Design and implementation of a privacy-enhanced microblogging architecture

Master Thesis
Sandro Heinzelmann
May 17, 2013

Advisors: Prof. Dr. S. Capkun, Dr. C. Soriente
Department of Computer Science, ETH Zürich
Abstract

Microblogging services like Twitter allow for large-scale information sharing and retrieval. They are used by millions of people and have become a tremendously popular platform to share both inconsequential as well as sensitive information such as political views, habits or health conditions. These services are designed and geared towards allowing easy and straightforward sharing of content, but they often overlook the privacy of this information. While some services allow a somewhat granular access control to information shared, there remains the problem that the services providers themselves have unlimited access to the information of their user base. This information consists both of the actual content shared as well as the interests of users in topics and publishers. Users have to trust the service provider that they do not misuse this wealth of information. While they are usually bound by reputation to safeguard it, there are often clauses in user agreements that allow providers to mine user content and deliver targeted advertisements or to resell information to their partners. In previous work some of these issues have been addressed by encrypting the shared content as well as the interests in specific topics. This drastically limits the information gained by the provider, but it still retains the information of whom/what users are interested in; i.e. user relationships.

In this master thesis we try to circumvent these problems by designing and implementing a Twitter-like microblogging service that gains no information at all about user relationships. Towards this goal we use a technique called Private Information Retrieval (PIR) which enables a user to retrieve data from a database without the database host gaining any information about what he retrieved. We use a practical approach to PIR and design and implement a fully functional microblogging system which makes use of the provider’s cloud and is user-friendly by doing most of the client-side work with an appropriate browser extension. It is general enough such that it can be extended by other proposed systems for further privacy through encryption of the shared content. We evaluate our system in a real cloud, discuss the results and draw conclusions about the feasibility of using PIR in a scalable, dynamic system.
## Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vi</td>
</tr>
</tbody>
</table>

### 1 Introduction

1.1 Motivation ........................................... 1
1.2 Contributions ................................. 2
1.3 Related work .................................. 3
1.4 Organization ......................... 5

### 2 Preliminaries

2.1 Twitter .............................. 7
2.2 Private Information Retrieval ............... 8
   2.2.1 PIR based on homomorphic operations .... 9
2.3 MapReduce ......................... 12
   2.3.1 Hadoop .......................... 13

### 3 Design

3.1 Goals and assumptions .......................... 15
3.2 Private Information Retrieval ............... 16
   3.2.1 Our approach: PIRMAP ................. 17
3.3 Database .................................. 21
   3.3.1 Granularity ....................... 21
   3.3.2 Windowing ......................... 23
3.4 Batch queries ............................ 24
   3.4.1 Reducing communication ............ 24
   3.4.2 Reducing computation ............. 26
3.5 Database indexing ..................... 27
   3.5.1 Private index retrieval solutions .... 28
A Appendix 77
A.1 Evaluation of Paillier .......................... 77
A.2 Misc. thoughts on Hadoop configuration .......... 77

Bibliography 79

List of Figures

2.1 MapReduce phases ........................................ 13
3.1 PIRMAP file block scheme .......................... 18
3.2 PIRMAP file mapping and reduction ............... 19
3.3 Hypothetical view of offset batch queries ......... 25
3.4 Amortizing work with hashing ...................... 27
3.5 Per-tweeter database element indexing ............ 30
4.1 Implementation component overview ................. 38
4.2 Virtually combining window files .................. 41
4.3 Generic matrix PIR instantiations ................. 44
4.4 Client data flow example ............................ 49
5.1 Trivial PIR matching time comparison plot ........ 53
5.2 Tweet volume matching time plot ................... 54
5.3 Tweet file vs. matching output size plot ......... 55
5.4 Tweeter volume matching time plot ................. 56
5.5 Tweeter volume setup time plot .................... 57
5.6 Tweeter volume query size/time plots ............. 58
5.7 Subscriber volume matching time plot .............. 59
5.8 Subscription volume matching time plot ............ 60
5.9 Subscription volume query size/time plots ......... 61
5.10 Subscription volume hash bucket matching time plot .. 61
5.11 Subscription volume hash bucket primary PIR and recursive PIR plot .................................. 62
5.12 Subscriber volume output combine and decrypt time plot ...... 63
5.13 Asymmetric database matching time plot .......... 64
List of Tables

A.1 Paillier encryption runtime ........................................ 77
Chapter 1

Introduction

1.1 Motivation

Microblogging services like Twitter have become a tremendously popular way of sharing short messages among many people. They allow content sharing on a very large scale with millions of people sharing and retrieving short bits of information.

Their mission statement usually focuses on the easy, open and fast sharing of messages and it usually does not focus on privacy besides the usual aspects of website data privacy. In particular, the providers of such services have unlimited access to all information shared on their systems and often user agreements allow them to mine this vast amount of potentially personal data for advertising purposes [25] and even forward it to their partners. Additionally providers may be subject to warrants from law enforcement [20] which forces them to hand out confidential data. One can easily imagine that such a scenario would be devastating for oppressed minorities in certain countries. With the ever increasing prevalence of microblogging it is a good idea to discuss and find ways to improve the privacy of its users. Service providers like Twitter form a centralized handler of a massive amount of information. Such public organizations usually have a vested interest in keeping user trust and it may be reasonable to assume that they would protect this information. On the other hand it is unfortunately the case that the collection of data for advertisement or even less transparent purposes has become somewhat accepted among users. Additionally it is often the case that users are not aware of the implications of an entity having full knowledge of all content and what it can derive from this.

Previous work [15] has addressed the more obvious aspects of this conundrum by encrypting the content of messages and introducing more finely grained access control which allows users to decide who is permitted to view their messages. While this solves many of the major privacy concerns
it does not address the fact that the provider still knows of the relationships of users and their interests. The knowledge that someone is subscribed to and reads the messages of a known radical tweeter or Justin Bieber can be problematic even without that user explicitly stating anything in his own messages. Knowing all such relationships allows the provider to infer a relationship profile which can be very telling. A large fraction of the user base of Twitter does not share many own messages [38] and instead uses Twitter to follow people they are interested in. They may believe that this lack of active information sharing prevents disclosure of any useful information to the service provider. On the contrary, it may gain a large amount of information about user behavior and interest by simply analyzing the relationships of a user. Not only can it build an extensive relationship graph, but it may infer information by proxy, so to speak, by analyzing the messages of the people someone follows and hence reads. This may expose critical information about someone’s political, sexual or medical interests without that person ever actively sending his own messages on these topics. The fact that Twitter allows any user to look at this relationship information of someone else does not help either, though it is more easily avoidable, of course.

If we can prevent this gain of information about user relationships for all involved entities, not only do we address the issue just mentioned, we also address some of the concerns that are handled by work like [15]. If the provider does not even know who is following whom, then the content of the messages is irrelevant. Only the actual poster of the message is potentially interested in keeping their content private; anyone who subscribed to that poster is protected because the provider is not aware that he is reading those messages.

1.2 Contributions

In this thesis we design, implement and evaluate a microblogging system in the vein of Twitter where the provider does not gain any information about user relationships. Our microblogging provider still functions as the centralized handler of all information and is responsible for the delivery of content based on user relationships and interests. But it does not gain any actual usable information from this and does the matching process obliviously. With the size of the likes of Twitter in mind, we put an increased focus on the scalability of our system.

We identify the need for Private Information Retrieval (PIR) and discuss existing methods in terms of use and scalability for our context. We choose a method based on the MapReduce [16] programming model and implement a fully working system that allows subscribing to users and topics without revealing the relationship. We try to minimize user overhead both in terms
of computation and required user knowledge, opting for a seamless integration of our system through browser extensions and back-ends. In a broader view, we discuss the use of Private Information Retrieval in a highly dynamic database. A lot of work on PIR has focused on, in our opinion, more impractical and theoretical databases which are static. Our system is highly dynamic and requires pragmatic solutions to how we structure the database and how we run PIR efficiently.

We evaluate the performance and scalability of our system critically and discuss specific and general problems of designing a privacy-enabled microblogging system. We analyze the effectiveness of our design and implementation and attempt to offer insight into problems that may occur in using PIR in a MapReduce context by providing in-depth performance data we have gathered during an extensive testing phase in a real world cloud setup. We discuss the open problems, of theoretical and practical nature, and future work that might be done on this topic.

1.3 Related work

In this section we present related work on the topic of microblogging and social network privacy as well as private information retrieval.

As mentioned before the work of De Cristofaro et al. [15] tackles a very similar aspect of microblogging privacy. They focus on content privacy as well as subscriber privacy. All messages are uploaded in encrypted form. A user subscribes to a specific topic of a specific publisher such that neither the provider nor the publisher know what topic the user is interested in. To this end oblivious pseudorandom functions are used such that a subscriber can derive a decryption key and database index for that topic without the publisher knowing the input, i.e. the topic of interest. Derived database indices are uploaded by both the publisher with the message it applies to, as well as the user as part of his subscription to that publisher. The microblogging provider then matches subscriptions to messages and delivers the relevant messages to the subscriber. Clearly the provider is aware of all relationships between subscribers and publishers (just not the exact topic). Our work focuses on exactly that point of concern. We do not encrypt message content, but our system is easily extendable to allow for this.

Bachrach et al. [3] look at private group communication in microblogging services and focus on censorship resistance and deniability through deliberate hash collisions.

Luo et al. [28] look at a broader spectrum of social networks and propose a solution to protect arbitrary private information by uploading fake information to the providers and storing the actual information in encrypted form on third-party servers such that only authorized users are able to retrieve it.
In a similar fashion Guha et al. [21] develop a system of encrypting and replacing parts of information such that a provider does not notice the use of their system. Dispensing with the problem of a centralized all-knowing service provider Cutillo et al. [13] propose a wholly different approach in using a peer-to-peer social network that leverages the inherent trust relationships of users and a system of trusted identification via certificates.

The problem of privately delivering data to subscribers in social networks can be seen as a special case of private Publish/Subscribe networks. In Publish/Subscribe networks subscribers can subscribe to content of publishers by using arbitrarily complex filters. These subscriptions are sent to brokers, who are usually arranged in a hierarchy and deliver content from publishers or relayed by other brokers to subscribers based on these filters. In essence, private Publish/Subscribe deals with the problem of obliviously routing messages in a multi-hop network based on filters. In common microblogging systems, there is only a single centralized broker - the provider server. Shikfa et al. [35] propose a privacy-preserving Publish/Subscribe network by using multiple layer commutative encryption. This allows subscribers and publishers to encrypt their content and filters with multiple layers and the intermediate brokers to strip off any single layer and compare the result to a specially built encrypted routing table. Ion et al. [22] use Attribute-Based Encryption together with searchable encryption for these attributes to allow for publisher and subscriber confidentiality while brokers can still match the encrypted attributes of published content and subscriber filters.

With regard to the scalable use of Private Information Retrieval there is a wealth of work aimed at increasing communication efficiency [10, 11, 18, 27]. Additionally there are a few papers which focus on computational efficiency. Blass et al. [7] as well as Mayberry et al. [29] leverage the natural parallelism of the basic PIR scheme and use the MapReduce paradigm to allow for scalable PIR. The former paper deals with private word search on encrypted data and uses hashing to generate a temporary database on which a form of PIR is used to retrieve the element corresponding to the word that is queried. The latter work uses MapReduce to retrieve large files from a relatively small database efficiently. Both papers make use of a very efficient new cryptosystem introduced by Trostle and Parrish [39]. In similar direction, Blass et al. [8] use MapReduce and a newly designed cryptosystem based on the work of Trostle and Parrish to allow for efficient frequency counting in encrypted databases. Aguilar-Melchor et al. [1] use a more local approach by leveraging the increasing parallel computation power of modern Graphics Processing Units and lattice-based homomorphic encryption.

Ishai et al. [23] offer an approach to reducing PIR computational overhead
when retrieving more than one element. They make use of hash functions or batch codes alternatively to structure the database such that the queries can be applied to different subsets of the database, thus reducing total computation.

Freedman at al. [17] present interesting approaches to private information retrieval using keywords instead of numerical indices. They use hashing of the database and oblivious polynomial evaluation to allow a user to retrieve elements using his keyword instead of having to determine the index of the element in the database.

Finally there are some papers [30, 36] which discuss the practicality of PIR schemes in general; specifically whether any PIR scheme can beat the time required to download the whole database.

1.4 Organization

This thesis continues with an introduction and explanation of the main elements of the rest of the work in Chapter 2 on page 7.
Continuing with Chapter 3 on page 15 we delve into the design of our microblogging system and discuss the choices we made as well as the reasoning behind them.
In Chapter 4 on page 37 we provide more detailed information about the actual implementation and talk about obstacles encountered and ways to circumvent these.
Chapter 5 on page 51 is dedicated to the evaluation of our system and includes a multitude of results and discussions thereof.
We conclude the thesis in Chapter 6 on page 73 by summarizing our work and talking about open problems and future work.
Chapter 2

Preliminaries

In this chapter we will present the necessary background information which we will make use of for the rest of the thesis.

2.1 Twitter

Twitter (https://twitter.com) is the most popular microblogging service in the world, boasting over 200 million active users [9] including celebrities, politicians, companies and more. It has long established itself as the go-to juggernaut of microblogging with users sharing over 400 million messages each day [9]. In the following we will briefly describe the average use case of Twitter and introduce the unique terminology used by Twitter and its users. We will make use of it for the remainder of this thesis.

- A user who publishes messages is a tweeter.
- A published message is a tweet.
- A topic associated with a message is a hashtag
- A user who subscribes to a publisher is a follower.
- The service provider is simply referred to as Twitter.

Whenever we use the term “user” without the distinction of tweeter or follower, we are referring to both categories or we are talking in a general context outside of our Twitter system. In particular, note that there is no active categorization of users in our system; a user can both post tweets as a tweeter and follow others as a follower.

A tweeter can post tweets on his profile or in response to other tweets. In the spirit of microblogging tweets may only be up to 140 characters long. A user can choose to follow a tweeter by clicking the appropriate button in the tweeter’s profile. As a follower the user will receive all tweets on his
profile page which is also referred to as his news feed. We call this process of delivering the appropriate tweets to followers matching. It is obviously performed by Twitter servers.

As we show in Example 2.1, a tweeter can mark his tweet with one or more hashtags which describe the topic of the tweet. These hashtags are part of the message and preceded by the number sign #.

Example 2.1 @jack I like my privacy! #privacy #thesis

Users can search for tweets by hashtag. For example searching for either “privacy” or “thesis” would return the tweet of Example 2.1. Because Twitter is open by default a search will return all tweets with the hashtag of interest, not just those of tweeters the searcher is following. Additionally all tweets are publicly accessible by looking at user profiles and indexed by search engines like Google. Tweeters can choose to keep their tweets protected [37] such that a user needs manual approval before he can follow that tweeter and read his tweets. Protected tweets are obviously not searchable unless the searcher is an authorized follower. Twitter as a provider still has access to protected tweets. Tweeters can refer to or address someone in their tweet by prepending an “@” to their user name. In Example 2.1 we address the user jack (Jack Dorsey, Co-founder of Twitter).

Finally, users can propagate a tweet by retweeting it to their own followers.

2.2 Private Information Retrieval

Private Information Retrieval, PIR in short, refers to the act of retrieving information from a database without the database operator gaining any information about what was retrieved. There has been a good amount of research and advances in this topic in the last years. We will begin with a problem statement as usually considered by PIR designers.

A database holds a public \( n \)-bit string. Each bit of the string can be seen as a database element. The data is public, i.e. not encrypted, but central to the database. A user wishes to retrieve a single database element, i.e. a single bit at position \( i \) in the bit string. The cryptographic game addressed by PIR is played by the database and the user who wants to retrieve this bit without the database gaining any information about which bit was retrieved. More formally, given two query distributions \( Q_i, Q_j \), to retrieve index \( i \) and \( j \) respectively, and any polynomially bounded distinguisher \( D \), the advantage \( \Delta_D \) of \( D \) in distinguishing between the two query distributions, i.e. in being able to determine that they refer to different indices \( i \neq j \) (or equal indices) is defined as:

\[
\Delta_D(Q_i, Q_j) := |P(D(Q_i) = 1) - P(D(Q_j) = 1)|
\]
2.2. Private Information Retrieval

For a PIR scheme to be considered \textit{computationally} private, we require that it produces query distributions such that \(\Delta_D(Q_i, Q_j)\) is negligible in a security parameter. For \textit{information-theoretic} privacy it must be 0 for unbounded distinguishers.

The naive solution, \textit{trivial PIR}, requires the database to send the whole \(n\)-bit string to the user who then reads the \(i^{th}\) bit. The database obtains no information at all, but the user needs to download the whole content of the database; the communication complexity from database to user is \(n\). The goal of Private Information Retrieval hence is retrieving data with communication smaller than \(n\).

The concept of PIR was introduced by Chor et al. \cite{12} in 1995 in a setting with multiple databases holding copies of the same data and not allowed to communicate among each other. In this setting it is possible to achieve information-theoretic PIR. The authors also showed that in the setting of a single database it is not possible to achieve information-theoretic security (except through trivial PIR). Henceforth the goal of researchers was to find computationally secure PIR schemes with good communication performance. We refer to this type of private information retrieval as computational PIR.

The first work on computational PIR was presented by Kushilevitz and Ostrovsky \cite{26} in 1997. It relies on algebraic properties of the Goldwasser-Micali public-key encryption system \cite{19} and achieves a communication complexity of \(O(2^{\sqrt{\log n \log \log n}})\). There have been many other works which improved communication complexity, such as Cachin et al.’s scheme \cite{10} based on the \(\phi\)-hiding assumption and Lipmaa’s endeavour \cite{27} of arranging the database in more efficient ways. We refer readers interested in an overview of PIR-related work to \cite{5, 32}.

2.2.1 PIR based on homomorphic operations

In the following we present a generalized way of looking at private information retrieval as a series of homomorphic operations on the database. This view is taken from Ostrovsky et al.’s survey on PIR \cite{32} and treated in more detail therein. Many of the concepts in the subsequent discussion will be repeatedly used in our own system.

Let \((\mathcal{E}, D)\) be a cryptosystem where \(\mathcal{E}\) is a probabilistic encryption algorithm and \(D\) the corresponding decryption algorithm. Key generation is omitted since it is of little immediate importance. Probabilistic encryption makes this system computationally secure against chosen ciphertext attacks; even if there are only two possible messages a computationally bounded adversary must not have a non-negligible advantage in distinguishing two ciphers. This is significant because in the following PIR scheme there are in fact only
two possible messages. Additionally, the encryption system shall support homomorphic operations on the ciphertexts. Let \((G, \cdot), (H, \star)\) be abelian groups describing the set of plain texts and ciphertexts respectively. Then the cryptosystem shall have the property that
\[
\mathcal{D}(\mathcal{E}(a) \cdot \mathcal{E}(b)) = \mathcal{D}(\mathcal{E}(a)) \cdot \mathcal{D}(\mathcal{E}(b)) = a \cdot b
\]

Effectively anyone can obliviously perform the group operation in \(G\) if he has the ciphertexts in \(H\), without the need of decrypting anything. Additionally we can perform the group’s equivalent of scalar multiplication by using the \(\mathbb{Z}\)-module operation with a scalar.
\[
\mathcal{D}(\mathcal{E}(a) \times \beta) = \mathcal{D}(\mathcal{E}(a)) \times \beta = a \times \beta
\]

Where \(\beta \in G\) and \(\times\) denotes the \(\mathbb{Z}\)-module operation in both groups with respective group operator.

Let \(\{x_i\}_{i=0}^{n-1}\) be the set of database elements \(x_i \in \{0, 1\}\) indexed with \(i \in [0, n-1]\). A user who wants to retrieve element \(k\) sends a set (or query vector) \(\{q_i\}_{i=0}^{n-1}\) to the database where
\[
q_i = \begin{cases} 
\mathcal{E}(g) & \text{if } i = k \\
\mathcal{E}(e_G) & \text{otherwise}
\end{cases}
\]

\(g\) and \(e_G\) are an arbitrary non-neutral element and the neutral element of \(G\) respectively. Note that each element in the query vector is encrypted separately, despite repeating plain text. I.e. \(e_G\) will not be encrypted a single time and the cipher reused, but rather encrypted \(n-1\) separate times. This ensures that the database cannot distinguish between query vector elements. The database then computes and sends the response
\[
R = \bigstar_{i=0}^{n-1} q_i \times x_i
\]

The user can then decrypt the database response
\[
\mathcal{D}(R) = \mathcal{D}(\bigstar_{i=0}^{n-1} q_i \times x_i) \overset{(1)}{=} \bigstar_{i=0}^{n-1} \mathcal{D}(q_i) \times x_i \overset{(2)}{=} \mathcal{D}(q_k) \times x_k = g \times x_k
\]

Transformation (1) is valid due to the homomorphic properties of our cryptosystem. Transformation (2) uses the fact that \(\mathcal{D}(q_i) = e_G\) for \(i \neq k\).
2.2. Private Information Retrieval

The user can thus derive that $x_k$ is 1 if and only if $D(R) = g$. Or in the case where the discrete logarithm is efficiently computable in $G$ he can compute $x_k = \log_g D(R)$.

In many (but not all) cases $G \subseteq \mathbb{Z}$ and $H \subseteq \mathbb{Z}^*$ with additive and multiplicative operators respectively. Now this method becomes more intuitive as the user sends $E(1)$ for $i = k$ and $E(0)$ for the rest. Then the user retrieves

$$D(R) = D(\prod_{i=0}^{n-1} q_i^{x_i}) = \sum_{i=0}^{n-1} D(q_i) \cdot x_i = D(q_k) \cdot x_k = x_k$$

Note the difference of the $\mathbb{Z}$-module operation. In $H$ we compute exponentiation and once we apply the homomorphic transformation (1) we use multiplication in $G$.

Intuitively, the database “zeroes out” all the database elements which the user is not interested in but leaves $x_k$ intact.

The problem with this PIR approach is, of course, that the user communication overhead is linear in the size of the database. As a matter of fact, since we are sending ciphertexts and $|H| > |G|$ it has worse communication complexity than trivial PIR.

A first straightforward optimization is to arrange the database in a square matrix $\{x_{ij}\}_{i,j=0}^{\sqrt{n}-1}$. The user can now send a shorter query vector $\{q_i\}_{i=0}^{\sqrt{n}-1}$ where $q_i = E(1)$ for $i = k^{\sqrt{n}}$.

The database calculates

$$R_j = \prod_{i=0}^{\sqrt{n}-1} q_i^{x_{ij}}$$

It sends back this vector $\{R_j\}_{j=0}^{\sqrt{n}-1}$ and the user retrieves the element of interest by looking at the vector element at position $k \mod \sqrt{n}$. Effectively the user has queried a whole row of interest instead of a single element. The communication is now proportional to $\sqrt{n}$. One could also say that communication is “balanced out” since the user has to send less, but the database more (a whole row).

We can further decrease the complexity by reducing the amount of data the database has to send back. It is easy to see that the row it calculates is simply a smaller database that the user can query as well, since he knows the position of his element of interest in that row. To do this we can define an injective map

$$\varphi : H \rightarrow \mathbb{Z}^l$$
such that for all \( x \in H \) each component of \( \varphi(x) \) is less than \( \text{ord}(g) \). A trivial way of doing this would be to divide the binary representation of \( x \) into bit-strings of \( \log \text{ord}(g) \) length. We need to do this because the encrypted elements \( R_j \) of the smaller database are elements of \( H \) so we need to “split” them up. The user can now send a second PIR query vector \( \{ \varphi_i \}_{i=0}^{\sqrt{n}-1} \) where \( \varphi_i = \mathcal{E}(1) \) for \( i = k \mod \sqrt{n} \).

The database calculates

\[
\overline{R}_t = \prod_{j=0}^{\sqrt{n}-1} \varphi_j(R_j)_t
\]

And the user can retrieve his element of interest by calculating \( \varphi^{-1}(D(\overline{R}_t)) \).

Note that the secondary, “recursive” PIR query does not rely on the intermediate result so both queries can be sent in a single round. This also implies that security is not affected by this rearrangement, because both query vectors are secure under the assumption that the linear scheme is secure.

This idea of rearranging the database and sending a “recursive” PIR query to minimize the database communication complexity is quite generic and used by a number of PIR schemes \([11, 26, 27]\). Particularly, it can be generalized to reducing the database to a \( d \)-dimensional hypercube and having the user send up to \( d \) PIR queries. Lipmaa \([27]\) uses this to great effect by also relying on a specific length-flexible cryptosystem which further reduces the database communication.

### 2.3 MapReduce

MapReduce is a programming model proposed by Google \([16]\) and designed for the processing of large amounts of (streaming) data in a massively parallel and easily scalable way on a cluster of commodity machines, i.e. the cloud. Programmers need only provide so-called map and reduce functions. A framework then handles the bulk of the background work like splitting the data smartly, distributing it efficiently, handling fault-tolerance, collecting output and more. The real beauty of this paradigm is that a programmer does not need to concern himself with the various difficulties of parallel processing. As a matter of fact, he only needs to program his functions a single time and they will scale to arbitrarily large clusters transparently.

MapReduce consists, as the name would suggest, of two main execution phases (cf. Figure 2.1 on the next page). In the map phase a set of workers, mappers, run the map function on disjoint parts, splits, of the input data. The map function receives one key-value pair at a time, generated from the data by a special reader, and processes this as the programmer specified in the
2.3. MapReduce

Figure 2.1: Simplified overview of the map, shuffle, and reduce phases during a MapReduce run.

map code. It outputs so-called intermediate key-value pairs to the framework. In the second phase the reducers receive these intermediate values, sorted and combined in the shuffle phase. They apply the reduce function to lists of values with the same intermediate key and thus can aggregate and combine the intermediate data. This reduced data is written to the output and made available to the programmer.

The mapper receives key-value pairs from key space $K_1$ and value space $V_1$. It outputs zero or more key-value pairs in intermediate key-value space $(K_2, V_2)$ per input. The reducer receives the shuffled mapper outputs as pairs keyed in $K_2$ and containing lists of $V_2$ values. It outputs zero or more key-pairs in $(K_2, V_2)$.

$$
\text{map} : (K_1, V_1) \mapsto (K_2, V_2)^* \\
\text{reduce} : (K_2, (V_2)^*) \mapsto (K_2, V_2)^*
$$

The typical example used to explain MapReduce is word-count, i.e. counting the number of occurrences of each word in a large text. To this end, the map function would be fed the words of a text. For each word $w$, it emits the key-value pair $(w, 1)$. A reducer will get all values pertaining to the same key, i.e. $(w, (1)^*)$ and can simply aggregate this by summarizing all values. It emits this sum into its output together with the key $w$.

2.3.1 Hadoop

Arguably one of the most well-known and used MapReduce frameworks is the open-source Apache Hadoop project (http://hadoop.apache.org). It is based on Java and offers a library that allows programmers to easily
write MapReduce programs in the Java programming language. Hadoop by default stores input and output data on a distributed file system called Hadoop Distributed File System (HDFS). HDFS handles the distributed data storage and replication across machines in the cluster. Hadoop leverages the distributed file system during MapReduce runs so that network communication is kept to a minimum. It will try to schedule machines which have HDFS blocks of data on their local storage such that they will work on that data instead of having to copy it from other machines. This data locality is an integral efficiency factor for Hadoop and MapReduce in general. HDFS also has a few drawbacks which influence our plan of running a dynamic database on it. These problems and some solutions are discussed later on in Section 4.2.1 on page 38.
In this chapter we discuss the main design choices we have made to arrive at a manageable and efficient microblogging system. We preface it by listing our goals and assumptions. Then we treat various important design aspects. These sections are usually of the same format: we start by describing the subproblem we wish to tackle, discuss several alternative solutions we have considered and finally describe the approach we have decided on and used in our system.

3.1 Goals and assumptions

Our goals in designing the microblogging system fall into the category of security and usability.

- **Centralized matching service:** There is a central service provider ("Twitter" / "server") which handles matching and delivery of tweets to followers. There is no significant additional overhead for followers in trying to find the necessary tweets.

- **User relationship privacy:** The server does not learn significant information about the subscriptions it handles. The only thing it learns about relationships is the number of tweeters a follower is following. If the user base is of size \(n\) then the server is not able to get a non-negligible advantage above \(\frac{1}{n}\) in guessing what tweeter a subscription relates to. It learns everything about the content of tweets but not about who reads them. As we have mentioned, private content has been addressed before [15] and our system will be extendable to allow for this.

- **Usability:** The system should be usable in a similar degree as the original Twitter. This does not necessarily refer to the ability to handle the same huge volume, but it should retain the basic property of a
short message news system and scale well with increasing number of tweeters and followers. Specifically tweets should be delivered in due time. A follower shall be able to subscribe to tweeters and also to certain hashtags.

The fact that the server knows the number of subscriptions of a follower can not easily be mitigated. Even with perfect PIR the server can assume that a batch of queries relates to distinct elements. In general we assume that without involved anonymization techniques like dummy subscriptions it is not possible to hide the number of subscriptions from the server.

Because the provider offers a public service and has a vested interest in keeping its reputation and user base we assume that it follows the Honest-but-Curious adversary model. Any adversarial action beyond this would quickly ruin its reputation and cause users to leave the service. In this model, the adversary faithfully follows protocol specifications but may try to violate our privacy goals by passively collecting and analyzing information. In particular, we assume that it faithfully executes its part of PIR and does not try to force additional user interaction by purposely providing wrong results. Obviously we assume that interactions with the server and interactions between users generally run on distinct channels such that the server is not able to eavesdrop on communication that is not addressed to it.

### 3.2 Private Information Retrieval

Because the server cannot know about user relationships, matching of tweets to subscriptions obviously cannot be done in the clear. In a regular Twitter context, matching can be seen as followers retrieving database elements (tweets) with their (stored) subscriptions. Clearly we need to use a system that enables a follower to retrieve his tweets from the server without it knowing what was retrieved. This is the exact problem statement that Private Information Retrieval aims to tackle. There are other methods applicable in this scenario and related to PIR, for example oblivious transfer. Oblivious transfer allows the retrieval of an element without the server knowing which and without the user gaining information about the other elements. This additional requirement generally makes oblivious transfer less efficient than PIR. We do not require database secrecy since Twitter is public and, as mentioned before, encryption can always be added on top of our system.

As we have seen in Section 2.2 on page 8 there is a slew of different PIR schemes that aim to minimize communication complexity. Our challenge lies in the fact that we aim to process large amounts of data in a scalable manner. An inherent invariant of PIR is that the server needs to touch all bits in the database during query processing, otherwise the server could infer information about the query by looking at what was not touched. Further-
more this “touch” usually amounts to one or more expensive cryptographic operations. While many PIR papers focus on the communication complexity of their schemes it is evident that in practical settings this computational overhead is of significant importance. As a matter of fact, Sion et al. [36] claim that this overhead makes any PIR scheme inferior to trivial PIR, i.e. sending the whole database. They argue that on current and, through extrapolation, future commodity machines, a cryptographic operation of PIR will always outweigh the overhead of transferring a bit over the network. In a rebuttal of kinds, Olumofin at al. [30] aim to disprove this notion by looking at various recent advances in private information retrieval. They show that a scheme proposed by Aguilar-Melchor et al. [1] which leverages the parallel capabilities of modern Graphics Processing Units is a prime counter-example which is generally faster than trivial PIR except for very small databases. Parallel processing is the keyword here. We are forced to touch the complete database so the only options we have is the reduction of the “touch overhead” with more efficient cryptosystems and by distributing the computation overhead across many parallel workers.

3.2.1 Our approach: PIRMAP

After surveying parallel PIR schemes we decided to focus our attention on Mayberry et al.’s PIRMAP [29]. This scheme uses the inherent parallel characteristics of homomorphic PIR as described in Section 2.2.1 on page 9. A database calculates the answer to a query with:

\[
R = \prod_{i=0}^{n-1} q_i \times x_i
\]

Each database element is applied to the appropriate query element in a homomorphic operation and then all intermediate results are summed up (where summed refers to the appropriate cipher group operation). The first step can clearly be parallelized such that multiple workers handle a subset of the database. As a matter of fact, it sounds perfectly suited for the MapReduce paradigm of mapping, i.e. applying the query elements, and reducing, i.e. summing up the intermediate results. Additionally the authors of PIRMAP handle file retrieval in an efficient fashion. Up until now, we have only talked about PIR in the context of retrieving single bits. In practical applications a user usually wants to retrieve a file, i.e. a bit-string, of information. This is easily achieved if we remember that homomorphic cryptosystems are not restricted to operations with binary values, but rather work for any plain text bit-string that is encodable as an element of plain text group \(G\).

Let \(k \geq |G|\) be the security parameter of the encryption used. PIRMAP di-
3. Design

![PIRMAP file block scheme](image)

Figure 3.1: PIRMAP file block scheme. Files are split into blocks and applied to the query element per row. Then outputs are summed up into blocks $R_j$ which correspond to the encrypted blocks of the file of interest.

vides each file of length $l$ and indexed by $i$ into $\lceil \frac{l}{k} \rceil$ blocks $b_{ij}^{\lceil \frac{l}{k} \rceil-1}$ of length $k$. These blocks can therefore be used in the generic PIR scheme (cf. Figure 3.1).

$$R_j = \sum_{i=0}^{n-1} q_i \times b_{ij}$$

The attentive reader will notice that this is reminiscent of the square matrix optimization in Section 2.2.1 on page 9. The only difference is that there is no further recursive PIR query because each $R_j$ encrypts one $k$-bit block of the file that the user is interested in.

The MapReduce realization is straightforward and can be seen schematically in Figure 3.2 on the next page.

- **Map.** Each mapper gets one or more files. It splits a file into blocks as defined above and applies the query element required for that file to each block. It emits each block with a key holding the block index.

- **Reduce.** A reducer, with each input, gets the list of blocks with the same block index $j$ for all files. It sums up all blocks and emits the result $R_j$.

**Somewhat homomorphic encryption**

The final piece of the puzzle to the efficiency of PIRMAP is a special cryptosystem originally proposed by Trostle and Parrish [39]. The following description of the cryptosystem is taken and adapted from the PIRMAP paper [29].

Let $q$ be the number of bits we want the cryptosystem to be able to encrypt and let $k > q$ be the security parameter. The cryptosystem works as follows:
3.2. Private Information Retrieval

KeyGen(1^k): Generate a modulus m of k bits and a random coprime b < m.
Encrypt E(x) = b \cdot (r \cdot 2^i + x) \mod m, for random r
Decrypt D(c) = b^{-1} \cdot c \mod m \mod 2^i

This system has the necessary homomorphic properties:

Addition
\[ E(a) + E(b) = b \cdot ((r_a + r_b) \cdot 2^i + (a + b)) = E(a + b) \]

and scalar multiplication
\[ E(a) \cdot \beta = b \cdot (r\beta \cdot 2^i + a\beta) = E(a\beta) \]

Note that as opposed to many cryptosystems with cipher in \( \mathbb{Z}^* \) and which therefore require modular multiplication and exponentiation for these operations, the Trostle&Parrish cryptosystem relies on the much more efficient integer addition and multiplication. The drawback is that this cryptosystem
is only somewhat homomorphic. In real homomorphic systems there can be an arbitrary number of homomorphic operations. In our cryptosystem the reader will note that after some point \((r_a + r_b) \cdot 2^q + (a + b)\) from above will exceed \(m\) causing a “wrap” that makes the decrypted value incorrect. We have to account for this by increasing the size of \(m\) such that it “supports” the additions and multiplications we intend to apply to the ciphers.

In the PIRMAP case we require one scalar multiplication with the file block and \(n\) additions to sum up the intermediate encrypted blocks. To support \(n\) additions we need to enlarge \(m\) by \(\log_2 n\) bits. A scalar multiplication can be thought of as up to \(2^q\) additions so we need to double the size of \(m\). Therefore we must choose \(m\) to be \(O(2^k + \log_2 n)\) bits. Additionally, the cryptosystem is tailor-made for PIR in that encrypting values larger than 0 or 1 and performing operations on the ciphers will lead the payload \((a + b)\) to overflow into the randomness \((r_a + r_b) \cdot 2^q\), which would lead to loss of information during decryption.

The security of this cryptosystem relies on the Hidden Modular Group Order assumption which states that an adversary only has negligible chance of correctly guessing the lower order bits of the inner bracket in the encryption, i.e. the unblinded value \(r \cdot 2^q + x\). This information could be used to then compute the secret trapdoor group modulus \(m\). In particular this assumption relies on the fact that the adversary only has a small number of blinded values compared to the size of \(m\), hence we cannot reuse keys. Cf. Trostle and Parrish’s paper [39] for more information and a security proof.

Adaptation to our system

The main problem of PIRMAP is that the user needs to send a query that is linear in the size of the database. Clearly the communication complexity is far from optimal. The authors of PIRMAP rationalize this by focusing on databases with few, but large files. In this case the communication from user to server is negligible compared to the communication from server to user, i.e. the large, encrypted file it returns.

As for our case, we actually generally have many small files, so at first glance it seems that PIRMAP is not very suitable. We justify our choice with several observations.

- **Cached user queries:** While sending a query might be a large overhead, in our system followers only send queries when they update their subscriptions. They do not need to send it every time the server matches tweets. Therefore the increased upload time is amortized both by its infrequent need and by the fact that in return we are able to use an efficient PIR scheme. We go into greater detail about query caching in Section 3.6 on page 33.
3.3. Database

- **MapReduce is practical**: We have not found much work on parallel schemes. There is, as mentioned before, the GPU-based work of Aguilar-Melchor et al. [1] and there are some based on MapReduce [7, 8, 29]. The advantage of MapReduce is its practicality. With today’s prevalence and increased use of large cloud data centers, MapReduce is a prime choice for companies dependent on large-scale computations. MapReduce can be run on off-the-shelf commodity hardware as well as specialized high-volume clusters. In short, using a scheme based on MapReduce makes our system widely usable.

Finally, we adapted PIRMAP to use the obvious progression as already seen in Section 2.2.1 on page 9. We rearrange the file database into a square matrix and query both the row of interest and, recursively, the column of interest in the intermediate result. Thus we decrease user communication to $O(2k\sqrt{n})$. The drawback of this step is that we need to run two sequential MapReduce jobs, the first reduces the database to row of interest, the second to column of interest. The inherent overhead of starting and managing a MapReduce job is the reason we did not extend this adaptation to a higher dimensional hypercube. There are other issues as well, pertaining to the cryptosystem, which we elaborate on during the evaluation in Section 5.4 on page 53.

3.3 Database

We describe how we design and structure the database such that it can be used efficiently by the PIR scheme.

3.3.1 Granularity

In a regular microblogging system one can assume that tweets are stored separately in an appropriate database system, e.g. SQL. This kind of tweet-level database granularity does not lend itself well to our system. Remember that a PIR query is always related to database size. If each tweet is a separate database element it is easy to see that the growth of the database and by relation the growth of the PIR query gets out of hand.

On the other extreme we could decrease query size by putting all tweets into a single file. We call this act of putting different tweets into a file clustering. Obviously clustering, like granularity, can be done on many scales. The aforementioned one-file cluster clearly is too extreme since the user would have to download the giant file. Some balance between tweet-level granularity and one-file clustering is the obvious solution.
3. Design

**Bin-packing clustering**

A potential solution that we have considered uses the fact that tweeter behavior of an average microblogging system can often be modelled with a Pareto distribution; a minority of the tweeters account for the majority of posted tweets [38]. With this in mind we could cluster several small sets of tweets of low-activity tweeters into a single file of about equal size as those of the active tweeters. The main problem with this approach is that distributed systems, or specifically HDFS, treat files as immutable once they have been closed. This means that they cannot be changed afterwards, except, thanks to recent improvements, by appending to them. As a result bin-packing clustering is hard to realize. Assume a low-activity tweeter shares a file with several others. If he suddenly starts to tweet more than expected, we have 3 choices:

1. Rewrite everything into at least two separate files
2. Start a new file that contains the second part of the tweeter’s tweets.
3. Accept our fate and from now on have a larger file than even those of the active tweeters.

Option (1) is obviously inefficient because we need to rewrite everything, which on a distributed system is generally even less efficient than on a local file system. Option (2) makes retrieval much more involved and inefficient because we need to serve / combine multiple parts so that a follower receives all tweets. Option (3) makes our database unbalanced; we have a few files that are larger than the rest and our PIR efficiency suffers for it. We could also do this bin-packing clustering a posteriori, i.e. on the fly as we serve queries. The problems are twofold. For one we consider it infeasible to do fast on-the-fly bin-packing of very large databases without using streaming bin-packing algorithms with the disadvantage of less than optimal packing density. Secondly, element indexing is more involved. Remember that users generate PIR queries based on the index of the element they wish to retrieve. With on-the-fly clustering we run into problems with making these dynamic indices available to the user. At the very least it makes any sort of query caching impossible because each new retrieval potentially encounters a database with different indices. Additionally synchronization becomes an issue as the database must not change between the user requesting a list of indices and him sending the query that pertains to these indices.

**Our approach**

We choose to do the clustering in a pragmatic way. To handle queries for subscriptions to tweeters we simply combine all tweets of a certain tweeter into a single file. Therefore a follower can retrieve files directly according to his subscriptions. It is easily achievable a priori because we can just append
a file when new tweets are uploaded. Since we also want to be able to subscribe to hashtags, we keep two databases with the same data arranged differently; one per-tweeter and one per-hashtag database. In the hashtag database we obviously cluster tweets into files per hashtag. We justify this redundancy by the efficiency this pragmatic approach brings and by the fact that database storage is cheap and continues to become cheaper.

The disadvantage of tweeter-/hashtag-level clustering is that a follower will always retrieve all tweets of a tweeter, including “old” tweets which he has already retrieved before and is no longer interested in. This problem of redundant data retrieval is partially addressed by another design choice in Section 3.3.2.

Note that in many of the following sections we will only refer to the per-tweeter database in the interest of concise treatment and without loss of generality with respect to the per-hashtag database.

3.3.2 Windowing

Microblogging services generate a large amount of messages every day. As we have noted in the previous section, followers retrieve all the tweets of a tweeter in a single file. This file contains a lot of redundant tweets which increases both communication and computation overhead. As a matter of fact, we view it as unscalable if not infeasible to return all tweets a tweeter has ever posted in this manner. It does not even matter how the database is arranged, since PIR needs to touch all its data it quickly becomes infeasible to store all tweets in a PIR-enabled database.

For this reason we include, what we call, a recentness window in our system. This enables our system to discard tweets which are no longer “recent” and keep the database small. We argue that this is compatible to the general use case of a microblogging system where followers wish to read “real-time” information and old information is generally not interesting. We define “recentness” in two ways.

Time recentness. This is the obvious definition. Any tweet which is older than a certain time span is discarded.

Capacity recentness. We only store the last $m$ tweets of each tweeter, effectively imposing a capacity on the tweeter files.

Both views have some advantages and disadvantages. Time recentness is awkward for inactive tweeters. If a tweeter does not post anything for a long time, all his old tweets will be discarded and it looks like the tweeter has never posted anything. With a capacity window, one would still see the last $m$ tweets of that tweeter, even if they are really old. An additional advantage of a capacity window is that over time, the tweeter files “balance” out. Low-activity tweeters slowly reach the capacity limit and high-activity tweeters cannot surpass it. This way the database converges into a state
where all tweeter files are equally large. This is usually preferable over an asymmetric database, especially in the context of parallel processing where it is not efficient to have asymmetric worker time, i.e. workers having to wait for the ones with more work. The obvious drawback of a capacity window is that very high activity tweeters may “break” the limit too quickly such that earlier tweets are discarded before their followers have had a chance to read them. We argue that with an appropriate capacity this is not a realistic concern since followers will not bother to go through hundreds or thousands of tweets that have been posted within a short time.

For our system, we have decided to use a hybrid approach. Files are limited to a certain capacity with earlier tweets discarded to make space for newer ones. Additionally we periodically purge old tweets. We justify this by the decrease of the overall database size and the resulting decrease in matching time. This, of course, reduces the symmetry of the database that the capacity window provides, but the decrease in size and appropriate configuration of the windows should mitigate this. Additionally the database asymmetry is partially addressed by certain implementation details treated in Section 4.2.2 on page 44.

3.4 Batch queries

Users in our system may follow more than one tweeter. This means that a follower wants to retrieve multiple files from the database. We describe two optimizations to mitigate the overhead of multiple queries.

3.4.1 Reducing communication

The naive solution to retrieving $k$ database elements with PIR is to send $k$ independent PIR queries and have the database evaluate each one. The main issue with this is that our follower communication and computation would increase linearly with the number of tweeters he follows.

We use a simple trick suggested in [18] which allows us to “reuse” a single PIR query $k$ times. The follower generates this query for the first element with index $i_0$ he wants to retrieve. For all subsequent elements of interest $\{i_t\}_{t=1}^{k-1}$, the follower calculates the offset $o_t = i_0 - i_t$ and stores that as a sub-query in the original query. For each sub-query, the database simply temporarily relabels the element indices by shifting them by $o_t$. Thus the original query can effectively be applied to retrieve $i_t$. Without loss of generality in terms of other database arrangements (e.g. into a square matrix), we can also view this as the follower telling the database to create a (hypothetical) $k \times n$ matrix by “duplicating” the original data $n$-vector $k$ times and shifting it according to the offsets (cf. Figure 3.3 on the next page). The follower then privately retrieves column $i_0$ from the matrix.
Figure 3.3: Hypothetical view of offset batch queries. They “instruct” the database to create a $k \cdot n$ matrix of shifted rows. The original query meant to retrieve element $i_0$ then retrieves the whole column $i_0$ containing all the elements the batch query is supposed to retrieve.

The distinct drawback of this method is the security loss due to the fact that the security of all the queries rely on the intractability of distinguishing the elements of the single query vector. If the adversary is able to gain information about this “starting point”, he will automatically be able to apply it to all subsequent indices because the offsets are in the clear.

Intuitively, the adversary only needs to get information about one column in our hypothetical matrix view instead of separate indices when using trivial batching. Furthermore this means that the adversary can distinguish full batch queries with 100% accuracy. If two batch queries contain offsets with different intervals between them, they cannot possible retrieve the same set of elements; i.e. the two queries generate different hypothetical matrices.

Assume that the adversary has no popularity information and therefore needs to assume a uniform distribution of the interest in the database elements. Note that this is a valid assumption since the whole point is to mask this with PIR. Then the probability of guessing all subqueries of a batch query is $\frac{1}{n}$ whereas it is only $\frac{(n-k)!k!}{n!}$ with trivial batching (if the server assumes that all queries are for distinct elements). The chance that any specific element is part of the queried set remains the same at $\frac{k}{n}$.

Because the security of this scheme still relies on the intractability of guessing the starting point of the offset batch query, we feel that this security loss is acceptable when weighed against its advantages which we show in more detail in Chapter 5 on page 51. Our rationale is that a follower relies on this intractability of gaining information about any single query no matter what batching method is used. Or in reverse, it is just as bad if our server gains information about only a single query as it is if it gains information about a whole batch of queries. Either way, the privacy of the follower is violated, so
our system relies on the single query security no matter what batch method is used.

3.4.2 Reducing computation

Even with the offset optimization of the previous section, the server has to process the whole database \(k\) times for a total computational complexity proportional to \(k \cdot n\). Ishai and Kushilevitz [23] offer a solution to amortize the server-side computation. The user randomly chooses a hash function \(h : [n] \rightarrow [k]\) from an appropriate family that provides a uniform distribution of elements. It sends this hash function to the server who can now partition the database into \(k\) buckets. Obviously the expected size of each bucket is \(\frac{n}{k}\) and ideally we would want that of the \(k\) elements we wish to retrieve there is exactly 1 in each bucket. In actuality we can only say that the number of these elements of interest in any bucket exceeds \(\sigma \log k\) only with probability \(2^{-\Omega(\sigma)}\). With this information, we can view each bucket as a smaller sub-database and the user can send \(\sigma \log k\) queries to each sub-database. Except for probability \(2^{-\Omega(\sigma)}\), in which case some bucket holds more than \(\sigma \log k\) of the elements of interest, the user will be able to retrieve all \(k\) elements.

Note that the overall server-side work has been reduced to \((\sigma \log k \cdot \frac{n}{k}) \cdot k = \sigma \log k \cdot n\) which for all but the smallest \(k\) is better than the trivial solution of \(k \cdot n\). In return the server communication increases to \(\sigma \log k \cdot k\) versus just \(k\). Similarly to the square matrix PIR optimization we can mitigate this by sending recursive queries to only retrieve the elements we really want from the \(\sigma \log k\) \(k\)-sized intermediate databases (cf. Figure 3.4) As a matter of fact there are quite a few similarities to square matrix PIR which we will discuss in more detail in Section 4.2.2 on page 42 of the implementation.

The main problem with this method is the fact that we no longer have perfect correctness. With a certain error probability the user will not be able to retrieve all elements. This could be mitigated by repeatedly choosing the random hash function until we are sure that the elements of interest are distributed as expected, but then we lose perfect (computational) privacy because the hash function is dependent on the indices we want to retrieve. Ishai et al. [23] propose the use of specially crafted batch codes that eliminate the error probability. Unfortunately we did not have time to implement a batch code based version. Instead, as we detail in Section 4.2.2 on page 42, we have implemented the imperfect privacy method that guarantees perfect correctness as an optional feature of our system, mainly so we are able to evaluate its efficiency.
3.5. Database indexing

Interestingly, one thing that is very rarely considered by works on Private Information Retrieval is how a user obtains knowledge about the database index of the element he wants to retrieve. Many authors assume a static, publicly known database. In our scenario this is obviously not the case with ever-changing databases and a multitude of elements. During a PIR run, elements are expected to be indexed continually in increasing order so that the database elements and the query vector “match up”. Because our database is dynamic, it is non-trivial to make these indices available to the followers so they can generate their queries accordingly. At first we would like to draw attention to the fact that the problem of finding the correct database index is in itself a PIR problem. Assume the follower has some other, independent descriptor for a file, e.g. the username of the tweeter he follows. The private index retrieval (PIndR) problem refers to how a follower queries a meta-database of (descriptor, database-index) pairs without the server gain-
3. Design

The obvious difference to traditional PIR is that the follower does not have a numerical database index to query this meta-database, only the descriptor.

3.5.1 Private index retrieval solutions

In the following we discuss several solutions that we have considered for PIndR.

Trivial PIndR

Much like with traditional PIR, the naive and obvious way of retrieving the database index is to send the full meta-database, i.e. send a list of descriptors and their associated numeric database index. The user can then peruse this list and pick the database index of interest. If \( l \) is the (maximum) length of descriptors and we assume the database is small enough to fit its indices into a 32 bit integer, the communication complexity of this is \( O(n \cdot (l + 32)) \). Obviously this can be considered to be better than trivial PIR with much larger files, but it is still not very scalable to large databases.

Minimal perfect hashing

A minimal perfect hash function is given a predefined input domain and is able to hash it into a range of consecutive integers without collisions. Such a function could be used so that both server and follower can derive a database index based on descriptors. The issue is that such a hash function would need to be recalculated and resent to followers every time the database changes and is linear in the database size as well, albeit with a much smaller constant [6] than trivial PIndR.

OPE-based keyword search

Inspired by Freedman et al.’s paper on keyword search [17] we have considered using oblivious polynomial evaluation (OPE) such that descriptors can directly be used to query the meta-database. With this method, the server uses a public hash function to (evenly) distribute the meta-database elements into \( L \) buckets. For each bucket \( i \), the server generates a polynomial \( P_i \), e.g. with Lagrange interpolation, such that \( P_i(x_j) = p_j|0^q \) for all descriptor-index pairs \((x_j, p_j)\) mapped to that bucket. \( 0^q \) functions as a statistical security string so that we can detect if a bucket even contains the descriptor we queried. Note that the degree of these polynomials is equal or less than \( m = \lceil \frac{n}{L} \rceil - 1 \), assuming an even hash function.

A follower can now send homomorphic encryptions of the \( m \) powers of the descriptor \( x \) he is interested in, i.e. \( E(x), E(x^2), \ldots, E(x^m) \). Because polynomials are linear combinations with respect to given powers, the server can
3.5. Database indexing

use the homomorphic properties of the encryption to obliviously evaluate the follower query for all $L$ bucket polynomials. It either sends all $L$ results back to the follower, or, similarly to the square matrix PIR optimization, the follower uses a recursive PIR query to only retrieve the result of the appropriate bucket (remember that he knows that index because the hash function is public). This scheme requires only $O(2^m \cdot k)$ follower communication for security parameter $k$ and $O(k)$ server communication. Note that in normal OPE, the server would further "mask" his polynomial such that the follower does not gain any information besides the evaluation at $x$. We do not require this kind of server privacy. Furthermore we note that using OPE to directly retrieve our tweeter files by keyword is inefficient because we would need to use polynomials over very large fields such that they can encode the tweeter files.

The drawback, similarly to minimal perfect hashing, is that the server needs to interpolate the polynomials each time the database changes. While there do appear to be some parallel algorithms to do so [4, 34], we have not found any based on MapReduce which would be nicely incorporable into our design so far. Additionally it needs to evaluate all polynomials for a follower query, resulting in $n$ exponentiations for an average homomorphic cryptosystem.

3.5.2 Our solution: Static indexing

In this rare case, we make a distinction between the per-tweeter and per-hashtag database.

Per-tweeter database

Because we cluster tweets into per-tweeter files, as described in Section 3.3.1 on page 22, we can keep indexing somewhat simple. All users are assigned numeric indices. These indices are directly applied to the respective tweeter files in the database as well. As we have mentioned before, usually a database will use continuous integers to index its elements during a PIR run and match it to queries. In our case database elements will always have the static index of the tweeter; even when certain elements are not present because some tweeters have not tweeted anything, the element will keep its index. While this may be seen as inefficient for sparse databases, it must be remembered that the PIndR problem in itself is also hard to solve efficiently. Note also, that while the follower may have to send "unnecessarily" large queries, he does so rarely, as we elaborate on in Section 3.6 on page 33, and therefore needs to account for denser databases that might occur later on as well (cf. figure 3.5 on the next page). Also, the server does not incur more direct work during PIR because the query elements referring to nonexistent database elements are simply ignored.
3. Design

Figure 3.5: Per-tweeter database indexing of database elements. Files are indexed according to tweeter user index. Even when the database is sparse and could be densely indexed, the files keep their tweeter user index. A query always assumes a full database, so when it later does fill up the query can properly be matched to the whole database including the “new” files.

Therefore the database indices are just as static as the user indices. Which brings up the question of how to handle changes in the user base. A new user is trivial, since we can simply assign him \( \text{max}(id) + 1 \). Note that existing queries, which only go up to \( \text{max}(id) \), do not need to be updated straight away. During a PIR run we simply do not process files that exceed a query’s limit.

We briefly discuss the privacy implications of this design. As we have stated, PIR inherently needs to touch all database elements, otherwise the database can derive information about the query. We posit that despite this the database does not gain any useful information with this design choice. We denote the problem of the grown database and old query as \( \text{GrownPIR} \) and use an informal reduction from PIR to \( \text{GrownPIR} \). Assume that the server has an algorithm \( A(S, Q, T) \) where \( S \) is the set of database elements prior to growth, \( Q \) is the “outdated” query pertaining to that set and \( T \) is the grown set of elements such that \( S \subset T \). This algorithm is able to extract information that allows the database to guess the retrieved element of \( Q \) with a non-negligible advantage above \( \frac{1}{|S|} \). If such an algorithm exists, a database can always generate random elements \( R \) and set \( T' = S \cup R \), since these are
not subject to input of the user of PIR. This corresponds to a new user being
added to the system, but the database viciously not informing the followers
such that they keep using their old query. Hence, the database can now use
$A(S, Q, T')$ to break PIR privacy. The one piece of information the server
gets is that a follower who does not update his query is not interested in the
new tweeters. Once the follower does update his query, the database is back
to square one, since any of the $|T|$ elements may be retrieved by the new
query.

In summary, using outdated queries does not leak any information about
that old query, nor about any future up-to-date queries.

If a user leaves the service, we have several options to handle the freed up
user index.

1. **Ignore it.** Do not reassign the free index. Simply keep using the next
   highest index. This is straightforward, but makes the database and
   hence the queries unnecessarily large.

2. **Compaction.** Ignore the free indices at first, but at a certain threshold
   reassign all user indices to “compact” the database. This requires that
   all followers are informed so they can update their queries to match
   the new indices.

3. **Free index list.** Keep a list of free indices and assign new users to those
   first before assigning new, high indices. Followers that are subscribed
to the deleted tweeter and not aware of his deletion will receive the
new user’s tweets instead. The service obviously cannot inform them
since it does not know of their subscription. It may broadcast the dele-
tion event though, such that any follower who does have a subscription
can remove it at some independent time, as otherwise that information
is leaked.

Though we currently simply use option 1 because our system does not sup-
port account deletion, we shall describe what we believe to be an efficient
way of handling deletion, based on option 2. The distinct advantage of op-
tion 2 is that the database does actually get smaller and hence queries after
a deletion can be smaller. The main problem is that forcing all followers
to upload an updated query is very inefficient. We propose a specialized
approach with a deletion history.

1. When a user $j$ is deleted, we compact the user base immediately: $i' =
i - 1$ for all $i > j$.

2. We keep a deletion history $H = \{(t, j)\}$ where $t$ is a timestamp of
when the deletion occurred and $j$ is the deleted index. We keep this
history ordered by decreasing timestamp, i.e. the first entry refers to
the newest deletion.
3. Design

3. New queries function as usual; they are based on the compacted database.

4. When the server processes a query it first checks the timestamp $t_q$ of the query. For each file $i$ in the compacted database to which the server intends to apply the query, it restores the old index that is used by the old query:

\[
\text{index} = i \\
\text{for each } h \text{ in } H \\
\quad \text{if } (h.t > t_q) \text{ and } (h.j \leq \text{index}) \\
\quad \quad \text{index} += 1 \\
\quad \text{endif} \\
\text{endfor} \\
\text{return index}
\]

5. If we do not want the history to get out of hand we can set a maximum size and force older queries to get updated; these are probably from inactive followers anyway.

Obtaining user indices

Our static indexing allows us to use user indices as “permanent” subscriptions. This makes any kind of PIndR scheme obsolete since local subscriptions (e.g., a list of user indices) can be directly converted to a PIR query without a need to first get up-to-date database indices. Of course, the problem remains of how a follower obtains the user index of a tweeter in the first place. Because we rather liked the idea of OPE-based PIndR we implemented a small service that uses it for user lookup. This means that a user who has a username of someone he wishes to follow can obliviously ask the server what the corresponding user index is and can then add it to his (local) subscriptions. We note that this is an optional gimmick that we did not optimize and therefore may not be scalable for the reasons mentioned in the OPE discussion. Furthermore, user lookup is used less frequently than in the case of PIndR used for index lookup in a dynamic database. This would have to be used every time the matching process is run to obtain up-to-date indices for all followers.

Without the use of PIndR for user lookup, we envision users to retrieve the index by simply visiting a tweeter’s profile. Obviously the server will be aware that the user has accessed the tweeter’s profile, but it does not know if a future query actually pertains to that tweeter. Cautious users may also obtain the index through a third-party channel like e-mail or a dedicated index lookup service not run by our provider.
3.6. Matching

Per-hashtag database

For tweeter subscriptions, it obviously makes sense that followers can subscribe only to existing tweeters. For hashtags on the other hand, we wish to give the follower the option to subscribe to future hashtags. E.g. we can subscribe to #privacy even if there are currently no tweets in the database containing that hashtag; we simply want to receive any future tweets on that topic. We solve this by having a public encoding function that encodes a given hashtag into a unique index. Basically it simply uses a compact binary representation of the hashtag string as a numerical index. Note that this is similar to the minimal hashing approach but does not require an actual minimal hash function nor any other costly hash function to be sent whenever the database changes, because we are not concerned with continuous indices for existing hashtags only.

The obvious drawback is that a follower needs to generate queries for a database consisting of the entire hashtag space, otherwise the server can derive information about the subset of possible hashtags used in the query. For this reason we impose somewhat strict limitations on our hashtag system: we currently only allow lowercase ASCII letters and numbers to be used for short hashtags. This limitation allows us to tightly encode them and keep the hashtag space reasonably small. Note that queries which are larger than the actual database pose no problem since elements that exceed the maximum database index simply will not be applied.

3.6 Matching

To reiterate, matching refers to the process of using the subscriptions of followers to deliver the appropriate tweets to each one. In our case the subscriptions are PIR queries and hence are not in the clear. We have considered several ways of doing this in a timely manner.

3.6.1 Online matching

A follower who wants to update his news feed generates a PIR query that matches his subscriptions and the current database size and sends it to the server. The server runs PIR and returns the results. It should be easy to see that for any decently sized database, this is absolutely infeasible, unless the follower likes to wait a considerable amount of time for an answer which is unthinkable in a webpage use case. Besides the fact that query generation itself is somewhat slow, the server-side PIR process is obviously much slower than retrieving the tweets with a plain text subscription as on Twitter. Just the fact that the server needs to touch all database elements makes this inferior to plain text matching, not to mention the additional overhead of cryptographic operations. Additionally synchronization becomes an issue.
since the database should not change (at least in terms of indices) between query sending and PIR run. The final proverbial nail in the coffin is the fact that a multitude of followers may request an update in a small interval, forcing the server to do many PIR runs in parallel. Clearly it would get overloaded very quickly.

3.6.2 Semi-online matching

In this protocol, the server periodically runs PIR matching based on follower queries it has received in the interval. It consists of two server phases.

**Write phase.** In this phase tweeters can post new tweets which immediately get added to the database. Any requests to get a news feed update are rejected or answered by cached results.

**Read phase.** The server locks down the database. It caches any newly posted tweets and commits them in the next write phase. At the start of this phase it informs all followers (either by push or pull event model) that it has entered the read phase and requests new, up-to-date subscription queries. Followers have the chance to upload their queries for a certain preamble phase. After the preamble the server runs one or more PIR processes and sends the results back to the followers. It then goes back to the write phase.

This method handles synchronization issues in a straightforward way and avoids overloading by collecting queries first and then retrieving results in batch. A drawback of this protocol is that followers can no longer use the regular webpage model. At the very least they need to run a background program which automatically sends queries in the preamble of the read phase. Furthermore, if a follower is offline or does not have the background program turned on, he will miss the read phase and will have to wait until the next write phase is over. In essence, we no longer have a strict server-client model because the server initiates the preamble protocol on its own and followers have to react to it.

3.6.3 Our solution: Offline matching

We have seen two instances of online matching. Both rely on interactive protocols before a matching run is commenced. This means that each time the follower is required to generate a query. While we wish to keep query generation fast by choosing appropriate cryptosystems and keeping the database small, it will always be a rather large overhead for the follower. It is regrettable that a follower would have to send a new query even though his subscriptions have not changed, simply because the database has been altered slightly.

We have a distinct advantage, though. Because we have a system with static element indexing (see Section 3.5 on page 27), we do not have to update
3.7. Summary

queries when the database changes. A query can be reused indefinitely because the set of elements it refers to remains static. Elements might get deleted, but the remaining elements keep their original indices. New elements always get appended to the “end” of the database. An “outdated” query $Q$ is simply applied only to the elements which it refers to, i.e. the first $|Q|$ elements. We have already shown that this does not break privacy in Section 3.5.2 on page 29. Trivially, repeatedly using a PIR query also does not affect privacy, since an honest-but-curious server may always do that outside of the defined “single-use” PIR protocols which it may need to respect. The only additional information that the server can retrieve over an online PIR protocol is “timing information”, i.e. that until the follower updates the query, he is not interested in any of the new elements, and that the follower’s interests have changed whenever he uploads a new query.

Due to the previous design choices, which, admittedly, have been geared towards this conclusion, we can easily implement an offline matching system. Followers only upload queries when their subscriptions change. The server stores all queries and periodically runs a PIR matching to obtain results. These results, which of course are encrypted, are also stored on the server. Whenever a follower requests his news feed he is served these stored results which are up-to-date to within the interval at which the server runs the matching. Clearly, our system will not be able to offer the same real-time information as a plain text microblogging system. As we have seen in the other matching solutions, this is not possible in any way since PIR is just not as responsive as plain text database querying.

Advantages

- Infrequent query generation and upload.
- Controlled server load (in terms of PIR runs).
- No user interaction needed for matching.
- Fast result retrieval due to caching.

Disadvantages

- Some loss of “real-time feel”.
- Server gains “timing” information.
- Dependent on static database indexing.

3.7 Summary

We briefly summarize the design choices we have made for our system such that the reader may quickly refresh his memory and keep them in mind for the remaining chapters.
3. Design

- **MapReduce-based PIR.** We use a PIR scheme that works well with the MapReduce paradigm of scalable parallelization and was suggested by and slightly adapted from [29]. Highlights are efficient treatment of files by splitting into blocks that “fit” into query encryptions and use of a somewhat homomorphic cryptosystem with efficient operations.

- **Per-tweeter database.** In our database each tweeter (and each hashtag) is represented by a file containing his tweets. A follower always retrieves all the tweets of a tweeter he follows.

- **Two redundant databases.** Due to the previous point, there are two redundant databases, one arranged per tweeter and one per hashtag.

- **Windowing.** Each file has a capacity. If it is surpassed old tweets are discarded. Additionally we also regularly discard very old tweets regardless of capacity.

- **Offset batch queries.** We save communication by “re-using” a PIR query for multiple element retrievals by also sending the respective offsets to the initial element such that the database can be temporarily rearranged and the initial query re-used.

- **Hash partitioning for batch queries.** We amortize server computation by partitioning the database such that a batch of elements can be retrieved with less computation. This is an optional feature we use because there is a trade-off between privacy and correctness, but it can be made perfect with batch codes.

- **Static indexing.** We make sure our files are statically indexed such that changes in the database do not invalidate existing queries.

- **Offline matching.** The server does all matching at certain intervals, using cached follower queries instead of interactively requesting new queries each time. It relies on many of the previous choices for this to be feasible.
Chapter 4

Implementation

After describing the major design choices for our system in the previous chapter, we will now talk about the implementation of the system. We will give a broad overview of the whole system and then step by step go into details that we believe to be important with regard to efficiency and usability.

4.1 Overview

In Figure 4.1 on the following page we provide a broad overview of all the components that make up our microblogging system.

- **Hadoop cluster.** This is the Hadoop framework (cf. Section 2.3.1 on page 13) running on a dedicated data processing cloud. Our database(s) of tweets is stored on its distributed file system for fast access and our offline PIR matching will be run on it.

- **Daemon.** This is the heart-piece of our system. The daemon is responsible for coordinating all the aspects of running the microblogging architecture. It commits tweets to the database, schedules PIR runs to match tweets, commits new follower queries, serves the matching results and more.

- **Web front-end.** This is a collection of standard web tools that provide standardized access to our system for users. Additionally the SQL database is of some importance to efficiently store meta-data required by the daemon.

- **Client.** Users can interact with the microblogging system with a standard browser. Followers use a browser extension that uses a back-end system which provides encryption and decryption services.

We will now give a top-down detailed view of the individual components.
4. Implementation

Figure 4.1: Overview of all the parts of the microblogging system.

4.2 Hadoop cluster

This is a standard cloud computing cluster running the Hadoop framework. In particular it offers access to the distributed file system HDFS and we can schedule arbitrary MapReduce jobs. We use HDFS to store our tweet databases in the manner we described in Section 3.3.1 on page 22. This allows us to directly leverage the distributed system during our PIR MapReduce jobs which rely on the data locality provided by HDFS.

4.2.1 HDFS limitations

As we have hinted at previously, HDFS has a few important limitations that we need to work around.

Speed

Obviously a distributed file system is slower than accessing a local file system since it potentially needs to access a remote file system on one of the machines in the cloud. We try to amortize write times and HDFS load by
having the daemon first store any incoming tweets in a temporary SQL table we call *tweet cache*. At a certain threshold, which needs to be determined depending on HDFS performance, it will start a write batch job and write all the tweets in the tweet cache into HDFS. Our aim is to smooth out the load by not making it directly dependent on how our tweeters tweet.

Read times are important when we serve matching results to a follower. A MapReduce job writes its output to the HDFS. Instead of directly copying that output to local storage and then serving it from there, we have decided to leave it on the HDFS and retrieve it from there when a follower needs it. The reason for this is that copying all outputs of a matching job will further increase the overall match phase which we wish to keep short for a better “real-time” feel. Instead, as we will see in the client details, we have a system of last-modified descriptors such that a follower who repeatedly requests the matching output will generally only get it once, unless it has changed, and stores it on his machine. This means that in each interval between two matching runs, a follower will only retrieve the output once from the HDFS. Note, of course, that retrieving it more than once in the interval is useless anyway since it does not change.

**File immutability**

Files stored on the HDFS are considered immutable once they have been completely written. This means that we cannot modify parts of a file except by rereading the whole thing, changing the necessary parts, and rewriting the new version to the HDFS. With newer Hadoop versions, it is at least possible to *append* files such that we can add new tweets to a tweeter file. This limitation is highly relevant for our system because windowing (Section 3.3.2 on page 23) requires that we discard old tweets in a file. Rewriting the whole file may be very inefficient depending on size. On the other extreme, we could store each tweet in a separate file to easily delete tweets, but as we have discussed, this is too much of a load on the file system and MapReduce. We use a pragmatic trick to make windowing possible without expensive file system operations.

1. When a tweeter file reaches its capacity (or is deemed “too old”), we rename it to mark it as old. Note that renaming is simply a matter of changing an entry in the meta-data table describing all files in the HDFS, which is stored centrally on what Hadoop calls the *name node*.

2. We start a new file (with the original filename) and from now on commit new tweets into this file. Note that we delete any previous old files such that there are only ever two “versions” of a file - the current file and the old file.

The reason we do not just delete the full file and start a new one is that we do not wish for such an abrupt change in the database. Assume a tweeter
breaks the capacity $c$ of his tweet file and we delete the full file and start a new one with the tweet that broke the capacity. Any follower will now only be able to see this new tweet and none of the older ones. This abrupt change from $c$ tweets to 1 tweet is not very user-friendly; for example when a follower has not checked his tweets in a while and only gets this single tweet now. In essence we want to maintain the *sliding window* that we could use in a fully mutable database.

We now have a maximum of two files per tweeter that combined contain anywhere between 1 and $2c$ tweets. There is an issue with this setup though.

We take this opportunity to briefly describe an important detail of HDFS. The basic unit that HDFS uses is a *data block*, this is a block of data that may contain more than one file or may contain only one part of a large file. A typical data block size would be 128MB. One data block is always stored on a single machine, but multiple blocks that may all contain parts of the same file or just multiple individual files, may be stored on different machines in the cluster. MapReduce by default uses this architecture to distribute the workload *by data block* to different mappers such that a mapper tends to get data blocks stored locally on the machine it is running on.

Our PIR scheme based on PIRMAP (cf. Section 3.2.1 on page 17) divides a database file into its own blocks and specifically relies on the block order such that blocks with the same index can be combined by reducers. This means that splitting up files by HDFS data block, like MapReduce may do, is not an option for us, since we need to have a full view of a file to determine block indices. For individual files it is relatively easy to configure a MapReduce job to give the full file to a mapper. Because our tweet files are not excessively large, we expect them to be in one, or at worst two, HDFS data blocks. This means it does not have an adverse effect on the data locality principle because most of our files are completely present in a HDFS block anyway.

In our system of 2 files per tweeter on the other hand, we need to return both files to a follower in a single PIR run. This means that they effectively need to be treated as a single database file where the new tweet file is appended to the corresponding old tweet file. But since the new file is written at an arbitrary time after the old file, we have no guarantee that they exist in the same local HDFS block. It is not efficient to force MapReduce to give *each* of the many file pairs to a single mapper; in fact we feel this is a severe hit to the concept of data locality. We solve this by *virtually* combining the files in such a way that two individual mappers can work on either file but the result is a concatenated file (cf. Figure 4.2 on the next page) The trick is simply to write the size of the old file at the start of the new file. This way a mapper can use it to calculate a block index offset and shift all the blocks of the new file it processes. When the blocks are written to the final output,
ordered by block index, they will make up the full concatenated file. The old file length could also be read on the fly by each mapper, but this would result in many “unnecessary” name node requests, whereas our method only reads file length when the new file is generated and the mappers read it directly during their normal file processing.

### 4.2.2 PIR

We describe some important details concerning the implementation of our PIR scheme in the Hadoop framework. We begin by listing a few important concepts that Hadoop MapReduce uses.

- **Input split.** This is a block of data that a single mapper works on. It often, but not necessarily, corresponds to a HDFS data block.
4. Implementation

- **Input format.** This family of Java classes is responsible for converting the entire input data of a MapReduce job into separate input splits that get distributed to mappers. The most commonly used, FileInputFormat, simply works on HDFS files and splits them up to best suit data locality.

- **Record reader.** A record reader is run by each mapper to process the input split it has received. It will interpret the raw byte data of an input split and convert it into a key-value pair which is then processed by the map function. This is obviously done in a streaming fashion; a record reader produces a key-value pair, hands it to the map function, and then produces the next pair.

- **Keys and values.** Keys and values used by MapReduce are serializable Java classes and, like the previous classes, can be subclassed to store application-specific key/value data.

**Input processing**

For our PIR MapReduce job, we have implemented specific subclasses. We use a special input format that handles the block offsetting for file pairs, as described in the previous section. Our record reader reads from a full tweet file and directly splits it into the $k$-bit blocks that our scheme requires. In particular, we do not actually parse any content of the tweet files, we always work on byte blocks. This makes our scheme extendable, for example to allow encrypted tweets, because it is independent of file content and format. Our record reader hands these byte block values to the map function, together with a key containing the block index as well as the file index. The file index is extracted from the file name and is needed to find the appropriate element of the PIR query.

**Map and reduce**

Our map functions process one $k$-bit block of tweet data at a time. Each block is subject to all sub-queries of all queries of all followers; we answer all the queries in the system in batch. The map function has knowledge of all PIR queries that need to be answered, since all queries need to be applied to all files and all their blocks. This means that we need to distribute the set of queries to all mappers. This is a potential bottleneck in our system, since this setup of transmitting it to all mappers in the cloud may become slow for very large amounts of queries. We do not think there is a different way of doing this, because, as said, all queries apply to all files. The only other option would be to also split up the queries according to what files a mapper gets, which we deemed similarly costly.
4.2 Hadoop cluster

The actual PIR calculations in the map and reduce functions are done in a very generic way that accommodates the 3 basic ways of doing our PIR: linear database, square matrix database (cf. Section 3.2.1) and hash buckets (cf. Section 3.4.2). We generalize these into the problem of retrieving multiple database elements when the database is arranged into an *arbitrary* \( r \times c \) matrix with \( r \cdot c \geq n \). Ignoring the specifics of each way, we say that our generic scheme first compacts this matrix into \( r' \) vectors of size \( c \), i.e. it compacts it into multiple rows depending on the query. Our generic scheme continues by further reducing this intermediate \( r' \times c \) matrix by retrieving an arbitrary number of elements from each row with a recursive query.

We show how the schemes we intend to use instantiate this generic scheme (cf. Figure 4.3 on the following page). Let \( k \) be the number of elements a (batch-)query wants to retrieve.

- **Linear PIR.** The initial query works on a \( n \times 1 \) matrix/vector and generates \( k \times 1 \) intermediate results. The recursive query simply takes each of the \( k \) results since we do not actually have to reduce any further.

- **Square matrix PIR.** The initial query works on a \( \sqrt{n} \times \sqrt{n} \) matrix and reduces it to \( k \times \sqrt{n} \) intermediate results. The recursive query extracts one element from each row of this \( k \times \sqrt{n} \) intermediate matrix.

- **Hash bucket PIR.** We first would like to point out the relation to the square matrix PIR scheme. Hash bucketing also arranges the database into a \( \frac{n}{k} \times k \) matrix. In particular we will actually not use \( k \) to determine the number of buckets (or columns), but rather \( \max(k, \sqrt{n}) \). This makes sure that we have a “balanced” matrix. Otherwise the matrix would degenerate into a vector for \( k = 1 \) and we would have the linear communication complexity of linear PIR. Now let us assume that \( k \leq \sqrt{n} \), i.e. our hash bucketing produces a \( \sqrt{n} \times \sqrt{n} \) matrix just like the square matrix PIR. We would like to note that the defining difference to square matrix PIR is that we distribute the elements differently. The “issue” with square matrix PIR is the fact that we need to apply all \( k \) queries to all columns because we have no guarantees that a single column does not hold all \( k \) elements of interest. This is the main advantage of hash bucketing: we are able to get guarantees about the distribution and use less queries per column. Hash bucket PIR therefore compacts the initial database matrix into a \( \sigma \log k \times \max(k, \sqrt{n}) \) intermediate matrix. The recursive query extracts up to \( \max(k, \sqrt{n}) \) elements (if the elements of interest happened to be distributed perfectly) from each intermediate row.

Note that we still only need one vector of encrypted elements for the initial query and one for the recursive query, no matter the instantiation. All
sub-queries are handled by appropriately categorizing the batch offsets described in Section 3.4.1 on page 24. The privacy implications discussed in that section apply equally in this generic view.

Using this generic view of our schemes, we can implement all three with a single query class which provides the necessary query vectors, offsets and information about how to arrange the database and distribute the elements, as well as single mapper and reducer classes to handle this query. In fact, different users may choose different ways of doing it simply by sending a different query. Of course in a deployed system this would preferably be restricted to a specific scheme.

**Miscellaneous**

We talk about a few miscellaneous details of our MapReduce implementation:
4.3. Daemon

- **Database files need not be of same length.** Assuming the largest file gets split into \( m \) blocks, it does not matter if another file gets split into fewer than \( m \) blocks. Blocks are "collected" by index and then summarized using the appropriate cryptosystem operation. The absence of blocks does not change the result. If our smaller file is the one the follower wants to retrieve, then the blocks of larger files will be "zeroed out" by the PIR query and end up as natural zero padding in the final result. There is no need for explicit zero padding.

- **Output for a follower is split into multiple files.** This is an inherent MapReduce property. Since there can be multiple reducers combining parts of the output, they each write to their own file. We can relatively easily combine these separate output files on the fly because the partitioning among reducers is deterministic and all outputs are ordered by criteria that we can choose. This way we can simply go through each of the output files in a specific manner and retrieve the full, correctly ordered output with almost the same computational and memory complexity as concatenating the output files.

- **Small files can be combined.** Hadoop provides a special extendable input format that will attempt to combine multiple small files that are "close" in terms of data locality such that a mapper will work on more than one file. This is very useful for our system since we generally have many files that are smaller than a HDFS data block and having one mapper per file would be inefficient since the short map time would be overshadowed by the setup overhead for the mapper. The disadvantage is a slightly longer setup time to calculate good combinations. Note that this does not affect database indexing, since files are only combined virtually and a mapper can still extract separate indices from the list of small files it works on.

4.3 Daemon

This is the centerpiece of the server-side system as it coordinates the various components and data flows between the cluster and the users.

4.3.1 Tweet management

When a tweeter posts a tweet, it is first written into a regular SQL table called the *tweet cache*. We have already briefly discussed it in the previous section. To reiterate, we use this additional stop so that the daemon can control the load on the cluster and its distributed file system. The daemon will commit the tweet cache to the distributed system whenever a configurable threshold is reached and before every PIR run so that the database is up-to-date. When the daemon commits tweets, it also checks the windowing constraints we
4. Implementation

have established during the design. It keeps track of the current number of
tweets for each tweeter (and hashtag) in SQL tables. If committing a tweet
would break the capacity of a tweeter file, the daemon discards tweets in the
manner we have described in Section 4.2.1 on page 39. The reason we use
SQL tables for capacity tracking is that this is much faster than reading and
counting tweets in the HDFS. Periodically, the daemon will discard all tweet
files that have not been modified in a certain time span.

4.3.2 Subscription management

Followers can update their subscriptions by uploading a new PIR query
(which contains information to retrieve tweets for all subscriptions). The
daemon simply commits the query to the appropriate location in the HDFS.
There is no caching for this, with the assumption that subscriptions are up-
dated less frequently than tweets posted and because caching queries re-
quires much more storage than tweets.

4.3.3 Matching

At regular intervals, the daemon will initiate a PIR matching run by schedul-
ing a MapReduce job on the Hadoop cluster. The daemon makes sure that
during a PIR matching run the HDFS database is not modified. This is nec-
essary for consistency and so the cluster does not suffer from unnecessary
load during a PIR run. The daemon rejects any requests to read the match-
ing output or to write query requests during a PIR run. Conversely, it will
not start a PIR run when reads or writes are still in progress and lets them
finish.

4.3.4 Result retrieval

Followers can retrieve the output generated by the last MapReduce PIR
matching run. As we have detailed in Section 4.2.2 on page 44 the daemon
will combine the output from several reducer outputs on the fly and directly
from the HDFS. As we have explained in the aforementioned section, this
can be done with little overhead due to how the individual output parts are
structured.

The daemon maintains a timestamp of the last matching run and expects
a timestamp with each retrieval request. This timestamp is supplied by
the client program of the follower and pertains to the last time the client
has successfully retrieved the output. If the matching run timestamp is not
newer, then the client has already retrieved the most recent output and the
daemon will not bother to send it again. Hence the daemon only needs to
retrieve any single matching run output a single time from the HDFS.
4.3.5 User lookup

In Section 3.5.2 on page 32 we mentioned that we implemented a version of OPE as a means to provide oblivious retrieval of user indices by username. As mentioned in that section we did not focus on making it particularly efficient since it is just an optional feature, but we describe some details here.

**Statistical security string.** The polynomial encoding of the (name, index) pairs requires a statistical security string such that we can determine if a result pertains to an actual pair or is an interpolated value that does not exist in the user base. In the paper that inspired this method [17] the authors suggest a string of 0’s. As it turns out there are some issues with this when user names are very similar in a continuous interval since the interpolation will actually generate values that maintain this 0-string. We solve this by using a MD5 hash of the username and using its lower order bits as the statistical string. This way we have a better value distribution that “breaks” the interpolation precision of nonexistent names. Alternatively, hashing the names prior to polynomial encoding would also have been an option to get a more uniform distribution.

**Encryption size.** All bucket polynomials are over the same field $F$. The encryptions of powers obviously need to fit $k = \log_2 |F|$ bits. The problem is that we also need to account for the intermediate result of polynomial evaluation. We homomorphically evaluate a polynomial, the user decrypts it and then applies the polynomial group modulus to retrieve the final value. This means that the cryptosystem must not “wrap” the evaluation result in its own group but maintain it until it is decrypted, since we cannot homomorphically apply the modulus. This means that, similar to the somewhat homomorphic cryptosystem of Trostle&Parrish, we need to adjust the cryptosystem to support $2k + \log_2 m$ bits, where $m$ is the (maximum) degree of the bucket polynomials. This is independent of whether our cryptosystem is only somewhat homomorphic or not.

4.4 Web front-end

Our daemon exposes the services previously listed through a custom TCP protocol. Theoretically clients could directly connect to the TCP server maintained by the daemon and communicate with it. Instead we chose to add another layer to the interaction with a dedicated HTTP front-end.

The web front-end consists of an Apache httpd (http://httpd.apache.org) server which serves the more conventional content to the users. It serves regular webpages like user profile and news feed pages. It handles posting tweets with regular techniques, i.e. HTML forms and HTTP POST, and
writes them into the tweet cache in the SQL database. It also writes them into a regular SQL table which is used to display their own tweets to tweeters in a non-private and efficient way.

Additionally we use an Apache Tomcat server (http://tomcat.apache.org) which handles the direct communication with the daemon. It handles the follower requests for results or uploading queries by connecting to the daemon over our TCP protocol, forwarding the necessary data from the HTTP protocol, retrieving a response from the daemon and translating it into a HTTP-appropriate response. One can think of it as a proxy between the users and the daemon that translates requests and responses between different protocols. Essentially the TCP communication is just a means of inter-process or local network communication between the daemon and the Tomcat server.

The main reason we do not use direct TCP connections to the daemon is that HTTP is obviously much more prevalent. We can keep the internals of our TCP protocol on the server side and in particular can modify it without requiring updated user software. Additionally, HTTP offers many features we can easily enable, like SSL/TSL and standard HTTP authentication. With our additional HTTP layer we can relatively seamlessly enable these features.

4.5 Client

Users will access and interact with our system through regular webpages provided by the previously discussed web front-end. Of course, tweets in a follower’s news feed are sent in encrypted form. Additionally the generation of PIR queries for a follower’s subscriptions is not easily handled with a normal browser. Furthermore the follower must store his interests (in the clear) locally such that he can easily update his PIR queries.

All these things are handled by a dedicated Java back-end. It keeps track of a follower’s interests using a SQLite database (http://www.sqlite.org/). With these it provides a service to generate the appropriate PIR query to retrieve all interests privately from our server. It generates the appropriate keys and stores the private key locally. When a follower reads his news feed, the back-end retrieves and decrypts the newest PIR results from the server. Note the fact that currently the back-end itself connects to the server and retrieves the results. This could also be done via the browser (for example in a hidden Base64-encoded string in the webpage code), as long as the back-end provides the timestamp of the last successful retrieval, which we have discussed in Section 4.3.4 on page 46. Because the server may respond with a message akin to “tweets not modified” if the timestamp is too new, the back-end saves a certain number of tweets in a small SQLite table such that it can always serve the latest tweets. This is also useful during PIR runs.
when the server will respond with “busy” since it does not want to access the HDFS database (cf. Section 4.3.3 on page 46). Note, of course, that these local tweets are already decrypted and do not cause a big storage overhead.

All of these back-end actions are triggered by a Firefox extension which communicates with the back-end at the appropriate times, depending on the webpage the user is on. The extension will update the webpage with the back-end results for a seamless user experience. The only hassle is that of starting the back-end because changes in recent Firefox versions have disallowed the integrated execution of Java (e.g. with Java LiveConnect) so instead it needs to communicate with a running back-end via TCP. For a schematic view of the client architecture see Figure 4.4.

4.6 Cryptosystems

As we have mentioned we use the somewhat homomorphic cryptosystem of Trostle and Parrish [39] based on the hidden trapdoor group assumption. Since this is a very specific cryptosystem, which in the instantiation suggested by the authors only works for PIR applications, we did not find any dedicated libraries. Instead we implemented it using Java BigInteger arithmetic. We took the opportunity to also implement Paillier’s [33] and the Damgård-Jurik [14] cryptosystem using the same class design as a point of comparison. We acknowledge that there is obviously room for performance optimization with dedicated libraries based on more efficient data structures than Java BigInteger, but for our tests and evaluations it will suffice. The advantage of having our own little library of cryptosystems is that we can make
the code generic enough such that any homomorphic cryptosystem can easily and interchangeably be used. As a matter of fact a follower may choose which cryptosystem he wants to use for a query. Again, in a deployed system this would have to be restricted to ensure optimal performance.
Chapter 5

Evaluation

In this chapter we evaluate the implementation of our system. The goal is to see how it behaves with different parameters such as database size or distribution. We pay special attention to how our design choices and implementation optimizations affect its performance. Furthermore we try to gain more information about how it interacts with MapReduce.

5.1 Philosophy and preliminaries

While our system has an important support system in the daemon and web front-end, the most important part is obviously the PIR matching. Hence this is what we focus our attention on. We test how our MapReduce-based PIR method behaves and scales with varying settings. We test the tweeter database only and disregard the hashtag database because the PIR method is exactly the same for both databases and it is more straightforward to modify the tweeter database.

We prepared various test phases in which we alter one or two system parameters only. All other parameters are set to a baseline which we have determined at the beginning. This baseline is purposely humble such that tests run quickly. In the test phases we usually increase the tested parameters until the system gets reasonably slow. Note that at this point some of the tested parameters are unrealistic with respect to an actual real life system, but this is done on purpose since it is the best way to get a clear picture of how they affect the system. In particular it gives us a feeling for the upper bounds on these parameters for a deployed system.

In the following sections we show many plots with empirical data. This data was usually derived by running the same test round for multiple runs. The number of runs is generally low due to time constraints, especially for the later test rounds which take long. In all plots with empirical data we provide
5. Evaluation

error bars using the standard error of the samples at each data point. We feel that our test cluster was stable enough to provide results with a low deviation. Relative standard deviation is below 2% on average with a peak value of 8%.

Unless otherwise specified, tweets in this testing system are always exactly 140 characters long and are saved with UTF-8 encoding.

5.2 Setup

We rented nodes from Amazon’s Elastic Compute Cloud (Amazon EC2). In particular we make use of their Elastic MapReduce (EMR) layer which allows us to easily run arbitrary MapReduce jobs on rented cloud nodes. Amazon limits the maximum number of nodes on EMR to 20. There are different types of nodes, or instances as Amazon calls them, with different specs and prices [2]. Predicting that the biggest load will be on memory due to the sheer amount of data that needs to be handled and in particular shuffled, we chose to pick a moderately priced 10 “High-Memory Double Extra Large” instances. Note that we believe this to be on the humble side of a cloud cluster and there is definitely room for improvement given the necessary budget in a real life deployed system. As it is, 10 instances of high memory nodes should give a good impression on how the system behaves and scales in a real cloud setting.

5.3 Comparison to trivial PIR

We start out by comparing our system to trivial PIR. Obviously we would like it to be faster than just downloading the entire database. To this end we test matching time for a single follower with a single subscription. We would like to repeat that our system is specifically intended to handle multiple followers with multiple subscriptions. This will be tested later, but for this test this is the most straightforward method.

In Figure 5.1 on the facing page we plot the matching time versus the hypothetical time of downloading the entire database, i.e. trivial PIR. Both axes are in logarithmic scale. We start at 32,000 tweeters in the database, each having 100 tweets in his file. This results in a database of roughly 520MB. For each additional data point we double the number of tweeters and hence the database size. For the hypothetical trivial PIR time we used data from [31] and set the follower’s download speed at 13.76Mbps. For our system, the follower needs to download an average of 225KB in the form of the encrypted matching result. Additionally he needs to upload up to 349KB in the form of the PIR query for the last data point. Both of these processes
5.4 Tweet volume

Figure 5.1: Log-log plot of the matching time for a single follower with a single subscription in a database with increasing number of tweeters with 100 tweets each. Compared to the hypothetical time of downloading the entire database at 13.76Mbps.

amount to a minuscule overhead which we did not add to the time in the plot.

Clearly our system outperforms trivial PIR. Even with the additional overhead of running MapReduce, it is faster to run our PIR method than downloading the entire database.

5.4 Tweet volume

In this test we evaluate the system with respect to the number of tweets stored for each tweeter. This translates to increasing the size of the files in the database. In terms of all parameters, we expected this to be the most “beneficial” to the system because it does not result in increased query size nor a larger number of individual database elements. In fact, the authors of PIRMAP [29] specifically base their efficiency argument on the fact that they deal with a small database of large files. With a single matching run we
answer the requests of 100 followers who each subscribe to a single tweeter, i.e. each follower wishes to retrieve one file. The database consists of files for 1,000 tweeters. We start with 100 tweets per tweeter and double this for each data point.

Figure 5.2 shows a plot of the matching time in log-log scale. It is clear that our system scales favorably for increasing tweet file size. Of interest is the fact that the first 3 data points are dominated by the MapReduce overhead which is why the plot looks non-linear. It bears repeating that in a real system it would be inadvisable to allow tweet files to become this large. Remember that we can limit them with our windowing system. In our opinion it does not make sense to set tweet size to more than a few hundred tweets because no follower would bother to read that many tweets; only the newest tweets are of interest.

Because the test results are, as expected, rather underwhelming in their implications, we take this opportunity to look at a few more details which are significant to our system and to later tests where they play a larger role.
Figure 5.3: Log-log plot of the size of a single tweet file versus the size of the PIR output that the follower needs to download for increasing number of tweets per file.

Figure 5.3 shows a plot of the file size of a single tweet file with increasing number of tweets and the encrypted PIR result size that the follower needs to download. Both grow linearly as expected when doubling the tweets, but clearly the PIR result is larger. It is, in fact, an order of magnitude larger than the tweet file to retrieve. The reason for this is twofold. There is a subtle blowup due to how the system has to pad the file if it does not perfectly fit into blocks of size $k$, where $k$ is the encryption security parameter. The more important factor is the fact that the Trostle&Parrish cipher of a block of size $k$ is larger than $k$ for obvious reasons. In fact, it is roughly 3.4 times larger after PIR computation due to our need to choose a modulus that accommodates all homomorphic operations (cf. Section 3.2.1 on page 18), as well as the fact that the PIR computation is done with integer operations without modulus (which is secret). Because we use a square matrix approach with a recursive PIR query we effectively encrypt the tweet file twice, resulting in a total blowup of $3.4^2$. Note that the single blowup factor 3.4 itself grows with the database size, but only at an (additive) rate of $O(\log_2 \sqrt{n})$ because the encryption needs to accommodate more additions which add a logarithmic amount of bits. This blowup has an effect both on PIR run time as well as download and decryption time of the follower. It also implies that using a higher dimensional hypercube rather than a 2D matrix to decrease query size is counter-productive because each dimension will further blow up the final result. There are much more efficient cryptosystems for this, e.g. the
5. Evaluation

Damgård-Jurik [14] cryptosystem used in Lipmaa’s scheme [27]. Obviously we would then lose the computational efficiency of Trostle&Parrish.

5.5 Tweeter volume

In this test we fix the number of tweets each tweeter has and instead increase the number of tweeters in the database. This is much closer to a realistic view of the system, as we would fix the maximum number of tweets with our windowing method and only the tweeter base would grow. We fix tweets per tweeter at 100 and again answer the requests of 100 followers who each subscribe to a single tweeter. We start at 1,000 tweeters and double the number at each data point. We took the opportunity to test the effectiveness of two of our optimizations. We compare the matching time to a system that does not combine small files (cf. Section 4.2.2 on page 44) and to a system that uses linear PIR instead of our square matrix approach.

Figure 5.4 shows plots of the matching times of the 3 different systems tested. “Regular” refers to a default system with no deliberate limitations, “No combine” refers to the system not combining small files, and “Linear PIR” refers to the system using linear PIR.

![Figure 5.4: Log-log plot of the matching time for 100 followers with 1 subscription each in a database with increasing number of tweeters with 100 tweets each. Shown are the plots for a regular system, one not combining small files and one using linear instead of matrix PIR.](image)
We first discuss the “No combine” case. Clearly it is significantly less efficient than combining small files. In fact, it was so slow during testing that we did not bother to get all data points since it would only unnecessarily prolong our testing. The main reason for its inefficiency is the fact that Hadoop will spawn one mapper for each file. This means that even at only 1,000 tweeters there need to be 1,000 mappers who each handle a tiny file. The overhead of launching a mapper is too significant to have it only work a few seconds on one tiny file. In comparison we only launch 7 mappers when using the combine method with 1,000 tweeters. The only drawback of combining is the setup overhead of running the combine algorithm. Figure 5.5 shows that this overhead is linear in the database size and may in fact play a larger role as the database gets very large.

![Figure 5.5: Setup time to determine the inputs for each mapper. Comparing the overhead of combining small files versus directly supplying the files to one mapper each. The x-axis is in log scale and the y-axis in linear scale.](image)

Linear PIR outperforms square matrix PIR only when the database is small because it does not incur the overhead of recursively applying PIR. Note that in our generalized implementation (cf. Section 4.2.2 on page 42) the system does actually execute a recursive MapReduce job during linear PIR, but
this is the identity function and incurs only the basic MapReduce overhead which is negligible for larger databases. As we have noted, using linear PIR has the straightforward disadvantage that follower queries grow directly linearly with the database size whereas matrix queries only grow at the square root. The most obvious consequence is that followers need to upload larger queries, but it also has an effect on our PIR run because all the workers need to load these queries so that they can be applied to the respective shares of the database. This is the main reason why square matrix PIR outperforms linear PIR even if we disregard upload times as in these plots. This can be seen in Figure 5.6 which clearly shows that linear PIR incurs a significant overhead in loading the queries. Note that for the larger databases, we start so many workers that not all of them are run at the same time (this is normal for larger MapReduce jobs), which means that this query loading is not done in parallel and therefore applies its overhead multiple times.

Figure 5.6: Plots relating query size to the overhead of loading them.

5.6 Follower volume

In this test we observe how the system’s batch matching behaves as we increase the number of followers with a subscription. We fix the tweet database at 1,000 tweeters with 100 tweets each. We also fix the subscriptions per follower at 1. We start with 100 followers who have a subscription and double the number of followers at each data point.

As seen in Figure 5.7 on the next page the system behaves as expected. As opposed to increasing the database size, increasing the query size not only has an effect on the primary PIR but also on the recursive PIR because more data “gets through” to this second stage. When increasing the number of tweeters, for example, the square matrix only grows proportionally to the
5.7 Subscription volume

Analogous to the last test, we now increase the number of subscriptions each follower has. Again we fix the database at 1,000 tweeters with 100 tweets each. We use only a single follower starting with 100 subscriptions and doubling these at every data point. We only use one follower to get a better understanding of some system specifics pertaining to this test. In particular, we have a close look at our offset batching method (cf. Section 3.4.1) and whether it affects matching time compared to trivial batching, where each subscription gets its own PIR query vector. Obviously trivial batching results in larger queries per follower and thus a higher upload time, but we are
Figure 5.8: Log-log plot of the matching time for a single follower with increasing number of subscriptions when using offset batching versus trivial batching in a database of 1,000 tweeters with 100 tweets each.

interested in the effects on the matching computation.

Figure 5.8 shows a plot of the matching time when using our default offset batch method versus using trivial batching.

Both methods are rather similar in terms of matching time, trivial batching being slightly slower, with the discrepancy being more pronounced for later data points. The main reason for this is, as we have seen in previous sections, the overhead of loading the query on each worker. With trivial batching the query is much larger and takes longer to load, as Figures 5.9 show.

We also used this opportunity to test the optional hash bucketing method of Section 3.4.2 on page 26. Figure 5.10 on the next page shows this method’s matching time compared to the default system using a regular square matrix.

As the reader can see, there are issues with it. It gets slow quickly so that we did not collect all data points, instead opting to use our testing contingent on more important tests. Nonetheless these results have given us an important insight. When we split up the matching time into time required for the
5.7. Subscription volume

(a) Total size of queries.

(b) Average query load time for a worker.

Figure 5.9: Query size and load time when using offset batching and trivial batching as the number of subscriptions of a single follower increases.

Figure 5.10: Log-log plot of the matching time for a single follower with increasing number of subscriptions, comparing the effectiveness of the hash bucket approach versus the default square matrix approach, in a database of 1,000 tweeters with 100 tweets each.
5. Evaluation

![Log-log plot of the matching time of default and hash bucketing PIR, split up into the primary PIR run (PIR1) and the recursive PIR run (PIR2). The solid lines represent the primary PIR of each approach, the dashed lines represent the recursive PIR. 1,000 tweeters with 100 tweets each and 1 follower with increasing number of subscriptions.](image)

The primary PIR and time required for the recursive PIR, as seen in Figure 5.11, it becomes obvious why hash bucketing performs so badly.

The primary PIR is significantly faster when using hash bucketing, as expected due to the decreased amount of computation required. But recursive PIR times get out of hand with hash bucketing. With regular matrix PIR schemes, the recursive query only needs to retrieve a single element from each intermediate matrix row generated by the primary PIR. With hash bucketing, we need to retrieve more than one element per intermediate row. This means that for each element in the intermediate matrix, the mappers of the recursive PIR emit more than one doubly encrypted element. As we have seen our cryptosystem has a rather bad blowup factor, so producing an extra amount of doubly encrypted elements puts a large load on the MapReduce run. Of course, after reduction, the final output is as small as for square matrix PIR, but the increased amount of data to be processed and shuffled by the MapReduce system heavily affects performance. One can argue that
5.7. Subscription volume

Doing a recursive query is not necessary for hash bucketing, because the intermediate matrix is already only $\sigma \log k \times k$ large. Basically we can draw another generalized conclusion about the effectiveness of recursive PIR by saying that it is most effective as the number of rows in the intermediate matrix approaches $k$, i.e. the number of total batch queries. In the hash bucketing case, as $\sigma$ grows larger to ensure a smaller chance of not correctly retrieving all elements, the intermediate matrix becomes sparser in terms of elements of interest, and the recursive PIR query more effective.

Finally we have a look at output processing speed. This is of interest when a follower requests the output of a matching run. For one, the server needs to combine the relevant separate reducer output files (cf. Section 4.2.2 on page 44) and the follower needs to decrypt this downloaded file. Both of these properties were tested directly on the cloud by having one of the nodes run both the output combination and the decryption which would normally be done by the follower. Figure 5.12 shows the time trends as well as the actual output size.

![Output size](image1.png)  ![Time to combine and decrypt](image2.png)

(a) Size of output. (b) Time to combine and to decrypt.

Figure 5.12: Output size and time required to combine and decrypt it respectively for matching runs done with 1,000 tweeters with 100 tweets each and 1 follower with increasing number of subscriptions.

Calculating average throughput we get about 10MB/s for the output combination and 4.5MB/s decryption speed. This is run on a single “High-Memory Double Extra Large” instance which uses 13EC2 compute units [2]. Neither decryption nor output combination are explicitly parallelized so we assume that both run on a single EC2 compute unit, which amounts to a CPU capacity of a 1.0-1.2 GHz 2007 Opteron or Xeon processor [2].
5. Evaluation

5.8 Asymmetric database

So far we have tested scenarios where all tweeters have the same number of tweets. In a real system this is usually not the case. On Twitter, for example, around 5% of the user base make up for 75% of the activity [38]. For this reason we set up a database which follows a power law distribution, where 5% of the tweeters make up for 75% of the tweets and the tweet amount falls off sharply afterwards. We note that this is again a test where we use unrealistic parameters in that some tweeters will have thousands of tweets. In a deployed system, asymmetry will be mitigated to a degree by windowing, depending on how large we allow individual tweet files to become.

As usual, we set the number of followers to 100 with 1 subscription each. So we can compare these results to the symmetric database of Section 5.5 on page 56 we increase the number of tweeters the same way and choose our distributions such that the total number of tweets in the database, and hence its total size, is the same.

Figure 5.13 shows the matching times for the asymmetric database compared to the symmetric database where each tweeter has 100 tweets.

![Figure 5.13](image)

Figure 5.13: Log-log plot of the matching time for 100 followers with 1 subscription each in a database with increasing number of tweeters. Comparing an asymmetric power law distribution of the tweets and a symmetric distribution with 100 tweets per tweeter.
The first thing we like to note is that the asymmetry in file sizes and therefore load on the workers is mitigated by our combine method which tries to bundle the files together such that each worker gets an equally sized share. As we have seen in Section 5.5 on page 56, not using the combine method has disastrous results. The asymmetry seems to have no adverse effect on the combine overhead as it is almost equal to the overhead in the symmetric case. To get a better understanding of the slower time for the asymmetric case we again split the time up into primary PIR and recursive PIR in Figure 5.14.

Figure 5.14: Log-log plot of the matching time for asymmetric and symmetric databases, split up into the primary PIR run (PIR1) and the recursive PIR run (PIR2). The solid lines represent the primary PIR of each approach, the dashed lines represent the recursive PIR. 100 followers with 1 subscription each, 100 tweets in each symmetric tweet file and increasing the number of tweeters.

Primary PIR time is similar in both cases. Larger files obviously result in more blocks to handle. In the primary mapping phase this does not lead to an overhead because the total number of blocks is still roughly equal to a symmetric database of same size. During reduction, some reducers will get the “higher” blocks of the larger files (cf. Figure 5.15 on the next page).
Figure 5.15: Simplified view of how a reducer compacts a column with differently sized files. For the “higher” blocks of the larger files, the sum the reducer calculates simply has fewer summands. The final reducer output, which will be an element of the intermediate matrix, is then as large as the largest file of the column. Note that usually it is not a single reducer that works on all blocks of a column, but a reducer always gets all the blocks with the same index.

In the extreme case one reducer will get blocks of only the one largest file because all other files end before that.

This obviously does not affect correctness since it just sums up the blocks (cf. Section 4.2.2 on page 44). This means that the computational overhead up until this point is comparable to symmetric databases because we do not need to explicitly pad the smaller files to the largest size and the number of homomorphic operations is the same. What it does affect, however, is the size of a single element in the intermediate matrix. As the primary PIR reducers compact columns, the compacted intermediate element is as large as the largest element in that column. This explains the slower time of the primary PIR because the reducers need to write significantly more for each column than in the symmetric case (cf. Figure 5.16 on the next page).

It also directly affects the recursive PIR which now has to work on the much larger intermediate matrix. This is why it is much slower in the asymmetric case. This result reinforces our belief that windowing is crucial to the perfor-
5.9. Tweet committing

We present some results on how long it takes to integrate a certain number of tweets into the HDFS database. Remember that tweets are stored in a cache first and committed in batch at certain intervals. We test committing cached tweets of an increasing number of tweeters with 10 tweets each. We test committing times for different scenarios: (i) There are no tweet files in the database for the tweets cached; (ii) There are tweet files in the database and we append the cached tweets; (iii) There are full tweet files

Figure 5.16: Log-log plot of the number of bytes written by reducers in the primary PIR stage for asymmetric databases versus symmetric databases. 100 followers with 1 subscription each, 100 tweets in each symmetric tweet file and increasing the number of tweeters.

mance of our system, as we do not want to let certain tweet files get too large and blow up the intermediate matrix. As an aside, the observations previously made directly imply that the final output which a follower downloads is as large as the largest file in the database (plus encryption blowup). This is an invariant of any PIR scheme since otherwise the database can derive information about what element was retrieved based on output size.

5.9 Tweet committing

We present some results on how long it takes to integrate a certain number of tweets into the HDFS database. Remember that tweets are stored in a cache first and committed in batch at certain intervals. We test committing cached tweets of an increasing number of tweeters with 10 tweets each. We test committing times for different scenarios: (i) There are no tweet files in the database for the tweets cached; (ii) There are tweet files in the database and we append the cached tweets; (iii) There are full tweet files
5. Evaluation

in the database and we move and mark them as the old window file; (iv) We do not use batching, i.e. for each tweeter we commit each of his tweets separately. Figure 5.17 depicts plots of the time required to commit in these four scenarios.

![Graph showing time required to commit](image)

Figure 5.17: Log-log plot of the time required to commit a tweet cache to HDFS containing tweets of an increasing number of tweeters, each having 10 tweets in the cache. Comparison of committing to new files ('New'), appending existing files ('Append'), marking existing files as old ('Mark as old'), and not using batching ('No user batch').

Clearly, not using batching has very adverse effects. When we batch, we still need to open separate files for each tweeter, but we can write all of that tweeter’s tweets into that opened file. Without batching, if a tweeter posts multiple tweets, the server needs to open the tweet file each time and append the single tweet. Appending, as can be seen in scenario (ii), imposes an overhead on commit time due to how Hadoop handles this internally. We would like to note that appending is a relatively new feature in Hadoop and may still be optimized. Finally, our system of marking full files as old and thus keeping two window files does not seem to have a large overhead. The process of deleting an existing old window file and renaming the current tweet file is relatively fast. Overall, tweet committing is reasonably fast and scales linearly with increasing tweet cache size. It is clearly only a fraction of the server load compared to the matching process.
5.10 Evaluation conclusions

We have run various tests to evaluate the key aspects of our system. Most of these tests, as noted in the section on our testing philosophy, are not directly realistic tests of a deployed system. With this approach we have learned and shown a lot of interesting things about the system which might have gone unnoticed in a more realistic setting with humbler parameters. In the following we list some major conclusions drawn from the test results as well as from the actual test procedure which gave us an in-depth insight into working in the cloud and with Hadoop in particular. For a more general conclusion about the merits and disadvantages of our system, see the next chapter.

- **Hadoop configuration is of paramount importance.** This applies to our system more than it may in other MapReduce processes. The most important settings of a MapReduce job are the choice of number of mappers and reducers. Choosing too few fails to leverage concurrency and has bad consequences in case of failures. Choosing too many will increase the overhead unnecessarily. In most systems, the number of mappers is determined automatically by the framework, which splits up the input as it is split up in the distributed file system. The main reason we cannot make use of this (aside from the need to have full files for each mapper) is because of batch queries. Each mapper’s work gets multiplied by a factor of the total number of queries to be answered. So even in many of the cases where we only use 100 queries, each worker does 100 times more work than just on the single input it receives from the system. For this reason we have to explicitly configure our input format to take this into account. In our tests, we have found that instead of giving one HDFS data block worth of data to each mapper, as is default, we set our combine format to give each mapper roughly $2lq$ bytes of input, where $l$ is the size of a HDFS data block and $q$ the total number of queries.

- **Data blowup is significant.** Even with a moderate input for our matching, the amount of data that gets generated and needs to be handled during a matching run is significant. This is especially true in the primary PIR run before we start compacting columns. Obviously the input data gets multiplied by the number of total queries to be answered; each file block is applied to each corresponding query element of each follower and each of his subscriptions. Additionally the encryption blowup also has a rather large effect. For example, with a 15MB database, answering 100 follower subscriptions results in the mappers producing roughly 5.5GB of data. Obviously these amounts of data is exactly what MapReduce is intended for, but nevertheless working on, and especially transferring, such volumes of data increases our
5. Evaluation

matching time.

- **Efficient implementations are of value.** Related to the previous point, this is an observation that applies to all MapReduce processes working on large volumes of data. Since the map and reduce functions get applied to giga- if not terabytes of data a multitude of times, it is important to make them as efficient as possible. With respect to our system, we obviously gain a tremendous boost by using the efficient Trostle&Parrish cryptosystem. Interested readers may have a look at Appendix A.1 on page 77 for a small comparison to using the less efficient Paillier cryptosystem. Additionally we suspect that another boost in performance is achievable if a dedicated cryptographic library is used instead of standard Java BigInteger arithmetic.

- **Batch queries are a bottleneck.** Most MapReduce processes apply somewhat simple functions to a large amount of input data. This input data can be efficiently split up among workers. In our case, we do not only have our input data in the form of the database, we also have a second input in the form of the queries to be answered. The most important issue here is the fact that this second input cannot be split up among workers. Each mapper needs to apply all user queries to his share if the database. This causes two major points of overhead. One, the queries need to be distributed among the workers. This is somewhat mitigated by a nice service provided by Hadoop, called a Distributed Cache, such that on each matching run, only the queries that have changed or were added need to be distributed to workers which already have a previous snapshot of the queries. Nonetheless this is not optimal, specifically because this distributed cache is not intended for very large volumes of data. Secondly, as we have seen in Section 5.5 on page 56, loading all of these queries can amount to quite the additional overhead. We have already optimized memory usage by making sure a worker only keeps the vector elements of each query which apply to his database share. In the current version of the system, queries are serialized using default Java serialization and as such can only be loaded by reading the full query object. We envision that with some work the query loading can be optimized by using a custom format such that reading queries, extracting their meta information (e.g. the shape of the matrix or public keys) and individual vector elements is more modular and efficient. Additionally, reducers also load all the queries to retrieve the public keys that are needed for certain cryptosystems. Note that this is not necessary if we fix our cryptosystem to Trostle&Parrish as its homomorphic operations do not require a public key and are simply integer operations. We note, that for the evaluation we did not disable reducer query loading.
• **Offset batching is a boon.** Following the deliberations of the previous point, using our method of offsets instead of new query elements is a significant advantage. Just looking at the greatly decreased query size when using offsetting inspires us to argue that the small loss of privacy described in Section 3.4.1 on page 24 is acceptable.

• **Windowing is important.** Obviously, and confirmed by our tests, having very large database files is detrimental to our matching speed. Somewhat unique to PIR, asymmetry has the effect that all the intermediate and final output is blown up to the largest file in the system. Therefore it is doubly important to have some upper bound on tweet size such that overly active tweeters cannot affect the system too much. Additionally it makes it easier to configure our system based on these fixed upper bounds.
Chapter 6

Conclusion

We have created a working privacy-enhanced microblogging architecture that prevents the service provider from determining interests of followers in tweeters and hashtags. It is based on a recent advancement in Private Information Retrieval which uses both parallelization as well as a special, homomorphically efficient cryptosystem. For obvious reasons, our system cannot provide real time messages to the same degree as a non-private system like Twitter, but our approach of offline matching allows us to match tweets to followers in a timely and efficient manner. Additionally we have made sure that the system is user-friendly and can be used with a small Firefox extension and back-end doing the bulk of the work.

Our system is scalable due to its use of the MapReduce paradigm for the crucial matching process as well as a number of optimizations which aim to keep user communication low. We have shown that it certainly is more efficient than trivial PIR, but our method of processing a large number of queries in batch greatly increases the matching time. In the next section we discuss this issue and possible solutions.

6.1 Open problems

We divide the open problems of our system into problems that currently exist but may be fixed with a little more work and time, and into problems which are more general and may be seen as open research topics.

6.1.1 Batch processing

Our method of processing multiple queries/subscriptions in a single matching process is scalable in essence because it is integrated seamlessly into MapReduce. The biggest open issue is the fact that these queries need to be distributed among, and loaded by, all workers because they apply to the
whole database and thus to all shares that the workers get. This is an invariant and cannot be circumvented, but there are various optimizations that should decrease or shift the overhead. For one, as we have already mentioned in Section 5.10 on page 69, the serialization of these queries could be carefully crafted such that it is not necessary to load the whole query to retrieve meta-information like the shape of the matrix requested or the offsets of our batch method. In particular, reducers currently need to load the whole query to gain the public keys required to do aggregation for certain cryptosystems. Additionally, while any single query needs to be applied to all database files, not all of the query elements in its vector are required for that. For example, the database file with index 7 will be applied to all queries, but only to the 7th element of each query vector. Hence, it seems feasible to further optimize serialization such that individual query vector elements only can be retrieved and in particular distributed. We note that we already use a memory optimization that discards useless query elements while loading the whole query, but it is still necessary to load the whole thing.

Besides this specific issue of batching queries, there is also the computational overhead of working with many queries on the whole database. As we have seen, things like hash bucketing to reduce the number of queries applied to each database element is an option, but has its own problems. Furthermore this may be improved with the batch code technique of the paper that inspired this method in the first place [23]. These methods allow us to save on computation for multiple subscriptions of a single follower, but it is an open research problem of how to reduce computation for multiple requests from different users (cf. [32] and a solution using anonymization [24]). Such optimizations would not only reduce the immediate computation but may also decrease the overhead caused by the significant data blow up as mentioned in Section 5.10 on page 69.

Finally, depending on the resources of the provider, splitting up the matching process into multiple jobs working on different splits of the queries may also be an option. This mitigates the distribution overhead and may be easier than just introducing more nodes to the cluster running a single matching process.

6.1.2 Batch query design

Our method of using offsets to reuse a query vector for multiple subscriptions is very efficient in terms of communication overhead, but has the drawback of linking all batch queries such that they all rely on the security of the single query vector.

We have opted to go the way of reducing the direct overhead of additional queries, but of course it may be sufficient to use a more communication
efficient PIR scheme in the first place such that trivial batching is accept-
able. This was not an option for us, as we have noted with regard to the
efficiency of the Trostle&Parrish cryptosystem and its drawbacks in terms
of recursively applying it more than once.

6.1.3 Private Index Retrieval

In Section 3.5 on page 27 we have discussed various methods of actually re-
trieving the index of the element we are interested in if we have a descriptor
of it which is independent of the database content, e.g. a user name which
remains constant no matter what other users exist in the database. We feel
that this is an important topic which often gets disregarded in favor of the
more immediate optimization of the actual PIR process. Using perfect mini-
mal hash functions could be considered an easy fix, but we feel that this is a
rather trivial solution which still induces linear communication, albeit with
a much smaller constant than trivial PIR if the database elements are larger
than a few bits. In particular, an option to directly retrieve elements with a
non-numeric descriptor would be exceedingly interesting. It would allow off-
line matching without the limitations or special designs that we have used.
Oblivious polynomial evaluation, for example, would allow the follower to
deposit his encrypted powers and then receive the appropriate element no
matter how the database changes, as long as the hash function remains con-
nstant. But as we have seen OPE is not a good fit for retrieving very large
elements.

6.1.4 Hashtag subscription size

Our system allows for subscriptions to hashtags which are not currently in
the database. For this reason we require these queries to handle the full
hashtag space. Even with the limitations we impose on the format of hash-
tags to reduce the size of this space, hashtag subscriptions are significantly
larger than tweeter subscriptions. Online matching where the follower sends
a query matching the current database each time would be the easiest way,
but then we lose the advantages of periodic offline matching. Descriptor-
based PIR as described in the previous section would be a valid option, too,
as we could use some encrypted derivatives of the hashtag to match tweets
offline.

6.1.5 Collusion

Collusion is a problem if users have to ask each other what their user index
in the database is. Any tweeter may share the requests he has gotten for his
user index with the service provider. As we have discussed in Section 3.5.2
on page 32, there is already a sub-optimal system in place in the form of our
Oblivious Polynomial Evaluation-based user lookup. We feel that extending
and optimizing this would increase the use of our system. Additionally, users may of course use simple anonymization techniques like pseudonyms; e.g. not directly asking with their microblogging username. More advanced approaches like anonymity networks or our suggestion of a third-party user lookup service may also be applicable.

6.1.6 Content privacy

Our system protects information about user relationships, but does not prevent the provider nor anyone else from reading the actual tweet content. As we have hinted at several times during this work, the core of our system is extendable in that the matching process does not rely on a specific tweet format. This means that we could easily encrypt the content and use a technique like in [15] to allow for access control and sharing of keys among users.

6.1.7 Twitter conveniences

At the start of this thesis we have described the general features of Twitter, among which is addressing specific users and retweeting tweets. These features are not supported by our system for the obvious reason that this directly leaks information about the relationship, because content is not encrypted. If encrypted content is available as described in the previous section, retweeting is straightforward, but reveals the relationship to the followers of the retweeting tweeter. I.e. if tweeter A follows tweeter B and retweets one of his tweets (which is marked as a retweet), then A's followers will be aware of the fact that A follows B. Addressing users is more complicated, because it is not straightforward how a user would retrieve tweets addressed to him by tweeters he does not follow. Using a manner of mailbox files in the database from which a user retrieves tweets addressed to him switches the problem to obliviously writing to that mailbox which is generally even harder (cf. Oblivious RAM).
Appendix A

Appendix

A.1 Evaluation of Paillier

We briefly present the results of our tests using the Paillier cryptosystem [33] instead of Trostle&Parrish. Paillier has significantly less efficient homomorphic operations because homomorphic addition requires modular multiplication and scalar multiplication requires modular exponentiation. The following tests are very rudimentary and only measure time for the primary PIR run and not the recursive PIR because of time constraints. Furthermore only a single run was done per data point. We use a database of 1,000 tweeters with 100 tweets each and a single follower with varying number of subscriptions as seen in the table below.

<table>
<thead>
<tr>
<th>Subscriptions</th>
<th>Paillier</th>
<th>Trostle&amp;Parrish</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>152</td>
<td>132</td>
</tr>
<tr>
<td>10</td>
<td>1039</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>151</td>
</tr>
</tbody>
</table>

Table A.1: Runtime in seconds of the primary PIR when using Paillier versus Trostle&Parrish.

Clearly, Paillier is exceedingly inferior to Trostle&Parrish as just 10 subscriptions make it an order of magnitude slower than Trostle&Parrish on 100 subscriptions.

A.2 Misc. thoughts on Hadoop configuration

We present miscellaneous thoughts on configuration details of Hadoop for our system.

- **Mapper and reducer count.** Number of mappers can be manipulated
by specifying the maximum size that the combine format uses to combine small files. It should be scaled by the total number of queries because they multiply the work of each mapper.

Reducers should be set such that each one handles roughly 1–2GB of data. A good estimate on data output by mappers and handled by reducers can be gained with $3.4 \cdot l \cdot q$ where $l$ is the database size, $q$ the total number of queries and 3.4 an estimate on the blowup factor of the Trostle&Parrish encryption.

- **Intermediate and final output compression is useless.** Hadoop offers compression of intermediate and final output. This helps reduce I/O operation overhead at the cost of CPU overhead. It is only useful if a good compression factor is achievable. Because most of our output is encrypted and hence has high entropy, compression is almost worthless and should not be used. It is enabled by default on Amazon EMR, for example.

- **Adjust spill buffer memory.** Hadoop tries to keep mapper output in memory for as long as possible, and only pre-sorts and spills it to the hard disk when certain thresholds are reached. These thresholds tend to be too low (on EMR) and cause our system to be very slow due to multiple spills to the hard disk caused by the large amounts of data our mappers generate.

- **Combiners are only useful with few subscriptions.** Combiners are special workers besides mappers and reducers which normally work the same way as reducers but only on the data of a local mapper. This is used to pre-aggregate its output before it gets transferred to other nodes in the cluster. It can be very useful in reducing our data volume and hence increase the reducer phase efficiency where this data gets moved around. The problem is that a combiner is called each time a mapper spills to the hard drive. And each time the combiner needs to load the queries to be able to aggregate the mapper output. Hence if this loading takes long, it severely slows down the system because combiners constantly reload. It will also be more useful when query serialization is optimized such that reducers and combiners no longer need to load the whole query.
Bibliography


