# ZeroDB whitepaper[a]

M. Egorov[†] and M. Wilkison[‡]
*ZeroDB, Inc.*
(Dated: February 23, 2016)

ZeroDB is an end-to-end encrypted database that enables clients to operate on (search, sort, query, and share) encrypted data without exposing encryption keys or cleartext data to the database server. The familiar client-server architecture is unchanged, but query logic and encryption keys are pushed client-side. Since the server has no insight into the nature of the data, the risk of data being exposed via a server-side data breach is eliminated. Even if the server is successfully infiltrated, adversaries would not have access to the cleartext data and cannot derive anything useful out of disk or RAM snapshots.

ZeroDB provides end-to-end encryption while maintaining much of the functionality expected of a modern database, such as full-text search, sort, and range queries. Additionally, ZeroDB uses proxy re-encryption and/or delta key technology to enable secure, granular sharing of encrypted data without exposing keys to the server and without sharing the same encryption key between users of the database.

## I. INTRODUCTION

Given the cost, performance, and scalability benefits of cloud environments, outsourcing of on-premise infrastructure is accelerating. However, concerns over data security and the frequency of high-profile data breaches continue to hinder cloud adoption in highly-regulated and security-sensitive industries. While strong encryption can alleviate many of these concerns by guaranteeing data confidentiality, once data is encrypted it is no longer usable. Several strategies for computing on encrypted data have recently emerged that seek to provide the benefits of encryption while preserving the functionality of encrypted data.

These techniques fall into two broad categories: (1) encryption schemes that enable server-side computation directly over ciphertext and (2) trusted modules that are assumed to be inaccessible by adversaries [**?** ]. These two approaches

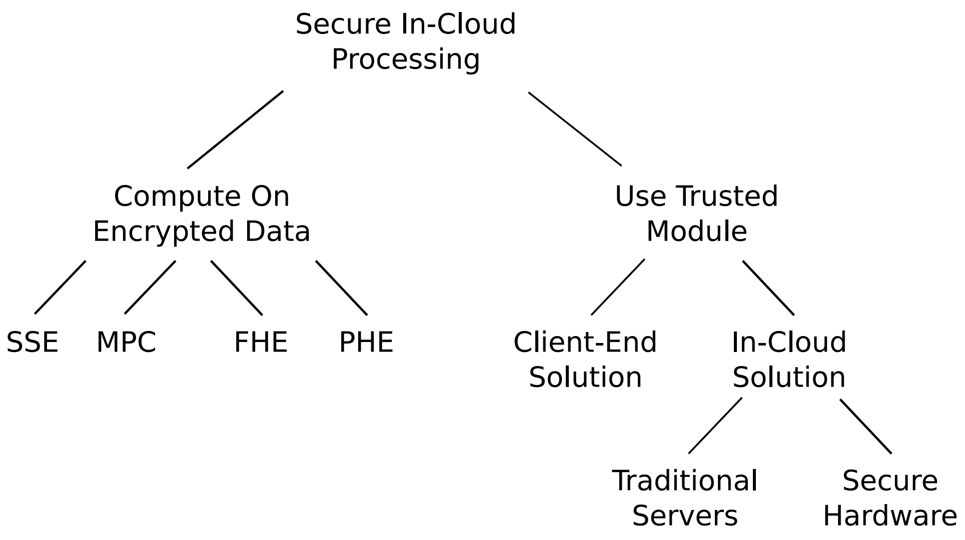
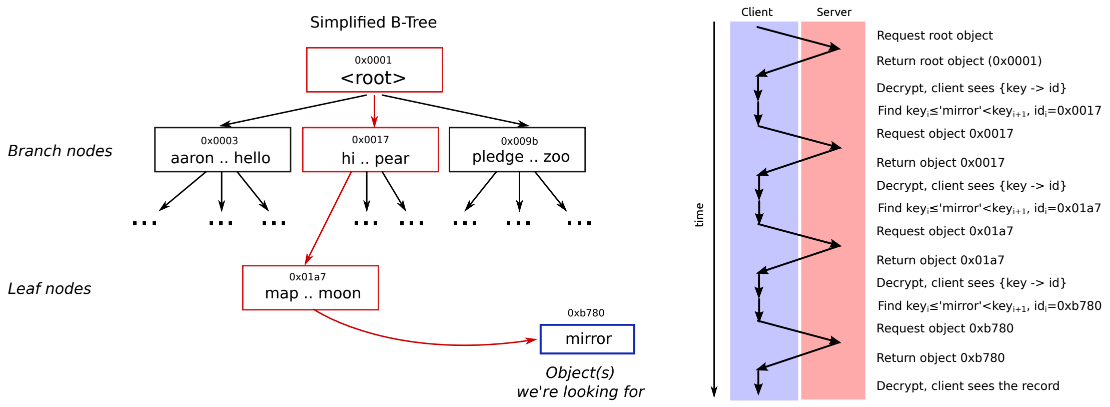


FIG. 1: Taxonomy of approaches to secure in-cloud processing. Computing directly on encrypted data can be done in a fully or partially homomorphic (FHE and PHE, respectively) manner. Searchable symmetric encryption (SSE) or secure multi-party computation schemes (MPC) may be used as well. Trusted Modules (TM) include client-end and in-cloud solutions.

[a] This whitepaper describes planned functionality but not necessarily the current state of ZeroDB
[†] michael@zerodb.io
[‡] maclane@zerodb.io


Simplified B-Tree
0x0001
<root>
Branch nodes
0x0003
aaron .. hello
0x0017
hi .. pear
0x009b
pledge .. zoo
Leaf nodes
0x01a7
map .. moon
0xb780
mirror
Object(s) we're looking for
Client
Server
time
Request root object
Return root object (0x0001)
Decrypt, client sees {key -> id}
Find key$_i$≤'mirror'<key$_{i+1}$, id$_i$=0x0017
Request object 0x0017
Return object 0x0017
Decrypt, client sees {key -> id}
Find key$_i$≤'mirror'<key$_{i+1}$, id$_i$=0x01a7
Request object 0x01a7
Return object 0x01a7
Decrypt, client sees {key -> id}
Find key$_i$≤'mirror'<key$_{i+1}$, id$_i$=0xb780
Request object 0xb780
Return object 0xb780
Decrypt, client sees the record

(a) Encrypted index traversal example (simple keyword search)

(b) Sequence of client requests for traversal of the example index

FIG. 2: Search protocol for equality query using an example of keyword search

can be further deconstructed into (1) searchable symmetric encryption (SSE), secure multi-party computation (MPC), and fully and partially homomorphic encryption (FHE and PHE, respectively) and (2) client-end solutions and in-cloud solutions, resulting in the taxonomy shown in Fig. **??**.

Notable works in this area include Gentry's seminal paper on homomorphic encrytion [**?** ], Popa's CryptDB [**?** ], which executes SQL queries over encrypted data using a collection of efficient SQL-aware encryption schemes, Dynamic Searchable Symmetric Encryption [**?** ], Efficient Secure Two-Party Computation Using Symmetric Cut-and-Choose [**?** ], which yields a 3x performance gain against existing garbled circuit schemes, and Cipherbase [**?** ], a comprehensive database system that provides strong end-to-end data confidentiality through encryption and secure hardware. Smart provides a more thorough exploration of the latest research and private sector efforts to achieve data encrypted in-use [**?** ].

ZeroDB provides confidentiality of data stored on the database server. It assumes a client(s) that initially owns the data and associated encryption keys. This client(s) wishes to outsource the encrypted data to an untrusted server, while maintaining usability of the data. It assumes passive (honest but curious) adversaries who do not return wrong results to the client. In the future, we may consider active adversaries and modify our protocol accordingly.

## II. QUERY PROTOCOL

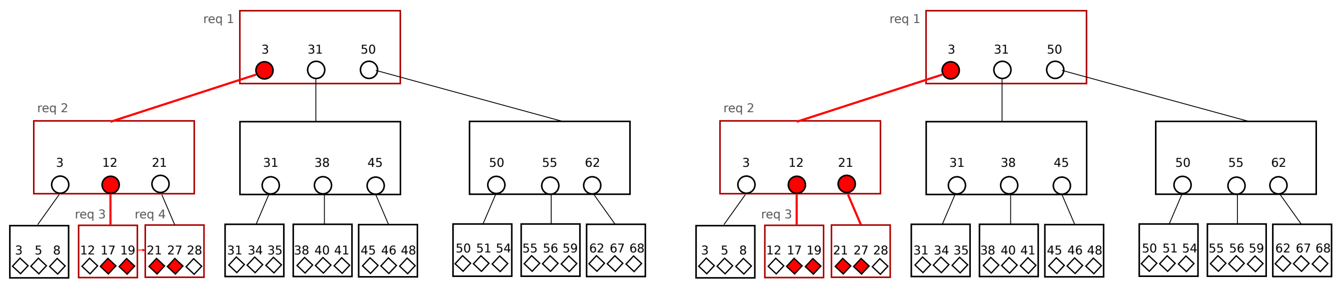
The basis of ZeroDB's end-to-end encrypted query protocol is as follows. The client interacts with the server during the execution of a query over a series of round trips. An encrypted index is stored on the server as a B-Tree, and the client traverses this index remotely to retrieve the necessary encrypted records. This remote traversal happens incrementally, with each round trip corresponding to a step down the B-Tree index. The index consists of buckets which are encrypted before being uploaded to the server and only decrypted client-side.

### A. Equality query (using an example of keyword search)

In ZeroDB, indexes are structured as encrypted B-Trees as in Fig. **??**. A B-Tree consists of encrypted buckets, each of which can be either a root, branch, or leaf node. The leaf nodes of a tree point to the actual objects being stored. Thus, searching the database is a simple tree traversal.

In order to make the database end-to-end encrypted yet still capable of performing queries, the client encrypts the buckets (at the time of creation or modification). The server, which stores the buckets, never knows the encryption key used. The objects referenced by the leaf nodes of the B-Tree indexes are also encrypted client-side. As a result, the server doesn't know how individual objects are organized within the B-Tree or whether they belong to an index at all. Since ZeroDB is encryption agnostic, probabilistic encryption can ensure that the server cannot even compare objects for equality.

When a client performs a query, it asks the server to return buckets of the tree as it traverses the index remotely and incrementally, as in Fig. **??**. The client fetches and decrypts buckets in order to figure out which buckets in the



(a) Query searching for objects with a property *weight* $\geq 16$, *limit* $= 4$ (takes 4 requests w/o cache)

(b) Query which fetches *all* objects with $16 \leq$ *weight* $\leq 27$ (takes 3 requests w/o cache)

FIG. 3: Range queries

next level of the tree to fetch next. Frequently accessed buckets can be cached client-side so that subsequent queries do not make unnecessary network calls.

The server can infer which level of the tree buckets belong to by observing search patterns, however it cannot figure out the ordering of elements since it can't see which link from the bucket was used.

The server is responsible for data replication, multi-version concurrency, object locking, user authentication, quotas, and optional immutability/write-once-read-many (WORM) guarantees. The client performs encryption/decryption and executes query logic. Thus, even if the server maliciously gives data belonging to user Alice to user Eve, Eve will not be able to decrypt the obtained data.

## B. Range queries

When data is organized in B-Trees, range queries are trivial. Take an example having records, *Record*, with an integer property, *weight*. Data pointers to *Record* objects are placed in a B-Tree sorted by *weight*.

Two different types of range queries could be performed. One is when we want a small subset of data in the beginning of the range (limit query). In this case, we find a pointer to the beginning of the range, as in Fig. **??**, then we incrementally fetch subsequent buckets if the range occupies more than one, then bulk-fetch the objects by their pointers.

The other case is when we want to get *all* objects in a range (select *). We download the subsection of the B-Tree matching the range query level-by-level in a logarithmic number of steps, as in Fig. **??**. To do that, we start with the root bucket. Then, we download all the child buckets which match the range at once, repeating until we get to the leaf nodes. After that we (optionally) bulk-fetch all the objects which match the range query at once. This is done in a way similar to prefetching all trees (Section **??**). Selecting all objects in a range reveals the approximate number of objects in this range to the server (the range and field names remain secret).

## C. Complex queries (multi-keyword search, multiple conditions)

So far we have described simple queries. But queries may be more complex and contain multiple conditions. Depending on the number of objects matching each condition, and our desired security properties, we can select between two different approaches. Making a query with an "or" condition requires simply zip-joining two sorted datasets, so we do not consider that to be problematic. Instead, let's consider a query where we select objects matching the condition $(v_1 = a)\,\&\,(v_2 = b)$ and ordered by $v_3$.

### *1. Prefetch approach*

In case the number of items with $v_1 = a$ is small, it could be efficint to download the entire subset of matching object IDs.

We can use the following indexes for this kind of query:

$$\begin{aligned}&\text{BTree}(v_1 \to \text{TreeSet}(ids)),\\&\text{BTree}(v_2 \to \text{TreeSet}(ids)),\\&\text{BTree}(id \to v_3).\end{aligned}$$

First, we estimate which condition has the smallest number of matching elements. We do so by fetching contours of the B-Trees (corresponding to smallest and largest elements of the range). Since we know the average size of a bucket, we can determine the height of the tree and approximate distance between the smallest and largest elements. This only requires a number of requests between client and server equal to the height of the tree, $H$.

We prefetch a TreeSet for the most lightweight condition (Sec. **??**). Then we bulk-search (Sec. **??**) these IDs in the larger TreeSet (and if we do not find an ID in the leaf node, we drop it).

After that, we need to sort the small subset of fetched IDs by $v_3$. We do a parallel traversal of the third B-Tree and find which value of $v_3$ corresponds to each ID.

This approach reveals the number of elements matching condition $v_1 = a$ to the server, although the condition itself remains unknown.

#### *2. Preorder approach*

The prefetch approach can be slow and reveal too much information to the server. So, whenever possible, values are pre-ordered in the index. It works in the following way.

The composite index to make a query selecting objects matching the condition $(v_1 = a) \,\&\, (v_2 = b)$ and ordered by $v_3$ is:

$$\text{BTree}((v_1, v_2) \to \text{BTree}(v_3 \to \text{TreeSet}(ids))).$$

For this query, we find a B-Tree (or multiple B-Trees) corresponding to the conditions and lazily traverse them in order of $v_3$. This makes limit orders much more performant and doesn't leak information about the dataset size to the server.

### D. Incremental full-text search

Full-text search with keywords, $t_1, t_2, \ldots, t_n$, requires ordering results by a net score or rank, which is determined according to a document's relevance to a given keyword. One rarely needs all the matching documents at once, and the result is almost always sorted by the score. This makes it possible to do the search in an incremental, pre-ordered manner, despite the score depending on the query.

Okapi BM25 [**?** ] is considered to be a state-of-art ranking function. However, when it comes to scalability, it doesn't work well because term frequency (TF) depends on both the length of a document and the mean length of documents in the database. This makes it impossible to pre-compute a score even for one keyword, $t$, without having to fetch the individual numbers of occurrences of $t$ in all the documents where it appears. Instead, we use Lucene's practical scoring function [**?** ].

In a simple form, the score is determined as follows. For a term, $t$, and document, $D$:

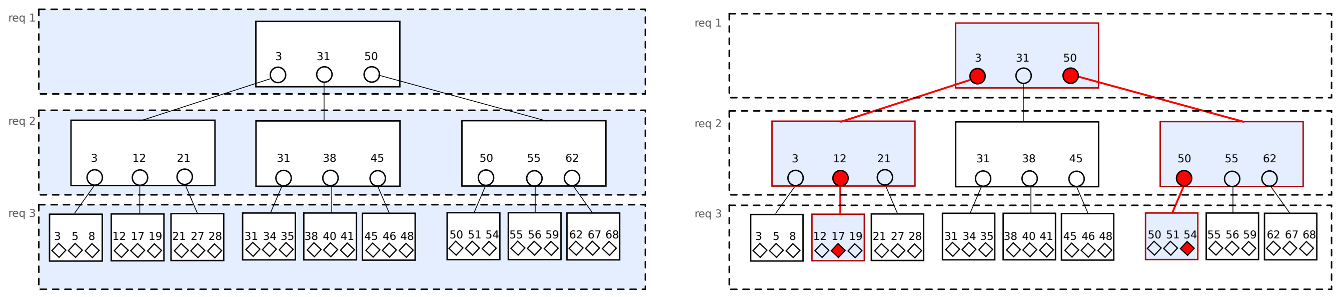
$$TF(t, D) = \sqrt{f(t, D)},$$

where $f(t, D)$ is the number of occurrences of $t$ in $D$. And the inverse document frequency (IDF) is:

$$IDF(t) = 1 + \ln\left(\frac{N_{docs}}{N_{docs}(t) + 1}\right),$$

where $N_{docs}$ is the total number of text documents indexed and $N_{docs}(t)$ is the number of documents which contain term $t$. The values of $N_{docs}$ and $N_{docs}(t)$ are stored in an encrypted form and updated when new documents are added. For easy conflict resolution, these values can be encrypted using additively homomorphic encryption (Pailier or exponential ElGamal). The score of a document for one term, $t$, will be:

$$s(t, D) = \frac{TF(t, D) \cdot IDF(t)^2}{\sqrt{|D|}},$$



(a) When a tree (or sub-tree) is small, it can be fully pre-fetched to the client

(b) When multiple branches of the tree needs reading/updating, tree traversal can be done in parallel

FIG. 4: ZeroDB-specific optimizations for working with B-Trees includ pre-fetching a tree or finding multiple values in a number of steps equal to the tree height

where $|D|$ is the total number of unique terms in document $D$. $TF(t,D)/\sqrt{|D|}$ is calculated at indexing time, while $IDF$ is calculated at query time since it doesn't depend on $D$. Thus, for one-keyword queries, documents are pre-ordered by relevancy.

Total document weight is defined as:

$$s(D) = \frac{\sum_{i=0}^{n} s(t_i, D)}{\sqrt{\sum IDF(t_i)^2}}.$$

This results in summing up the scores, $TF(t,D)/\sqrt{|D|}$, by which document references are pre-ordered (in descending order) with weights, $IDF^2/\sqrt{\sum IDF}$, calculated at query time.

In order to produce document IDs sorted by $s$ incrementally, we first fetch tuples $(s_j(t_i,D), docid_j)$ with the highest values of $s$ for all the query terms, $t_i$. Then we calculate possible ranges of document scores, given our knowledge of the highest known scores for terms $t_i$ and scores for particular *docids* which we've read. For example, a range of scores for a document, $D$, appearing in a sorted data structure would be:

$$\min(s(D)) = \sum_{\forall i \;\; s_j(t_i,D)\in\text{cache}} s_j(t_i, D), \qquad \max s(D) = \min(s(D)) + \sum_{\forall i \;\; s_j(t_i,D)\notin\text{cache}} \min_{\in\text{cache}} \left(s(t_i, *)\right).$$

If the known documents which haven't yet been returned don't overlap by their range of possible scores when sorted by $\min(s)$, it is fine to grab documents from the cache. If the ranges of possible scores overlap, we don't have enough information and we read more document scores of the keyword for which the difference between scores is currently the highest.

This algorithm doesn't reveal any dataset sizes when we're interested in only a fraction of documents, which is beneficial for both security and scalability. It is efficient when document weights for different keywords are significantly different from each other.

## E. Related objects

ZeroDB is based on ZODB, which uses object references instead of joins [? ] (similar to MongoDB's database references [? ]). In order to keep everything consistent, ZODB fires events at commit time [? ] to keep back-references and indexes up to date. In ZeroDB (as in ZODB), we can create indexes on attributes of related objects, much as we create normal indexes.

## F. Optimizations specific for remote client

In most cases, all the query logic in ZeroDB happens client-side and the client and storage are separated by a network channel with high latency. We use two primitives specific for this architecture.

#### 1. *Bulk-fetching small (sub)trees*

When a tree (or sub-tree) is small enough, it can be more performant to fetch the entire tree to the client. We can do that in a logarithmic number of steps, rather than bucket-by-bucket, as we would do with an index stored on a local hard drive.

To do so, we fetch the root first, as in Fig. **??**. The client decrypts the root bucket and fetches all of its children in a second query. It unpacks all the child nodes and learns the IDs of their children. This process repeats until the entire B-Tree is fetched. Thus, the number of queries is equal to the height of the B-Tree, $H$, proportional to the logarithm of its size, $\log S_{\text{ix}}$.

When we fetch a tree, we implicitly show the number of objects it contains, $S_{\text{ix}}$, to the server. Based on the size of the read data, the observer would be able to infer a number close to $S_{\text{ix}}$ and associate it with the bucket IDs just read. Therefore, this technique should be used rarely, and in combination with reading other data at the same time or with oblivious RAM [**? ? ? ? ?** ]. The latter would prevent the server from learning bucket IDs.

#### 2. *Parallel tree traversal*

Often, queries involve several values of the same field (or different fields). For example, indexing a new document containing 100 words can be an expensive operation since it would involve $100\,H$ requests to the server.

In order to fix that, we traverse the B-Tree in parallel, as in Fig. **??**. We first fetch the root of the tree. Then, we fetch only those child-buckets which are relevant to the values in our query, repeating as necessary. This way, we do tree traversal for all the necessary values in $H$ steps.

When we do parallel traversal, the server would be able to see access patterns but it would not be able to distinguish access patterns belonging to each of the values individually.

## III. SHARING DATA

It is often advantageous to share a dataset (for data analysis by third parties, to give temporary accesss to employees or regulators, etc.).

When there are multiple users of the same encrypted dataset, there are serious shortcomings with sharing the encryption key with all parties. Since the architecture of ZeroDB allows using any encryption algorithm, we can use proxy re-encryption [**? ?** ] or delta-keys [**? ?** ] to share data with other users (third parties) of the same dataset. These algorithms allow the third party to hold an encryption key other than that used to encrypt the data, yet still query the data, until the data owner revokes their access. By combining these sharing algorithms with ZeroDB's query functionality, we can granularly share subsets of data.

### A. Proxy re-encryption

Sharing an entire dataset is done as follows. Suppose user $A$ owns the dataset and wants to share it with user $B$. In the case of proxy re-encryption algorithms [**? ?** ], user $A$ has a key pair, $priv_a/pub_a$, and user $B$ has a key pair, $priv_b/pub_b$. All objects and index buckets are encrypted with random, distinct content-encryption keys, $cek_i$, using a block cipher algorithm. The $cek$ is encrypted with a key encrypting key, $priv_a$. So, we store $e_a(cek_i) = \text{encrypt}(priv_a, cek_i)$ and $c_i(obj) = \text{encrypt}(cek_i, data)$.

When user $A$ wants to give user $B$ access to the entire dataset and indexes, he constructs a re-encryption key, $r_{ab}$, such that the server can transform the content-encryption key as $e_b(cek_i) = \text{transform}(r_{ab}, e_a(cek_i))$. The re-encryption key, $r_{ab}$, is given to the server, along with an optional revocation time. While the server has this re-encryption key, it transforms CEKs of encrypted objects, $e_a(cek_i) \to e_b(cek_i)$, as client $B$ requests them. The server can revoke the key by removing $r_{ab}$. Thus, while $r_{ab}$ is valid and present on the server, client $B$ is able to do tree traversals in the same manner as client $A$ (Sec. **??**).

### B. Granular access permissions

Let's first consider granular sharing of data in a scheme where $A$ shows encryption keys to $B$ and shares $M$ objects selected by a range query. It is possible to share data granularly in such a way that $A$ has to only send $B$ a number

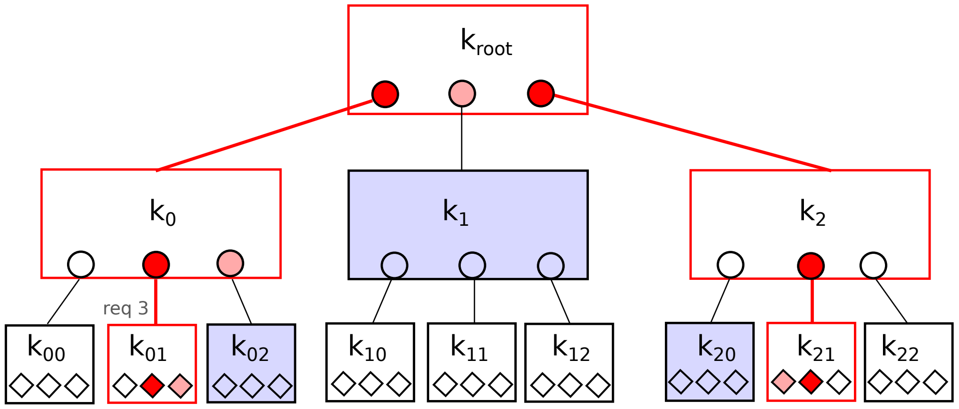


FIG. 5: B-Tree structure which supports selecting keys for any range of data. The hierarchy of keys is the same as its corresponding hierarchy of buckets. Giving keys from inside the contour would give access to objects only within the range.

of keys proportional to $\log M$ rather than $M$.

Keys, $k^a$, must form a tree hierarchy, as in Fig. **??**. Each node of a B-Tree has a key associated with it. Keys for child nodes $k_{*i}$ are derived from a key for the parent node $k_*$. The keys can be derived as $k_{*i} = \text{SHA256}(k_* + i)$. Client $A$ traverses the tree along the beginning and end of the range. It shares keys for those buckets with $B$, giving $B$ the ability to do tree traversal within this range (the buckets with blue background in Fig. **??**). However, there may be buckets which are partly within the range. Client $A$ passes key-reference pairs for buckets which belong to the contour of the range (depicted as red diamonds). There should be no more than $2\,s_b$ such pairs, where $s_b$ is the maximum capacity of a bucket. The set of keys giving access to the in-range fraction of the index and the in-range contents of the buckets from the contour of the range are given to $B$, encrypted with his public key.

The described sharing mechanism suffers from the fact that $B$ is given all the information to decrypt objects within the range at once. In order to make this access revokable, we need to use proxy re-encryption. In this scenario, client $A$ gives hierarchical rights to re-encrypt the data to the server rather than to client $B$ directly. That is, the server should only be able to re-encrypt objects within the range, given a set of re-encryption keys, $r_{ab}$. It is possible to do this using a conditional proxy re-encryption scheme (C-PRE) [**? ? ? ?** ].

Usually, such proxy re-encryption schemes support giving a product of multiple equality conditions. The conditions which we apply would be object IDs in the bucket hierarchy, starting from the top. For example, C-PRE encryption of a bucket in the second layer of the tree, $b_1$, would give the server the ability to derive re-encryption keys for all of its children, $b_{1*}$. This gives the server the ability to re-encrypt all buckets within the range specified by $A$, while only requiring $A$ to calculate a number of keys proportional to the log of the number of records shared.

## IV. SECURITY ANALYSIS

ZeroDB is agnostic to the encryption algorithm used. By default, we use authenticated strong encryption (AES-256 in GCM mode). Nothing is decrypted on the server.

Using deterministic or order-preserving encryption visible to the server can be detrimental to the security of the database in certain cases [**?** ]. However, even traversal of B-Trees allows an attacker to collect access statistics and infer the data in the database [**?** ]. We provide a security analysis and an example of the number of queries necessary for such an attack.

### A. Threat model

We adopt a threat model similar to the one used for the security analysis of CipherBase [**?** ]. We introduce a *resident adversary*, who has unlimited observational power and can view the contents of memory and disk at every moment of time. An *instant adversary* can observe only a snapshot of memory and disk at one particular moment in time. We assume that both resident and instant adversaries are passive (honest-but-curious) and do not return false results to the client. In the future, we may consider active adversaries and modify our protocol accordingly.

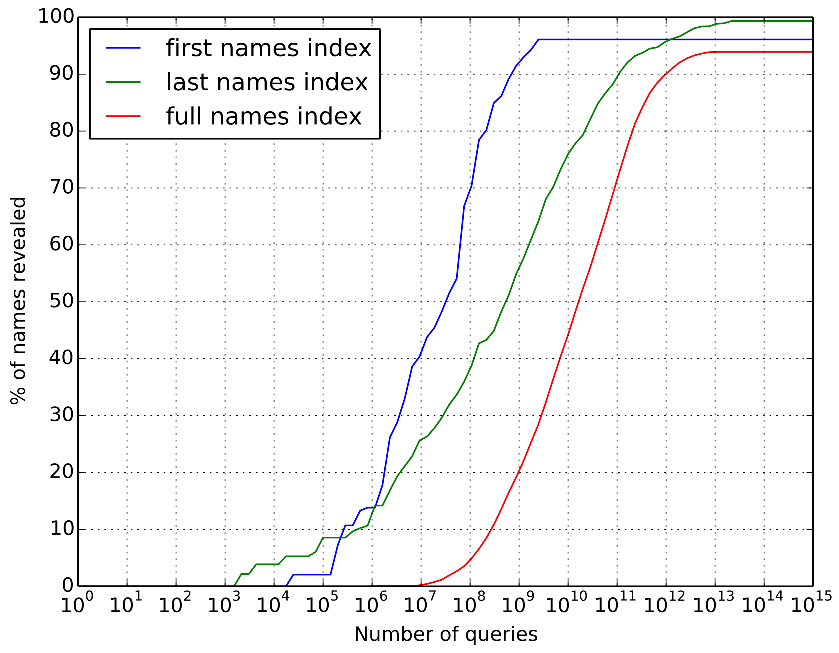


FIG. 6: Fraction of queries where last name can be inferred by a passive observer depending on the number of queries made to the database. The fraction of names able to be inferred depends on whether we index first and last names separately or together. In CryptDB, this information is available to an attacker immediately [**?** ].

## B. Data confidentiality

In ZeroDB, all contents of the database and its indexes are stored as encrypted objects. Therefore, an instant adversary doesn't learn anything except for database size (including the number of fields). So, ZeroDB provides *semantic security* against an instant adversary.

A resident adversary, on the other hand, is able to observe access patterns and collect statistics of those over time. So, we provide *operational security* against a resident adversary.

First, we consider search queries which only require tree traversal. For example, limit orders for queries which do not involve pulling a large subset of the index to the client. For simplicity, let's assume that all we have in a database index is a B-Tree of customer surnames. A resident adversary is able to acquire access statistics of the index leaf nodes when searches of, say, 10 people with a given last name are performed. However, the amount of statistics should be large enough to distinguish different records in the database. If we assume the probability of a given surname appearing in a given query is the same as in the U.S. Census 2000 data [**?** ] and that keywords are distributed across queries according to a Poisson distribution, an adversary needs to observe two keywords at least $N_1$ and $N_2$ times, such that $|N_1 - N_2| > \sqrt{N_1 + N_2}$. The fraction of last names which can be revealed by a resident adversary from a query, given some number of queries, is shown on Fig. **??**. For example, the surnames in 10% of new queries can be inferred by a resident adversary after observing a million queries (although surnames not touched by queries are not revealed). However, if we index full names as one field (first names are taken from UK Office of national statistics [**?** ]), we would need to do 300 millions queries to infer 10% of full names in new queries. This assumes that an attacker is aware of the statistics of data in the database, and nothing other than names are queried. So, these estimates provide a lower bound for the number of queries required to infer useful information.

Unlike CipherBase [**?** ], our approach doesn't leak information about the order of the elements. By observing access patterns, the server can only figure out which buckets are leaf nodes and which are branch nodes, but not whether the child node is located to the left or right of the parent node.

In cases where we need to prefetch a large chunk of a B-Tree (Sec. **??**), we immediately reveal information about the size of the sub-dataset to the server. Of course, this is undesirable. To avoid this, we can download adjacent records in the same query, so that the downloaded size is always approximately constant and equal to that of the largest sub-dataset. This results in similar security guarantees to those for tree traversal. In order for this to work we need to unfold nested trees into one tree (Sec. ??). The way we perform full-text searches (Sec. ??) doesn't reveal the numbers of documents per keyword since it's done in incremental manner.

We can significantly improve security guarantees by using ORAM techniques, which allow us to hide access patterns by writing back data after reading it. Several algorithms are available, though the most promising ones are Ring ORAM [? ] with XOR technique [? ] and oblivious data structures [? ]. Still, these require writing back to the database about 5-20 times the amount of data read (although these numbers improve every year). It is also possible that hiding access patterns in a multi-cloud or fully decentralized arrangement has quite attractive performance properties [? ], which is especially interesting in light of decentralized storage layers such as Storj.io [? ] and IPFS [? ].

## V. PERFORMANCE

### A. Optimizing bucket size

ZeroDB requires several interactions between client and server in order to return query results. Thus, both connection bandwidth, $b$, and time of round-trip between client and server, $\tau$, determine the performance of queries. Moreover, there is an optimal bucket size, $s_b$, for any given parameters.

Suppose we have a B-Tree index of size $s_i$ bytes. The size of one record in the bucket is $s_r$ bytes. Apart from encryption we can apply compression. The compression rate is defined as the ratio of initial bucket size to final (compressed + encrypted) bucket size is $c$. The number of requests necessary to perform a query with an empty cache is equal to the height of B-Tree, $h$, though we also consider the case where we cache up to $s_c$ bytes.

A good approximation for B-Tree height is [? ]:

$$h = \log_m n,$$

where $m = s_b/s_r$ is the number of records in one bucket and $n = s_i/s_r$ is the total number of records in the index, assuming the best and the worst case heights are close to each other. Thus:

$$h \approx \frac{\ln s_i - \ln s_r}{\ln s_b - \ln s_r}.$$

When caching, we cache predominantly the top of the B-Tree with an average cached height, $h_c$:

$$h_c \approx \frac{\ln s_c - \ln s_r}{\ln s_b - \ln s_r}.$$

The average number of interactions between client and server per query will be $k = h - h_c$. Thus, the time which this interaction takes is:

$$\Delta t = k\tau + k\frac{s_b}{cb} = (h - h_c)\left(\tau + \frac{s_b}{cb}\right) = \frac{\ln s_i - \ln s_c}{\ln s_b - \ln s_r}\left(\tau + \frac{s_b}{cb}\right).$$

We make queries as fast as possible (i.e. minimize time $\Delta t$ necessary for a query to complete) by seeking for the optimal bucket size, $s_b$. The largest possible bucket size is equal to the size of the index and is obviously not efficient for sufficiently large indexes (it takes too long to download the index). On the other hand, when bucket size is too small, the number of requests is too large and query performance suffers due to latency between client and server. An optimal compromise can be determined by equalizing the first derivative to zero:

$$\frac{\partial \Delta t}{\partial s_b} = 0,$$

which yields the optimal bucket size:

$$s_b = \frac{cb\tau}{W\left(\frac{cb\tau}{es_r}\right)},$$

where $W$ is the real part of Lambert W function [? ] and $e = 2.718281828\ldots$.

In ZeroDB, the typical size of a record in a branch bucket is $s_r = 29$ bytes, with a zlib compression rate of $c = 3$. For a typical network bandwidth of $b = 10^6$ (1 MB/s) and roundtrip time of $\tau = 0.05$ s, the optimal bucket size is

$s_b = 10$ kB when compression is not used, or 8 kB when compression is used. For a typical cache size of 5 MB and index size of 50 GB, an average query takes $t = .07$ s and requires network traffic of 11 kB of compressed data (without caching that would take 0.17 s and network traffic of 24 kB of compressed data), reducing the average number of roundtrips to 1.3. We can finetune the optimal bucket size on the fly as the connection speed improves over time.

When we need to traverse several indexes at once, we bunch requests for several indexes in one query. For example, the first request for indexes $A$ and $B$ would be "read the roots of indexes $A$ and $B$ (by object IDs of the roots)". In the past, that would make heavier queries impractical, but the growth in internet bandwidth according to Nielsen's law [**?** ] alleviates this.

### B. Unfolding trees with subtrees into one tree

By default, we often use nested B-Trees of TreeSets. While that yields performance benefits in traditional databases, it is not ideal when working with large latencies. And, more importantly, it has security drawbacks, since we cannot easily read adjacent records to hide the size of the dataset from the server.

Consider a B-Tree linking word IDs, *wid*, with TreeSets of object IDs, *oid*. We can instead use a treeset of tuples (wid, oid) which will be unique. The tree will have approximately the same height for all *wid*s. Also, it is easy to pre-set the approximate size of the dataset to download if we don't want to show how many *oid*s we have per *wid*.

## VI. CRYPTOGRAPHIC PRIMITIVES USED

Even though the server never decrypts the data, we use encryption in transit (using SSL) in order to prevent third parties from analyzing access patterns. The client authenticates with an X.509 certificate or a self-signed certificate where the private key is derived from a passphrase. When a passphrase is used, the scrypt algorithm is used to derive a private key from the passphrase. Information about the salt is stored server-side and is given before establishing an authenticated encrypted connection.

For symmetric encryption, we use AES256 in GCM mode. The symmetric key is derived from either the passphrase, the private key in an X.509 certificate, or provided by a key management service (KMS).

For proxy re-encryption, we use the AFGH algorithm [**?** ] to encrypt a random content key for an AES256 block cipher in GCM mode.

## VII. ACKNOWLEDGEMENTS

We thank Prof. David Evans from the University of Virginia and Prof. Nigel Smart from the University of Bristol for reviewing this whitepaper and giving advice for further improvements in the security and performance of ZeroDB. We also thank Prof. Nickolai Zeldovich, Christopher Fletcher, Albert Kwon and Ling Ren from MIT for their advice at earlier stages of ZeroDB's development. Additionally, we thank Dr. Raluca Ada Popa from UC Berkeley for fruitful discussions about ways to offload complicated query processing to the server in a secure manner.