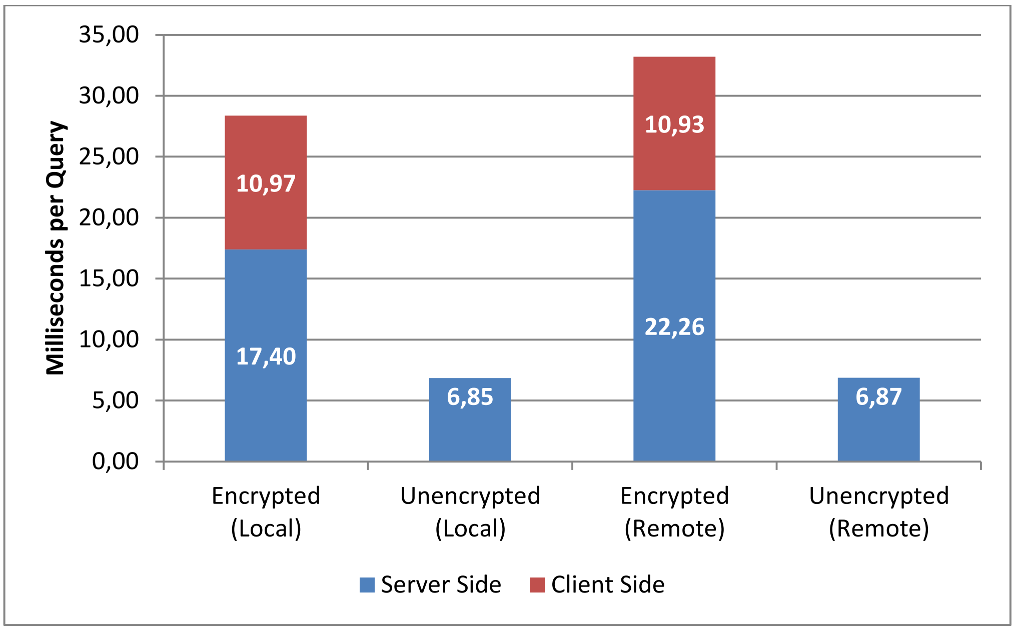


**Figure 21:** Results of Benchmark 14

Benchmark 14 shows the select performance of EncryptedCassandra's indices compared to the native ones of Cassandra itself. Since one unencrypted select query is split up into multiple individual selects, as the index reference is resolved, the time spent inside the database is significantly higher for the encrypted queries.

### 5.2.6 Counters

**Benchmark 15**: Populating the keyword table using one update statement at a time per keyword per email.

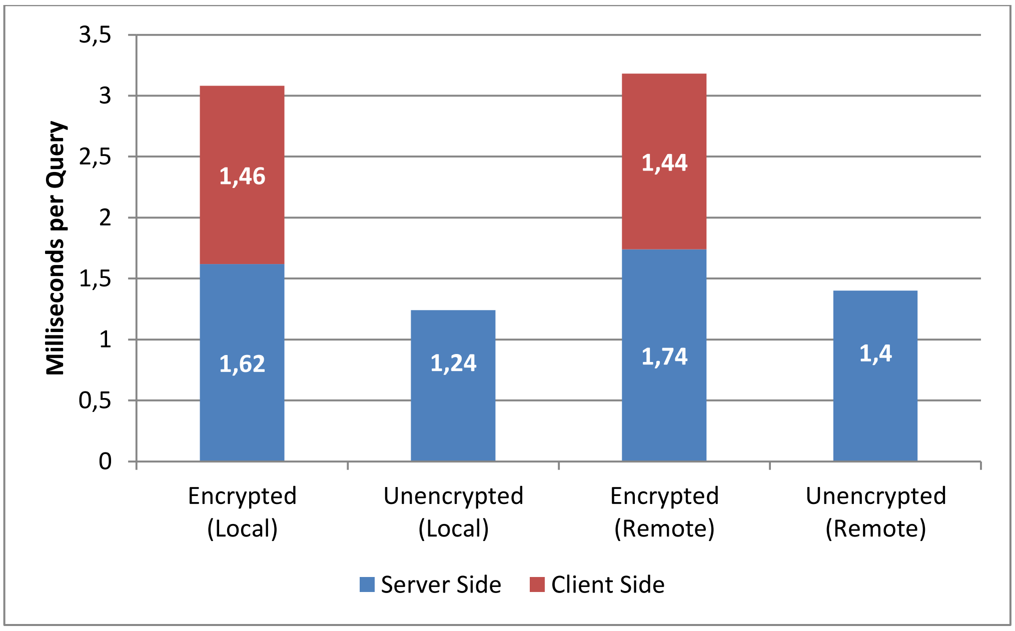


**Figure 22:** Results of Benchmark 15

Benchmark 15 shows EncryptedCassandra's impact for updates on counter columns. Since the original update query is first substituted by an insert that is then encrypted, the time spent rewriting and encrypting the query is higher than for more straight-forward inserts, which were shown in Benchmark 10.

**Benchmark 16**: Looking up the keyword counts by id_user, folder and keyword (at random).

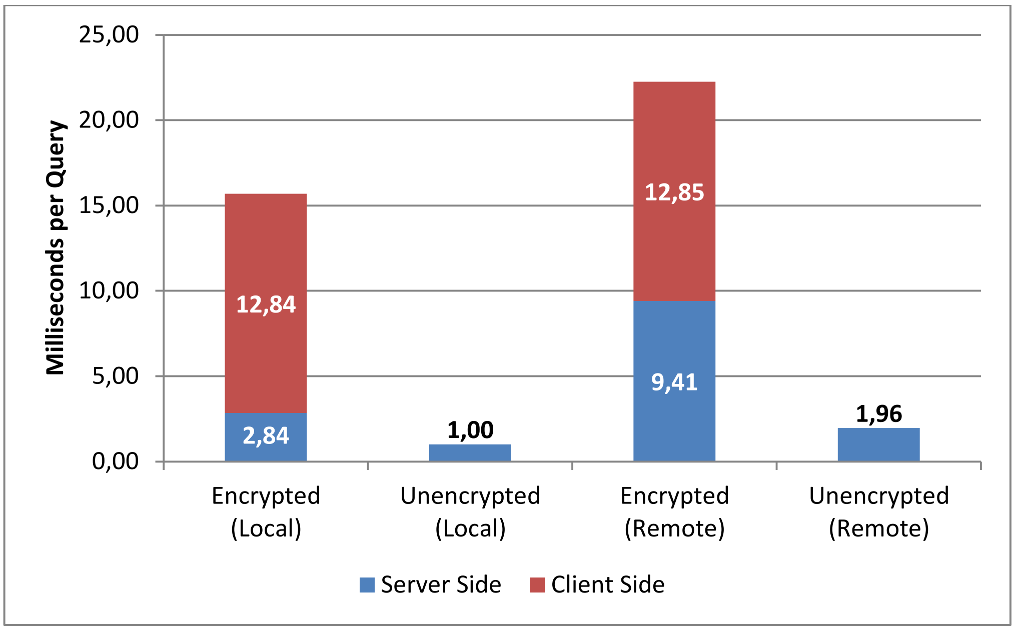


**Figure 23:** Results of Benchmark 16

Benchmark 16 shows EncryptedCassandra's impact to be relatively high, compared to the original query, when selecting counter columns, especially so in the networked benchmark. This is due to counters having very large resultsets, depending on how many updates the counter has received so far.

**Benchmark 17**: Select performance of EncryptedCassandra's counters, depending on the number of counter mutations stored. (This benchmark uses generated data and not the Enron dataset).

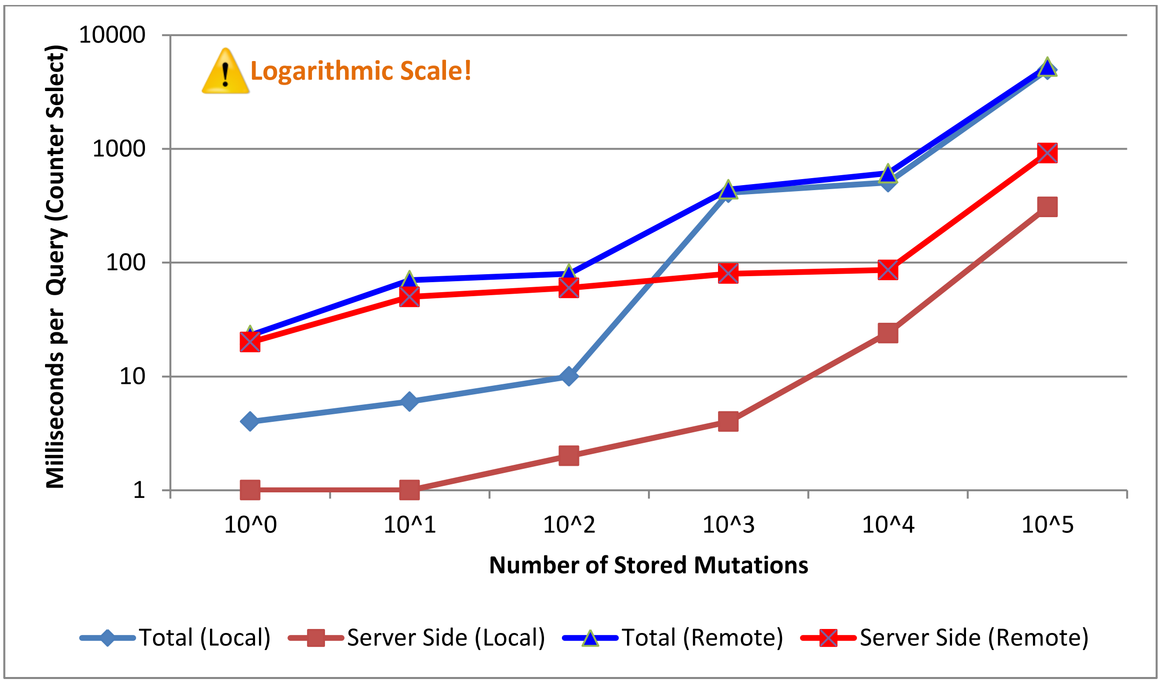


**Figure 24:** Results of Benchmark 17

Benchmark 17 provides another perspective on EncryptedCassandra's performance for selecting counter values, by showing how it evolves depending on the number of stored counter updates. Note that there is still a notable difference between local and remote queries even in the $10^5$ case (0,7ms or 6%), which does however not show up in the chart in Figure 24 due to the use of a logarithmic scale.

### 5.2.7 Discussion of Results and Further Insight

The overhead added by EncryptedCassandra is fairly manageable in straight-forward use cases. For conventional I/U/D and select queries EncryptedCassandra adds a constant 0,3 to 0,7 milliseconds of overhead, as shown in benchmarks 1, 3, 7, 10, and 11, depending on the amount of columns present in the query. The encryption overhead being mostly static means that on batched queries, as shown in Benchmarks 3 and 11 which utilize Cassandra's CQL interface as well as network interfaces more efficiently, EncryptedCassandra has overall a larger relative impact on the performance.

For single select queries that return multiple rows on the other hand (see Benchmark 9) the relative impact remains mostly unchanged, as the size of the resultset has barely any effect, even for larger resultsets with hundreds of rows. Further profiling of the code reveals that in fact very little time is spent inside actual crypto code (i.e. converting plain text into ciphertext and vice versa) and most of the added overhead is instead caused by the query rewriting process itself. A large part of this is due to the values (that are to be encrypted) needing to be first be deserialized prior to encryption and then needing to be re-serialized after encryption. Additionally the encryption process also increases the size of the values in the query.

The exception to this is OPE. If OPE values are used in a query, as was done in Benchmarks 2, 4, 6, and 8, either by being written through insert or update queries, or being queried in the WHERE clause of delete, update, or select queries, then they completely dwarf all other code parts in execution time. However queries that merely select OPE-encrypted columns are not affected by this, since EncryptedCassandra also stores a duplicate, which was encrypted using conventional block ciphers, of the value internally.

Most of the overhead caused by EncryptedCassandra is spent on the client side rewriting the queries, as shown in e.g. benchmarks 1 and 5. However a slight increase in server side execution time can still be noticed, since the queries become longer (due to the encryption increasing the size of the data values), as well as the data values in encrypted queries being incompressible (due to being encrypted). This affects the amount of I/O (on both the network as well as the disks) that needs to be performed to execute as well as parse the query by Cassandra.

With more involved queries, e.g. queries needing to be split up due the usage of IN() or queries manipulating or querying columns that are indexed by EncryptedCassandra's indices, as shown in Benchmarks 9, 13, and 14, the encryption overhead is significantly larger. Especially since these features perform more than just a rewriting of the unencrypted query into an encrypted one, and instead issue multiple queries to the database transparently, thus also increasing the time encrypted queries spend executing on the server side significantly.

One final performance edge case are EncryptedCassandra's counters, as shown in Benchmarks 16 and 17, where eventually the time required to decrypt the resultset completely dominates the result, since every individual counter update needs to be selected and decrypted in order to obtain the final value.

# 6 Conclusion

A solution for maintaining the confidentiality of data stored in cloud-hosted NoSQL databases was presented in this thesis and a sample implementation for Cassandra was provided. The goal of confidentiality was achieved by encrypting the data on the client side, thus precluding any possible privacy concerns through the usage of cloud services. Additionally, the encryption scheme was designed such that plain text data patterns do not leak through the encryption.

The tradeoff for this is some database functionality as well as performance needing to be sacrificed, in addition to using encryption primitives that are not yet accepted industry standards. Nevertheless, the final result is an encryption scheme that is practically viable.

Several other solutions, striking different tradeoffs and targeting different types of databases, were contrasted to EncryptedCassandra. Additionally, alternate algorithms and approaches for Encrypted-Cassandra itself were discussed. The overall theme here was that while there have been some great advances in both cryptography as well as NoSQL databases recently, there is still no perfect way of encrypting databases – i.e. without sacrificing something in the process.

Nevertheless, the recent advances in homomorphic cryptographic primitives, as well as ever more mature and flexible implementations of NoSQL databases, indicate that the situation is rapidly improving.

## References

[1] Apache Cassandra™ 2.1 Documentation [Online], 2015. Available: http://docs.datastax.com/en/cassandra/2.1/cassandra/gettingStartedCassandraIntro.html. Accessed: 2015-08-03.

[2] M. Armbrust et al., "A view of cloud computing" in Communications of the ACM 53.4 pp. 50-58, 2010.

[3] A. Arasu et al., "Orthogonal Security with Cipherbase," CIDR, 2013.

[4] S. Bajaj, R. Sion, "TrustedDB: A trusted hardware-based database with privacy and data confidentiality," in Knowledge and Data Engineering, IEEE Transactions on 26.3, pp. 752-765, 2014.

[5] "Banking -- Procedures for message encipherment (wholesale) -- Part 2: DEA algorithm," ISO 10126-2:1991, 1991.

[6] M. Bellare, et al., "The EAX mode of operation," in Fast Software Encryption, Springer Berlin Heidelberg, 2004.

[7] A. Boldyreva, et al. "Order-preserving symmetric encryption," in Advances in Cryptology-EUROCRYPT 2009. Springer Berlin Heidelberg pp. 224-241, 2009.

[8] A. Boldyreva et al., "Order-preserving encryption revisited: Improved security analysis and alternative solutions," in *Advances in Cryptology–CRYPTO 2011*. Springer Berlin Heidelberg, pp. 578-595, 2011.

[9] Z. Brakerski et al., "(Leveled) fully homomorphic encryption without bootstrapping," in ITCS, pp. 309–325, 2012.

[10] R. Cattell, "Scalable SQL and NoSQL data stores," in ACM SIGMOD Record 39.4 pp. 12-27, 2011.

[11] F. Chang et al., "Bigtable: A distributed storage system for structured data," in ACM Transactions on Computer Systems (TOCS) 26.2, 2008.

[12] CQL for Cassandra 2.x [Online], 2015. Available: http://docs.datastax.com/en/cql/3.1/cql/cql_intro_c.html. Accessed: 2015-08-03.

[13] G. DeCandia et al., "Dynamo: amazon's highly available key-value store," in ACM SIGOPS Operating Systems Review. Vol. 41. No. 6. ACM, 2007.

[14] D. Dinkwater. "Microsoft says NSA spying hit trust in the cloud," in SC Magazine UK [Online], 2014. Available: http://www.scmagazineuk.com/microsoft-says-nsa-spying-hit-trust-in-the-cloud/article/376564/. Accessed: 2015-08-03.

[15] J. Ellis, "Atomic batches in Cassandra 1.2," in Datastax Developer Blog [Online], 2012. Available: http://www.datastax.com/dev/blog/atomic-batches-in-cassandra-1-2. Accessed: 2015-08-03.

[16] J. Ellis, "CQL3 for Cassandra experts," in Datastax Developer Blog [Online], 2012. Available: http://www.datastax.com/dev/blog/cql3-for-cassandra-experts. Accessed: 2015-08-03.

[17] J. Ellis, "Lightweight transactions in Cassandra 2.0," in Datastax Developer Blog [Online], 2013. Available: http://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0. Accessed: 2015-08-03.

[18] S. Fau et al., "Towards practical program execution over fully homomorphic encryption schemes," in P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on. IEEE, 2013.

[19] J. Feigenbaum et al., "Cryptographic protection of databases and software," in Distributed Computing and cryptography 2 pp. 161-172, 1991.

[20] C. Gentry, "A fully homomorphic encryption scheme," Diss. Stanford University, 2009.

[21] C. Gentry, S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," in Advances in Cryptology–EUROCRYPT 2011, Springer Berlin Heidelberg, pp. 129-148, 2011.

[22] B. Goetz, "Java theory and practice: Anatomy of a flawed microbenchmark," in IBM Developerworks [Online], 2005. Available: https://www.ibm.com/developerworks/java/library/j-jtp02225/. Accessed: 2015-08-03.

[23] B. Goetz, "Java theory and practice: Dynamic compilation and performance measurement," in IBM Developerworks [Online], 2004. Available: http://www.ibm.com/developerworks/java/library/j-jtp12214/. Accessed: 2015-08-03.

[24] J. Gray, A. Reuter, "Transaction Processing: Concepts and Techniques," Morgan Kaufmann Publishers, 1993.

[25] H. Hacigümüş et al., "Executing SQL over encrypted data in the database-service-provider model," in Proceedings of the 2002 ACM SIGMOD international conference on Management of data, ACM, 2002.

[26] B. Kaliski, "PKCS #7: Cryptographic Message Syntax," RFC 2315, 1998.

[27] K. Kingsbury, "Call me maybe: Cassandra," in Aphyr [Online], 2013. Available: https://aphyr.com/posts/294-call-me-maybe-cassandra/. Accessed: 2015-08-03.

[28] L. Lamport, "Paxos made simple," in ACM Sigact News 32.4, pp. 18-25, 2001.

[29] S. Lesbrene, "Coming in 1.2: Collections support in CQL3," in Datastax Developer Blog [Online], 2012. Available: http://www.datastax.com/dev/blog/cql3_collections. Accessed: 2015-08-03.

[30] H. Lipmaa et al., "CTR-mode encryption," in First NIST Workshop on Modes of Operation. 2000.

[31] D. McCandless, "World's Biggest Data Breaches," in Information is Beautiful [Online], 2015. Available: http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/. Accessed 2015-06-15.

[32] P. Paillier, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes," in Advances in Cryptology-EUROCRYPT'99. Springer Berlin Heidelberg, 1999.

[33] T. J. Parr, R. W. Quong. "ANTLR: A predicated-LL(k) parser generator," in Software: Practice and Experience 25.7, pp. 789-810, 1995.

[34] R.A. Popa, et al., "An ideal-security protocol for order-preserving encoding," in Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013.

[35] R.A. Popa et al., "CryptDB: protecting confidentiality with encrypted query processing," in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. ACM, 2011.

[36] L. Richard, "The sweet spot for Cassandra secondary indexing," in Richard Low's blog [Online], 2013. Available: https://web.archive.org/web/20141109185152/http://www.wentnet.com/blog/?p=77. Accessed: 2015-08-03.

[37] D. Scherer, "Those Are Not Transactions (Cassandra 2.0)," in FoundationDB Blog [Online], 2013. Available: https://web.archive.org/web/20150325003231/http://blog.foundationdb.com/those-are-not-transactions-cassandra-2-0. Accessed: 2015-08-03.

[38] B. Schneier, "Applied Cryptography: Protocols, Algorithms, and Source Code in C," John Wiley & Sons, 2007.

[39] M. Sharpio et al, "A comprehensive study of Convergent and Commutative Replicated Data Types," 2011

[40] F.X. Standaert et al., "A unified framework for the analysis of side-channel key recovery attacks," in Advances in Cryptology-EUROCRYPT 2009. Springer Berlin Heidelberg, pp. 443-461, 2009

[41] G. Stewart, C. Reedjik, "Apache Cassandra at ING: Testing the Waters – Consistency Required!" in Cassandra Summit 2014 [Online], 2014. Available: http://www.slideshare.net/planetcassandra/cassandra-summit-2014-apache-cassandra-at-ing-testing-the-waters-consistency-required. Accessed: 2015-08-03.

[42] L. Xiao, I. Yen, "Security analysis for order preserving encryption schemes," in Information Sciences and Systems (CISS), 2012 46th Annual Conference on. IEEE, 2012.

[43] A. Yeschenko, "What's New in Cassandra 2.0: Prototype Triggers Support," in Datastax Developer Blog [Online], 2013. Available: http://www.datastax.com/dev/blog/whats-new-in-cassandra-2-0-prototype-triggers-support. Accessed: 2015-08-03.

## Source Code

If the disc is missing or has been damaged,
a new copy can be obtained at

http://www.qeleshi.at/ec2015