Big Data Analytics over Encrypted Datasets with Seabed

Antonis Papadimitriou[1†], Ranjita Bhagwan[*], Nishanth Chandran[*], Ramachandran Ramjee[*], Andreas Haeberlen[†], Harmeet Singh[*], Abhishek Modi[*], Saikrishna Badrinarayanan[1‡]

[†]University of Pennsylvania, [*]Microsoft Research India, [‡]UCLA

Abstract

Today, enterprises collect large amounts of data and leverage the cloud to perform analytics over this data. Since the data is often sensitive, enterprises would prefer to keep it confidential and to hide it even from the cloud operator. Systems such as CryptDB and Monomi can accomplish this by operating mostly on encrypted data; however, these systems rely on expensive cryptographic techniques that limit performance in true "big data" scenarios that involve terabytes of data or more.

This paper presents Seabed, a system that enables efficient analytics over large encrypted datasets. In contrast to previous systems, which rely on *asymmetric* encryption schemes, Seabed uses a novel, additively *symmetric* homomorphic encryption scheme (ASHE) to perform large-scale aggregations efficiently. Additionally, Seabed introduces a novel randomized encryption scheme called *Splayed ASHE*, or SPLASHE, that can, in certain cases, prevent frequency attacks based on auxiliary data.

1 Introduction

Consider a retail business that has customer and sales records from various store locations across the world. The business may be interested in analyzing these records – perhaps to better understand how revenue is growing in various geographic locations, or which demographic segments of the population its customers are coming from. To answer these questions, the business might rely on a Business Intelligence (BI) system, such as PowerBI [4], Tableau [6], or Watson Analytics [7]. These systems can scale to large data sets, and their turnaround times are low enough to answer interactive queries from customers. Internally, they rely on the cloud to provide the necessary resources at relatively low cost.

However, storing sensitive business data on the cloud can raise privacy concerns, which is why many enterprises are reluctant to use cloud-based analytics solutions. These concerns could be mitigated by keeping the data in the cloud encrypted, so that a data leak (e.g., due to a hacker attack or a rogue administrator) would cause little or no damage. Systems like CryptDB [39] and Monomi [44] can accomplish this by using a mix of different encryption schemes, including deterministic encryption schemes [13] and partially homomorphic cryptosystems; this allows certain computations to be performed directly on encrypted data. However, this approach has two important drawbacks. First, these cryptosystems have a high computational cost. This cost is low enough to allow interactive queries on medium-size data sets with perhaps tens of gigabytes, but many businesses today collect terabytes of data [15, 32, 33, 43]. Our experimental results show that, at this scale, even on a cluster with 100 cores, it would take hundreds of seconds to process relatively simple queries, which is too slow for interactive use. Second, deterministic encryption is vulnerable to frequency attacks [36], which can cause some data leakage despite the use of encryption.

This paper makes two contributions towards addressing these concerns. First, we observe that existing solutions typically use *asymmetric* homomorphic encryption schemes, such as Paillier [38]. This is useful in scenarios where the data is produced and analyzed by different parties: Alice can encrypt the data with the public key and upload it to the cloud, and Bob can then submit queries and decrypt the results with the private key. However, in the case of business data, the data producer and the analyst typically have a trust relationship – for instance, they may be employees of the same business. In this scenario, it is sufficient to use *symmetric* encryption, which is much faster. To exploit this, we construct a new *additively symmetric homomorphic encryption* scheme (or, briefly, ASHE), which is up to three orders of magnitude more efficient than Paillier.

Our second contribution is a defense against frequency attacks based on auxiliary information – a type of attack that has recently been demonstrated in the context of deterministic encryption [36]. For instance, suppose the data contains a column, such as gender, that can take only a few discrete values and that has been encrypted deterministically. If the attacker knows which gender occurs more frequently in the data, she can trivially decode this column based on which ciphertext is the most com-

[1]Part of this work was done while Papadimitriou and Badrinarayanan were doing internships at Microsoft Research India.

mon. We introduce an encryption scheme called *Splayed ASHE (SPLASHE)*, that protects against such attacks by splaying sensitive columns to multiple columns, where each new column corresponds to data for each unique element in the original column. For columns with larger cardinality, SPLASHE uses a combination of splaying and deterministic encryption padded with spurious entries to defeat frequency attacks while still limiting the storage and computational overhead.

We also present a complete system called *Seabed* that uses ASHE and SPLASHE to provide efficient analytics over large encrypted datasets. Following the design pattern in earlier systems, Seabed consists of a client-side planner and a proxy. The planner is applied once to each new data set; it transforms the plain-text schema into an encrypted schema, and it chooses suitable encryption schemes for each column, based on the kinds of queries that the user wants to perform. The proxy transparently rewrites queries for the encrypted schema, it decrypts results that arrive from the cloud, and it performs any computations that cannot be performed directly on the cloud. Seabed contains a number of optimizations that keep the storage, bandwidth, and computation costs of ASHE low, and that make it amenable to the hardware acceleration that is available on modern CPUs.

We have built a Seabed prototype based on Apache Spark [2]. We report results from an experimental evaluation that includes running both AmpLab's Big Data Benchmark [1] and a real, advertising-based analytics application on the Azure cloud. Our results show that, compared to no encryption, Seabed increases the query latency by only 8% to 45%; in contrast, state-of-the-art solutions that are based on Paillier (such as Monomi [44]) would cause an increase by one to two orders of magnitude in query latency.

To summarize, we make the following four contributions in this paper:

- ASHE, an additive symmetric homomorphic encryption scheme that is three orders of magnitude faster than Paillier (Section 3.1);
- SPLASHE, an encryption scheme that protects against frequency-based attacks for fields that require deterministic encryption (Sections 3.3+3.4);
- Seabed, a system that supports efficient analytics over large-scale encrypted data sets (Section 4); and
- a prototype implementation and experimental evaluation of Seabed (Section 6).

2 Overview

Figure 1 shows the scenario we are considering in this paper. A *data collector* gathers a large amount of data, encrypts it, and uploads it to an untrusted cloud platform.

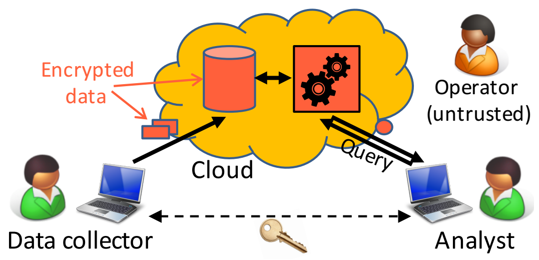


Figure 1: Motivating scenario.

An *analyst* can issue queries to a query processor on the cloud. The responses will be encrypted, but the analyst can decrypt them with a secret key she shares with the data collector.

The workload we wish to support consists of OLAP-style queries on big data sets. As our analysis in Section 5 will show, these queries mostly rely on just a few simple operations (sum, sum-of-squares, etc.), so we focus on these in our server-side design. Our goal is to answer typical BI queries on large data sets within a few seconds – that is, quickly enough for interactive analysis.

2.1 Background

One common approach to solving the above problem is to use *homomorphic encryption*. For instance, there are cryptosystems with an additive homomorphism, such as Paillier [38], which means that it is possible to "add" two ciphertexts $C(x)$ and $C(y)$ to obtain a ciphertext $C(x + y)$ that decrypts to the sum of the two encrypted values. This feature allows the cloud to perform aggregations directly on the encrypted data. There are other systems with different homomorphisms, and even *fully homomorphic* systems [26] that can be used to compute arbitrary functions on encrypted data (Section 7).

Homomorphic encryption schemes are typically randomized, that is, there are many different possible ciphertexts for each value. These schemes enjoy standard semantic (or CPA) security, which informally means that no adversary can learn any information about the plaintext, even given the ciphertext.

However, there are situations where it is useful to let the cloud see some property of the encrypted values (property-preserving encryption). For instance, to compute a join, the cloud *needs* to be able to match up encrypted values, which randomization would prevent. In this case, one can use *deterministic encryption* [13], where each value v is mapped to exactly one ciphertext $C(v)$. However, such schemes are susceptible to *frequency attacks* [36]: if a column can only take a small number of values (say, country), and the cloud knows that some value (say, Canada) will be the most common in the data, it can look for the most common ciphertext and infer that this ciphertext must decrypt to that value. Another example of an operation achievable by a property-preserving encryption scheme is selecting rows based on a range of values (say, timestamps) in an encrypted column. Here, one can use an *order-preserving encryption (OPE)* [17], which can be used to decide whether $x < y$, given only $C(x)$ and $C(y)$. Obviously, if the cloud can perform the comparison, then so can the adversary, so in these schemes, there is a tradeoff between confidentiality, performance, and functionality.

2.2 Threat Model

In this paper, we resolve the above tradeoff in favor of confidentiality and performance. We assume an adversary who is honest but curious (HbC), that is, the adversary will try to learn facts about the data but will not actively corrupt data or otherwise interfere with the system. We do, however, assume that the adversary will attempt to perform frequency attacks as discussed above; this is motivated by recent work [36], and it is the reason we developed SPLASHE.

We are aware that there are much stronger threat models that would prevent the adversary from learning anything at all about the data. However, current solutions for these models, such as using oblivious RAM [37, 29] and fully homomorphic encryption, tend to have an enormous runtime cost (fully homomorphic encryption [26] causes a slowdown by nine orders of magnitude [27]). Our goal is to provide a practical alternative to today's plaintext-based systems (which offer very little security), and this requires keeping the runtime overhead low.

2.3 Alternative approaches

As discussed in Section 2.1, one possible approach to this problem is to use homomorphic encryption. This approach is taken by systems like CryptDB [39] and Monomi [44], which use Paillier as an additive homomorphic scheme. While Paillier is *much* faster than fully homomorphic encryption, it is still expensive. For example, a single addition in Paillier on modern hardware takes about 4 μs (Section 4), so the latency for operations on billions of rows can easily reach several minutes.

An alternative approach is to rely on trusted hardware, such as Intel's SGX [35] or ARM's TrustZone [10]. This approach has a much lower computational overhead, but it introduces new trust assumptions that may not be suitable for all scenarios [22, 23]. It would be good to have options available that offer a low overhead without relying on trusted hardware.

2.4 Our approach

In Seabed, we solve this problem by replacing Paillier with a specially designed *additively symmetric homomorphic encryption* (ASHE) scheme. Since symmetric encryption schemes tend to be much more efficient than asymmetric schemes, this yields a big performance boost (Section 4). Symmetric encryption imposes a restriction that the encrypted data can only be uploaded by someone who has the secret key but this is not a constraint for the typical BI scenario. Thus, the additional protections of asymmetric cryptography are actually superfluous, and the performance gain is essentially "free".

Additionally, in order to protect against frequency attacks that occur when using deterministic or order preserving encryption, we construct a randomized encryption scheme − *SPLayed ASHE*, or SPLASHE that can still enable us to perform many queries on encrypted data that in prior work required deterministic encryption, but without leaking any information on frequency counts. Finally, for those queries that SPLASHE cannot support (e.g., joins), we support deterministic and OPE schemes that leak (a small amount of) information about the underlying plaintext values; we take this decision with the performance of the system in mind.

3 Seabed Encryption Schemes

In this section, we describe the ASHE and SPLASHE schemes in more detail. ASHE and the basic variant of SPLASHE satisfy the standard notion of semantic security (IND-CPA, that leaks no information about plaintext values) while the enhanced variant of SPLASHE provably leaks no more information than the number of dimension values that occur frequently and infrequently in the database. A formal security proof is available in Appendix A.

3.1 ASHE

ASHE assumes that plaintexts are from the additive group $\mathbb{Z}_n := \{0, 1, \ldots, n-1\}$. It also assumes that the entities encrypting and decrypting a ciphertext (the sender and the recipient, respectively) share a secret key k, as well as a pseudo-random function (PRF) $F_k : I \to \mathbb{Z}_n$ that takes an identifier from a set I and returns a random number from $\mathbb{Z}_n$.

One possible choice for the PRF is $F_k := H(i \,||\, k) \bmod n$ for $i \in I$, where H is a cryptographic hash function (when modeled as a random function), $||$ denotes concatenation and the size of the range of H is a multiple of n. Another choice is AES, when used as a pseudo-random permutation.

Suppose Alice wants to send a value $m \in \mathbb{Z}_n$ to Bob. Then Alice can pick an arbitrary, unique, number $i \in I$ – which we call the *identifier* – and encrypt the message

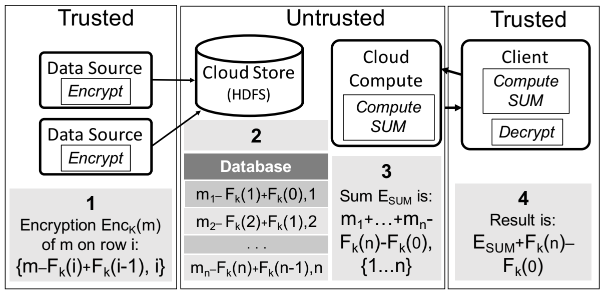

Trusted
Data Source
Encrypt
Data Source
Encrypt
1
Encryption Enc$_K$(m) of m on row i:
{m–F$_k$(i)+F$_k$(i-1), i}
Untrusted
Cloud Store
(HDFS)
2
Database
m$_1$– F$_k$(1)+F$_k$(0),1
m$_2$– F$_k$(2)+F$_k$(1),2
. . .
m$_n$–F$_k$(n)+F$_k$(n-1),n
Cloud Compute
Compute SUM
3
Sum E$_{SUM}$ is:
m$_1$+…+m$_n$-F$_k$(n)-F$_k$(0),
{1...n}
Trusted
Client
Compute SUM
Decrypt
4
Result is:
E$_{SUM}$+F$_k$(n)–F$_k$(0)

Figure 2: Seabed components and the ASHE scheme.

by computing:

$$\mathsf{Enc}_k(m, i) := ((m - F_k(i) + F_k(i-1)) \bmod n, \{i\})$$

In other words, the ciphertext is a tuple (c, S), where c is an element of the group $\mathbb{Z}_n$ and S is a multiset of identifiers. Note that the ciphertext c consists of the plaintext value m plus some pseudo-random component, hence it appears to be random to anyone who does not know the secret key k.

To create the additive homomorphism, we define a special operation $\oplus$ for "adding" two ciphertexts:

$$(c_1, S_1) \oplus (c_2, S_2) := ((c_1 + c_2) \bmod n, S_1 \cup S_2)$$

That is, the group elements are added together and the multisets of identifiers are combined. To decrypt the ciphertext, Bob can simply compute

$$\mathsf{Dec}_k(c, S) := (c + \sum_{i \in S} (F_k(i) - F_k(i-1))) \bmod n$$

Thus, after the homomorphic operation,

$$\mathsf{Dec}_k(\mathsf{Enc}_k(m_1, i_1) \oplus \mathsf{Enc}_k(m_2, i_2)) = (m_1 + m_2) \bmod n$$

Figure 2 gives a high-level overview of ASHE in the context of Seabed. We show that the above scheme satisfies the standard notion of semantic (CPA) security in Appendix A.

3.2 Optimizations for ASHE

The reader may wonder why the first element of the ciphertext is computed as $(m - F_k(i) + F_k(i-1)) \bmod n$ and not simply as $(m - F_k(i)) \bmod n$. The reason is that we have optimized ASHE for computing aggregations on large data sets. Suppose, for instance, that Alice wants to give Charlie a large table of encrypted values, with the intention that Charlie will later add up a range of these values and send them to Bob. Then Alice can simply choose the identifiers to be the row of numbers $(1, 2, \ldots, x)$. Later, if Bob receives an encrypted sum (c, S) with $S = \{i, \ldots, i+t\}$ (i.e., the sum of rows i to $i+t$), he can decrypt it simply by computing $(c + F_k(i+t) - F_k(i-1)) \bmod n$, since the other F_k values will cancel out. Thus, it is possible to decrypt the sum of a range of values by evaluating the PRF only twice, regardless of the size of the range.

Other optimizations including managing ciphertext growth and use of AES encryption support in hardware for efficient PRF computation are discussed in Section 4.

Plaintext Schema

country	salary
Male	1000
Female	2000
Female	200

Encrypted Schema

sender	salary
DET(Male)	ASHE(1000)
DET(Female)	ASHE(2000)
DET(Female)	ASHE(200)

Schema with Basic SPLASHE

genderMale	genderFemale	salaryMale	salaryFemale
ASHE(1)	ASHE(0)	ASHE(1000)	ASHE(0)
ASHE(0)	ASHE(1)	ASHE(0)	ASHE(2000)
ASHE(0)	ASHE(1)	ASHE(0)	ASHE(200)

Figure 3: SPLASHE instead of deterministic encryption.

3.3 Basic SPLASHE

SPLASHE is motivated by frequency attacks on deterministic encryption [36]. Recall that, unlike ASHE, in deterministic encryption, there is only one possible ciphertext value for each plaintext value. This enables the server to perform equality checks but also reveals frequency of items. The attacker combines the frequency of ciphertexts with auxiliary information to decode them.

We begin by describing a basic version of our approach. Consider a column C_1 that can take one of d discrete values and let the value of C_1 in row t be $C_1[t]$. If we anticipate counting queries of the form `SELECT COUNT(`C_1`) WHERE` C_1`=x`, we can replace the column C_1 with a family of columns $C_{1,1}, \ldots, C_{1,d}$. When the value of $C_1[t]$ is v, we set $C_{1,v}[t] = 1$ and set $C_{1,w}[t] = 0$ for $w \neq v$. If the resulting columns are encrypted using ASHE, the ciphertexts will look random to the adversary, but it is nevertheless possible to compute the count: we can simply rewrite the above query to `SELECT SUM(`$C_{1,x}$`)` and then compute the answer using homomorphic addition.

A similar approach is possible for aggregations. Consider a pair of columns C_1 and C_2, where C_1 again takes one of d discrete values and C_2 contains numbers that we might later wish to sum up using a predicate on C_1 (and possibly other conditions). In other words, we anticipate queries of the form `SELECT SUM(`C_2`) WHERE` C_1`=x`. In this case, we can split C_2 into d columns $C_{2,1}, \ldots, C_{2,d}$. When $C_1[t] = v$, we set $C_{2,v}[t] := C_2[t]$ and set $C_{2,w}[t] := 0$ for $w \neq v$. C_1 and C_2 can then be omitted. Thus, the above query can be rewritten into `SELECT SUM(`$C_{2,x}$`)`, which can be answered using homomorphic addition. An example of SPLASHE is shown in Figure 3 for C_1 as `Gender` and C_2 as `Salary`.

3.4 Enhanced SPLASHE

Basic SPLASHE increases a column's storage consumption by a factor of d, which is expensive if d is large. Next, we describe an enhancement that addresses this.

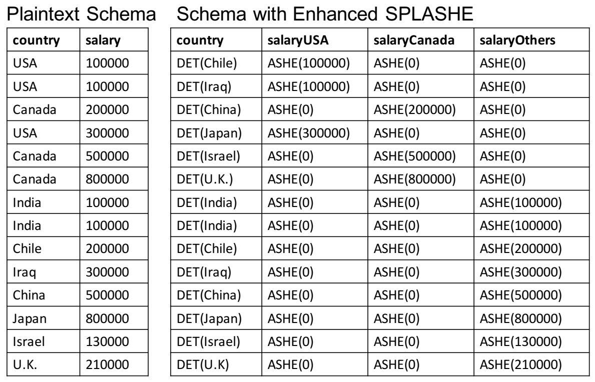
Consider again a pair of columns C_1 (say, `country`) and C_2 (say, `salary`), where C_1 takes one of d discrete values and C_2 contains numbers that we might later wish to sum up using a predicate on C_1. Suppose k of the d values are common (e.g., a Canadian company with offices worldwide but with most employees located in `USA` or `Canada`; $k = 2$, $d = 196$). Then we can replace C_2 by $k + 1$ columns – one for each of the common values (`salaryUSA` and `salaryCanada`) and a single column for the uncommon values (`salaryOther`). Figure 4 shows an example. As before, for each row, we place the ASHE encrypted value of salary from C_2 in the appropriate salary column, while we fill the other k salary columns with ASHE-encrypted zeros. We then encrypt C_1 *deterministically* for each of the uncommon countries to enable equality checks against encrypted values.

At this point it is possible to compute aggregations on C_2 for all values v of C_1: if the value v is common (`USA` or `Canada`), we can compute a sum over the special column for v; otherwise we can select the rows where `country` in C_1 equals the deterministically encrypted value of v and compute the sum over `salaryOther`.

However, C_1 now is susceptible to frequency attacks. To prevent this, in C_1, we ensure that *all ciphertexts occur at the same frequency*. How is this possible? Note that the cells corresponding to common countries in C_1 were so far unused. We can reuse these cells to normalize the frequency count of the uncommon countries. For these reused cells, since the corresponding values in the `salaryOther` column are set to ASHE encrypted values of zero, this approach preserves correctness while preventing frequency attacks.

When is this approach possible? Let $n_1 \geq n_2 \ldots \geq n_d$ be the number of occurrences of each of the d values. Then the number of splayed columns should be chosen to be the minimum k such that $\sum_{i=1}^{k} n_i \geq \sum_{i=k+1}^{d} (n_{k+1} - n_i)$: this is because $\sum_{i=1}^{k} n_i$ are enough unused cells in column C_1 that can be used to make the number of occurrences of all non-splayed values at least n_{k+1}. Such a k will always exist; the more heavily skewed the distribution of values is, the smaller the k will be, and the more storage will be saved. This approach can be followed even if the exact number of occurrences is unknown; we do, however, need to know the distribution of the values.

Figure 4 shows an enhanced SPLASHE example with $k = 2$ and $d = 9$. Notice how the first six rows of the deterministically encrypted column have been reused to equalize the frequency of all elements in that column while still ensuring the correctness of aggregation queries on any of the country predicates.



Plaintext Schema

country	salary
USA	100000
USA	100000
Canada	200000
USA	300000
Canada	500000
Canada	800000
India	100000
India	100000
Chile	200000
Iraq	300000
China	500000
Japan	800000
Israel	130000
U.K.	210000

Schema with Enhanced SPLASHE

country	salaryUSA	salaryCanada	salaryOthers
DET(Chile)	ASHE(100000)	ASHE(0)	ASHE(0)
DET(Iraq)	ASHE(100000)	ASHE(0)	ASHE(0)
DET(China)	ASHE(0)	ASHE(200000)	ASHE(0)
DET(Japan)	ASHE(300000)	ASHE(0)	ASHE(0)
DET(Israel)	ASHE(0)	ASHE(500000)	ASHE(0)
DET(U.K.)	ASHE(0)	ASHE(800000)	ASHE(0)
DET(India)	ASHE(0)	ASHE(0)	ASHE(100000)
DET(India)	ASHE(0)	ASHE(0)	ASHE(100000)
DET(Chile)	ASHE(0)	ASHE(0)	ASHE(200000)
DET(Iraq)	ASHE(0)	ASHE(0)	ASHE(300000)
DET(China)	ASHE(0)	ASHE(0)	ASHE(500000)
DET(Japan)	ASHE(0)	ASHE(0)	ASHE(800000)
DET(Israel)	ASHE(0)	ASHE(0)	ASHE(130000)
DET(U.K)	ASHE(0)	ASHE(0)	ASHE(210000)

Figure 4: Enhanced SPLASHE example.

The reader can find a more detailed description of enhanced SPLASHE's security properties in Appendix A. Briefly, enhanced SPLASHE satisfies simulation-based security; the adversary learns only the number of rows in the database, and the number of infrequently and frequently occurring values.

3.5 Limitations

ASHE: Homomorphic encryption schemes have traditionally been defined with a *compactness* requirement, which says that the ciphertext should not grow with the number of operations that are performed on it. This is done to rule out trivial schemes: for instance, one could otherwise implement an additive "homomorphism" by simply concatenating the ciphertexts $Enc(m_1)$ and $Enc(m_2)$ and then have the client do the actual addition during decryption. ASHE does not strictly satisfy compactness, but the evaluator (the cloud) still does perform the bulk of the computation on ciphertexts; also, the techniques in Section 4 ensure that the length of ASHE's ciphertexts does not grow too much.

In terms of performance, growing ciphertexts can create memory stress at the workers. In the case of a system without encryption, the worker nodes only need enough memory to hold the dataset. When using ASHE, the workers need to have some extra memory to construct the ID lists. This should not be a big problem in practice: as we will show in Section 6, the overhead is small enough for real-world big data applications that involve billions of rows. Nevertheless, this extra memory requirement can become a problem if workers have very limited memory, or if the dataset is very large (e.g., if it has trillions of records).

SPLASHE: SPLASHE has three main drawbacks: (1) its requirement for a-priori knowledge of query workload

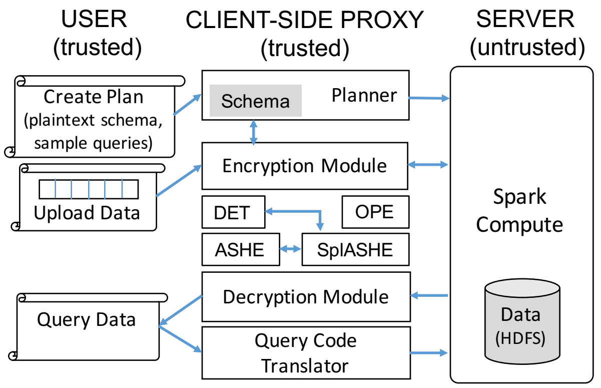


Figure 5: Seabed system design

or data distribution, (2) its difficulty in handling data with rapidly changing distribution, and (3) its storage overhead.

First, SPLASHE requires knowing what the expected query workload is. This is because we need to confirm that the splayed column will not participate in joins or inequality predicates – for such cases we need to fall back to deterministic encryption (DET). In addition, to get the storage reduction of enhanced SPLASHE, we need to know the distribution of values that a column can take. If this information is not available, only basic SPLASHE can be used.

Second, enhanced SPLASHE is most appropriate for columns whose distribution does not change dramatically. For columns whose distribution fluctuates significantly, data insertions will start skewing the distribution of the DET column (C_1 in our example) away from the uniform distribution SPLASHE constructs. This happens because a significant change in distribution will require reusing more cells than those available in the rows that were previously common. However, even in such an extreme case, SPLASHE is still better than using plain DET; DET reveals the exact distribution of values, whereas SPLASHE reveals a noised version of it.

Finally, both basic and enhanced SPLASHE increase storage needs. Section 6.6 shows that a real-world ad analytics database can be supported with enhanced SPLASHE at a storage overhead of about 10x.

4 Design

We now provide a functional overview of Seabed, and then describe each system component in more detail. For simplicity, we describe the design using the example of only one data source and one client. In practice, multiple data sources and users can share the same system as long as they share trust.

4.1 Roadmap

Figure 5 shows the major components of Seabed. A user interacts with the Seabed client proxy that runs in a trusted environment. The proxy in turn interacts with the untrusted Seabed server. As with previous systems, Seabed is designed to hide all cryptographic operations from users, so they interact with the system in the same way as they would with a standard Spark system. The user can issue three kinds of requests:

Create Plan: First, the user supplies a plaintext schema and a sample query set to the Seabed planner. The planner uses these and the procedure specified in Section 4.2 to determine the encryption schemes for the columns.

Upload Data: Next, the user sends plaintext data to the Seabed encryption module described in Section 4.3. The data is encrypted with the required encryption scheme and records are appended to the table stored in the Cloud. This is a continuing process; database insertions are handled in the same way.

Query Data: During analysis, the user sends a query script to the Seabed query translator, which modifies queries to run on encrypted data before sending them to the server (Section 4.5). The server runs the queries and responds to the proxy's decryption module (Section 4.6). After decryption and further processing (if any), the results are sent back to the user.

4.2 Data Planner

The data planner determines how to encrypt each column in the schema, given a list of sensitive columns by the user. The user also supplies a sample query set, which is used by the planner to decide on the encryption algorithms. In addition, to use enhanced SPLASHE, the user provides the number of distinct values each column can take and the frequency distribution of these values.

By parsing the sample query set, the planner first classifies each sensitive column as a *dimension*, a *measure*, or both. A measure is a column (e.g., `Salary`) over which a query computes aggregate functions, such as sum, average and variance. A dimension is a column (e.g., `Country`) that is used to filter rows based on a specified predicate before computing aggregates. After the classification, the planner uses the following strategies to determine which encryption schemes to use.

ASHE: If a sensitive measure is aggregated using linear functions, such as sum and average, we encrypt it using ASHE. If a sensitive measure is aggregated using quadratic functions (e.g., variance), we compute the square of the values on the client side and add it to the database as a separate column, so it can be used in computations on the server side. Whenever we use ASHE on a column, we give a unique ID to each row, which is used in the encryption as discussed in Section 3.1; to enable

Operation	Time (nanoseconds)
AES counter mode	47
Paillier encryption	5,100,000
ASHE encryption/decryption	12-24
Plain addition	1
Paillier addition	3800
Paillier decryption	3,400,000

Table 1: Cost of operations on a 2.2 GHz Xeon core.

compression, we assign consecutive row IDs. We choose a different secret key k for each new column we encrypt.

SPLASHE: If a sensitive dimension is used in filters, and if no query uses joins on this dimension, then the dimension is a candidate for SPLASHE. However, given the storage costs, we determine whether to use SPLASHE for the dimension as follows. First, we determine the measure columns that are used in conjunction with this dimension in the queries: only these measure columns need to be SPLASHE-encrypted. Based on this subset of measure columns, the planner uses the algorithm described in 3.4 to compute the storage overhead. Then, if a user specifies a maximum storage overhead, the planner prioritizes the dimensions that use SPLASHE based on their cardinality (lowest cardinal dimension first, in order to maximize protection against frequency attacks). We show how this approach works with a real dataset in Section 6.6.

DET or OPE: If a sensitive dimension cannot use SPLASHE – say, because it is used as part of a join – we warn the user and then use deterministic encryption (DET). If the dimension requires range queries in query filters, then we use order-preserving encryption (OPE). We require an OPE scheme that works on dynamic data and hence the OPE scheme of CryptDB [39] is not suitable in our case. We use the recent scheme from [21], which is efficient (based on any PRF) and has low leakage: for any two ciphertexts, in addition to the order of the two underlying plaintexts, it reveals the first bit where the two plaintexts differ and nothing more. For more details, please refer to Appendix A.

Note that some queries (such as averages) cannot be directly executed on the server because they are not supported by Seabed's encryption schemes. In such cases, the Seabed planner borrows techniques from prior work [44] to divide the query into a part the server *can* compute (e.g., a sum and a count), and a part that the client/proxy will need to compute after decryption (e.g., the final division).

4.3 Encryption Module

The Encryption Module encrypts plaintext records into the encrypted schema. Note that ASHE encryption and decryption are quite lightweight compared to Paillier operations. As shown in Table 1, one AES counter operation (implemented using hardware support on a Intel Xeon 2.2GHz processor) takes 47 ns whereas one Paillier encryption takes 5.1 ms, a difference of *five orders of magnitude*. Hence, by using ASHE instead of Paillier, we reduce the encryption load on the client significantly.

We optimize ASHE encryption and decryption further by using a single AES operation to generate multiple ciphertexts. Each AES operation works on 128-bit vectors. Numeric data types are typically much smaller: 32-bit or 64-bit integers are common. One AES operation can therefore generate two or four pseudo-random numbers for 64-bit or 32-bit data types, respectively.

Also, note that unlike conventional cryptographic techniques, ASHE encryption and decryption are inherently parallelizable because multiple AES operations can be computed simultaneously in a multi-core environment. We therefore run a multi-threaded version of the encryption and decryption algorithm, and this further reduces latency.

If the system needs a way to revoke the access privileges of individual users, the proxy can additionally implement an access control mechanism, analogous to the approach in CryptDB. Typically, revocation is difficult when symmetric encryption schemes are used: once a symmetric key is shared, the only way to invalidate it is to re-encrypt the data. However, since the proxy handles all queries, it does not need to share the secret keys with the clients, so it can revoke or limit their access without re-encryption.

4.4 Query Translator

The goal of the Query Translator is to intercept the client's unmodified queries, and rewrite them in a way appropriate for the schema of the encrypted dataset. Our design follows the principles introduced by CryptDB and Monomi: we encrypt constants with the appropriate encryption scheme, and we replace operators with the custom functions that implement ASHE aggregation, or DET/OPE checks. One technical difference to the previous systems is that these operated on relational databases, so both the source and target language of the translator was SQL. However, Seabed works on Spark, so the target language is Scala and the Spark API.

The Seabed Query Translator makes three additions to the query rewriting process to accommodate the new encryption schemes it uses; we show examples for all three in Table 2. First, the schema of the encrypted dataset in Seabed includes an additional ID column. This column is necessary for ASHE aggregation, so the Query Translator preserves it even if the client has not explicitly done so in the projection fields of the original SQL query. That way, Seabed can support aggregation on the result of sub-queries. Second, for columns that use SPLASHE, Seabed follows the rules outlined in Section 3 to rewrite

Query type		Query
ID preservation	SQL	`SELECT sum(tmp.a) FROM (SELECT a FROM table WHERE b > 10) tmp`
	Spark API	`table.filter(x => x(2) > 10).map(x =>x(1)).reduce((x,y) => x+y)`
	Seabed	`table.filter(x => OPE.leq(x(2),Enc`$_{OPE}$`(10)).` `map(x =>(x(id), x(1))).reduce((x,y) => ASHE(x,y))`
SPLASHE	SQL	`SELECT count(*) FROM table WHERE a = 10`
	Spark API	`table.filter(x=>x(1) == 10).count()`
	Seabed	`table.map(x=>(x(id),x(3))).reduce((x,y)=>ASHE(x,y))`
Group-by optimization (and ID preservation)	SQL	`SELECT a, sum(b) FROM table GROUP BY a`
	Spark API	`table.map(x=>(x(1),x(2)).reduceByKey((x,y)=>x+y)`
	Seabed	`table.map(x=>(x(1)+":"+r.nextInt%10,(x(id),x(2))).` `reduceByKey((x,y)=>ASHE(x,y))`

Table 2: Examples of query translation. $x(1)$ corresponds to table column a, $x(2)$ to b, $x(3)$ to splayed a for value 10, and $x(id)$ to the identifier column used by ASHE.

Technique	Example	
	Integer/List	Encoding
Range encoding	[2. . .14,19. . .23]	[2-14,19-23]
Diff. encoding	[2,3,4,9,23]	[2,1,1,5,14]
Combination	[2. . .14,19. . .23]	[2-12,5-4]
VB-encoding	Encoded with minimum #bytes	

Table 3: ID list encoding techniques used in Seabed.

queries. This implies that the client has to maintain a small data structure with information about the splayed fields. Finally, if the client enables our group-by optimization, which is described in Section 4.5, the Query Translator may also modify the group-by fields of the query. This requires that the client maintains some state about the expected number of groups in a query result.

4.5 Seabed Server

Performing aggregations using ASHE requires the server to manage growing ciphertexts. This can result in need for large in-memory data structures and high bandwidth. We now describe how we optimize these overheads.

Reducing ID list size: To keep the size of the ID list small, we evaluated several integer list encoding techniques [34], including bitmaps [20], for good compression rates, low memory usage and high encoding speed. We eventually decided that a combination of the techniques listed in Table 3 were the most appropriate for Seabed. We begin with range encoding, which compresses contiguous sequences of integers by specifying the lower and upper bound. Next, we apply differential (Diff) encoding, which replaces the (potentially large) individual numbers with the (hopefully small) difference to the previous number; the result of this second step is labeled “Combination” in Table 3. Finally, we apply variable-byte (VB) encoding, which uses fewer bytes to represent smaller numbers.

Variable-byte (VB) and differential encoding (Diff) strike a nice balance between performance and compression and can be efficiently implemented in software. Range encoding, i.e. describing contiguous integers by specifying the bounds of their range, is not widely used in the literature because it can bloat up lists of non-contiguous integers. In Seabed, though, data is uploaded to the server with contiguous IDs, so range encoding can provide great benefits, especially for queries that select a large portion of a dataset. In Section 6.4, we show how combining VB, Diff, and range encoding reduces the size of the ID list and speeds up aggregation.

Reducing server-to-client traffic: Every Spark job consists of one driver node and several worker nodes. The workers send their partial results to the driver which then aggregates and sends the combined result to the client. To further reduce the size of ID lists, we applied standard compression. However, there are two options here: applying compression at the worker nodes or applying compression after aggregation at the driver node. The latter can lead to higher compression rates, but we found that this caused a bottleneck at the driver. Instead, we found that applying compression at each of the worker nodes benefits from parallelization and results in lower overall latency.

Handling group-by queries: Group-by queries are in general challenging for ASHE, because all row IDs are included in the final result, which can grow quite large. Moreover, using range encoding seems to incur unnecessary costs for group-by queries: when the result of a group-by query contains many groups, the ID lists of each group tend to be very sparse. As we noted earlier, range encoding is wasteful for sparse ID lists, so we decided to use only VB and Diff encoding for group-by queries.

Group-by queries lead to one more complication: when the number of groups in the result is small, the traffic between mapper and reducer workers becomes a bottleneck. There are two underlying reasons for this. First, with few groups, the ID list of each group becomes denser, and not using range encoding starts to show up. Second, when the number of groups is less than

Query set	Total	Purely on Server	Client Pre-processing	Client Post-processing	Two Round-trips
Ad Analytics	168,352	134,298	0	34,054	0
TPC-DS	99	69	2	25	3
MDX	38	17	12	4	5

Table 4: Different categories of queries that Seabed supports.

Dataset	Rows	Dimen-sions	Measu-res	Disk size (GB)			Memory size (GB)		
				NoEnc	Seabed	Paillier	NoEnc	Seabed	Paillier
Synthetic - Large	1.75B	-	1	35.4	70.4	521.1	84.7	121.9	638.6
Synthetic - Small	250M	-	1	5	9.8	74.2	12.1	17.7	91.4
BDB - Rankings	90M	1	2	7.9	12	58.3	18.6	28.1	80.4
BDB - User Visits	775M	8	2	194.9	287.5	673.6	581	832.5	1269.4
BDB - Query 4, Phase 2	194M	2	1	35	38.3	88.3	73.5	86.5	140
Ad Analytics	759M	33	18	132.3	142.45	176.3	1004	1027.3	1254.4

Table 5: Characteristics of our synthetic dataset, the Big Data Benchmark (BDB) and the Ad Analytics dataset (AdA).

the available workers, some reducers will remain idle in the reduce phase. This means that more data (because of denser ID lists) is shuffled between fewer workers (because of idle workers). This can create a bottleneck for very large datasets where ID lists are large.

To make use of more worker nodes in the reduce phase and to mitigate the above effect, we artificially increase the number of returned groups. We accomplish this by appending a random identifier to each value of the group-by column. For example (table 2), if a query returns 10 groups $\{g_1, \dots, g_{10}\}$, and there are 100 workers available, then we can append a random identifier to the group-by column, which takes values from 0 to 9. This means that the result will contain $10 * 10 = 100$ groups $\{g_1{:}0, \dots, g_1{:}9, \dots, g_{10}{:}0, \dots, g_{10}{:}9\}$, the computation will utilize all available workers in the reduce phase, and we will avoid the bandwidth bottleneck. Of course, the client has to perform the remaining aggregations to compute the sum of the actual groups (e.g., add results for groups $\{g_1{:}0, \dots, g_1{:}9\}$ to get the result for group g_1). As a heuristic, we inflate the number of groups to the number of available workers when we expect fewer groups than workers.

4.6 Decryption Module

The Decryption Module uncompresses the ID lists, uses the techniques from Section 4.3 to calculate the pseudorandom numbers to add to the encrypted value, and returns the result to the user. If the query has some part that cannot be computed at the server, the Decryption Module can additionally perform that part before presenting the final answer to the user. Since we have assumed that the adversary is honest but curious, the Decryption Module performs no integrity checks; thus, an active adversary could return bogus data without being detected by Seabed itself.

The decryption cost of ASHE depends on the number of aggregated elements; this is different from Paillier, which requires only one decryption for each aggregate result. However, Paillier decryption is five orders of magnitude slower than ASHE decryption (Table 1), and the overall client decryption costs for Seabed remain smaller than Paillier (Section 6).

5 Applications

An important question is whether Seabed supports a wide range of big data analytics applications. To understand this, we performed three studies. First, we systematically analyzed two common interfaces that BI applications use at the back-end: MDX (the industry standard) and Spark. Second, we evaluated a month-long query log made on a custom-designed advertising analytics OLAP platform to determine how effectively Seabed can support the functionality of these systems. Finally, we analyzed the TPC-DS query set. Detailed results of our MDX/Spark analysis can be found in Appendix B. Briefly, the analysis revealed that Seabed's functionality support falls into four categories:

Support fully on the server: Seabed's encryption techniques can fully support operations with no client support. Examples of such operations are computing the sum, average, count, and min.

Support with client pre-processing: Seabed can support quadratic computation necessary for more complex analytics such as anomaly detection, linear regression in one dimension, and decision trees that are supported by Watson Analytics [7] and Tableau [6]. To support this, the Seabed client has to compute squared values of the necessary columns, and encrypt them with ASHE.

Support with client post-processing: All applications and APIs we studied allow users to specify arbitrary functions of data. When these functions are complex, Seabed cannot perform them at the server and data has to be post-processed at the client. This is similar to how Monomi splits queries into server- and client-side components.

Support with two client round-trips: Some queries require the client to compute an intermediate result, re-encrypt it and send it back to the server for further processing.

Table 4 shows the numbers of queries that fall into these categories for the three query sets we analyzed. We analyzed the MDX API/TPC-DS query set manually; for the ad analytics query set, we used heuristics based on the query structure. For Ad Analytics and TPC-DS, about 75-80% of the queries can be supported purely on the server. This implies that these query sets mostly use simple aggregation functions. About 20-25% need client-side support. The TPC-DS query set and MDX API have a few queries (5-15%) that require two round-trips.

6 Evaluation

In this section, we report results from our experimental evaluation of Seabed. Table 5 summarizes the datasets used in our experiments. We evaluate the system with microbenchmarks (Synthetic), an advertising analytics data workload and query set (AdA), and the AmpLab Big Data Benchmark (BDB).

Our evaluation has two high-level goals. First, we evaluate the *performance benefits* of Seabed over systems that use the Paillier cryptosystem. Second, we quantify the *performance and storage overhead* incurred by Seabed as compared to a system with no encryption.

6.1 Implementation and Setup

We built a prototype implementation of Seabed on the Apache/Spark platform [2] (version 1.6.0). We chose Spark because of its growing user-base and performant memory-centric approach to data processing. The server-side Seabed library was written in Scala using the Spark API. The Seabed client uses Scala combined with a C++ cryptography module for hardware accelerated AES (with Intel AES-NI instructions). We implemented Paillier in Scala using the `BigInt` class. Data tables are stored in HDFS using Google Protobuf [3] serialization. In total, our Seabed prototype consists of 3,298 lines of Scala and 2,730 lines of C++.

Our experiments were conducted on an Azure HDInsight Linux cluster. The cluster consists of tens of nodes, each equipped with a 16-core Intel Xeon E5 2.4 GHz processor and 112 GB of memory. Machines were running Ubuntu (14.04.4 LTS) and job scheduling was done through Yarn. In our experiments, we compare the following system setups:

NoEnc: Original Spark queries over unencrypted data,

Paillier: Modified Spark queries over encrypted data; measures are encrypted using Paillier, and dimensions with DET and/or OPE, and

Seabed: Modified Spark queries over encrypted data; measures are encrypted using ASHE, and dimensions with DET and/or OPE.

For our microbenchmarks, we generated a synthetic dataset (see Table 5). The NoEnc and Paillier datasets consist of one column of plaintext integers and 2048-bit ciphertexts, respectively. The ASHE dataset consists of two columns: an ID and an integer value encrypted with ASHE (IDs are contiguous). In order to model predicates that choose selected rows of a table, we use a parameter called *selectivity* that varies between 0 and 1 and use it to choose *each row* randomly with the corresponding probability. Note that this random selection model allows us to study the various system trade-offs in these schemes, e.g., the total length of ID lists, and it also enables us to understand the worst-case behavior. (At first glance, a query that selects all even or odd rows may appear to be the worst case for Seabed, since range encoding with such a non-contiguous set of IDs will double the size of the resulting ID list. However, in this case, the ID list is in fact highly compressible because the differences between consecutive IDs is always two, so stock compression techniques work very well.)

All experiments, unless otherwise mentioned, used 100 cores and 1.75 billion rows of input data. For end-to-end results, we place the client in one of the nodes in the same cluster as the server. Thus, by default, the client is connected by a high-speed, low-latency link to the server (TCP throughput of 2 Gbps). However, we also perform experiments by varying this bandwidth (using the `tc` command).

6.2 Microbenchmark: End to End Latency

We first compare end-to-end latency for the three approaches with varying input sizes (250 million to 1.75 billion rows). In Figure 6, we show the median latency after running 10 queries for each input size. For Seabed, we show two lines: one with selectivity 100% and the other with selectivity 50%. We shall show in Section 6.4 that the former gives best-case latency while the latter gives worst-case latency for Seabed. For NoEnc and Paillier, we use a selectivity of 100% (their performance is linear with respect to selectivity).

Figure 6(a) shows the results for NoEnc and Seabed.

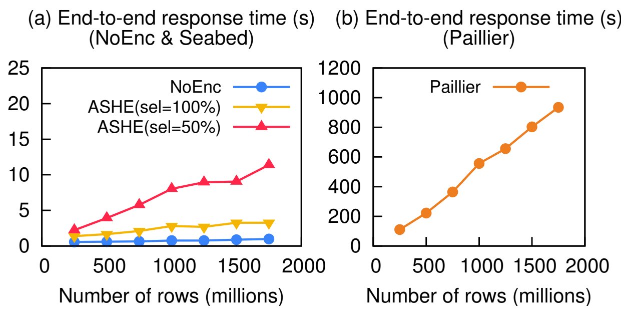


Figure 6: Median latency for aggregation vs data size.

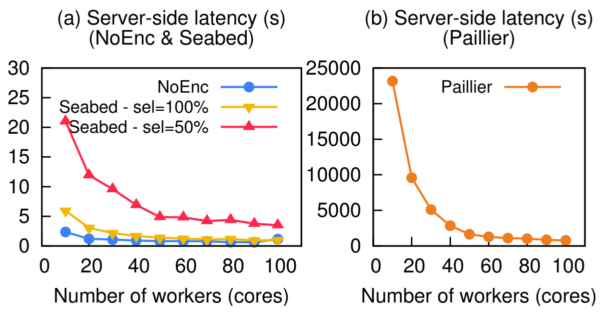


Figure 7: Median latency for aggregation vs cores.

NoEnc has a constant latency of approximately 0.6s. This is because addition is a simple operation and the overall latency is dominated by task creation costs. Seabed's aggregation is more complex, so latency for both Seabed selectivity 50% and 100% increases linearly with the dataset size. Nevertheless, the cost of aggregation in Seabed is still small even for large datasets, varying between 1.8s to 11s in the worst-case as the number of rows increase. On the contrary, Paillier results in a latency of over 1000s when aggregating 1.75 billion rows.

For Seabed selectivity 100%, about 80% of time is due to server-side compute, 20% is due to client-side decryption, and network latency is minimal. For Seabed selectivity 50%, the server-side contributes 55% of the latency, the decryption contributes 35% and network transfer contributes the remaining 10%.

We observed occasional stragglers, i.e., tasks that took longer to complete and delayed the entire job, for all three systems. The underlying cause of these stragglers was usually garbage collection being triggered at some node in the cluster. Paillier jobs took several hundreds of seconds to complete, so the comparative effect of stragglers was small. However, NoEnc and Seabed jobs took only few seconds at the server, so whenever there was a straggler task, the delay was more pronounced.

6.3 Microbenchmark: Server Scalability

One important aspect of big data systems is how they scale with larger clusters. Since using a larger cluster can only speed up the server side, we consider *server-side latency* as we evaluate Seabed's scalability. Fixing the dataset at 1.75 billion rows, we varied the number of cores from 10 to 100. Figure 7 shows how Seabed, NoEnc and Paillier scaled with the number of cores. NoEnc reached its best latency, which is approximately 1s, with 20 cores. Both Seabed selectivity 100% and Seabed selectivity 50% achieved their best latency of 1.35s and 8.0s respectively with only 50 cores. Even with 100 cores, Paillier's server latency was close to 1000s, which is more than two orders of magnitude higher than Seabed's. This implies that, for large datasets, Paillier would require increasing the number of cores by orders of magnitude in order to achieve latencies that are comparable to Seabed. Seabed's overhead over NoEnc primarily comes from managing the ID lists. Next, we look into this in more detail.

6.4 Microbenchmark: Seabed Overhead

In this section we examine the server-side overheads incurred by Seabed's ASHE and the use of OPE.

ASHE list construction: For ASHE, the server manages ID lists using a variety of compression techniques (Section 4). In this experiment, we show how these compression techniques perform. The bitmap algorithms performed poorly, so we omit them here for brevity. We varied selectivity from 10% to 100%, and we measured the size of the ID list and the server-side response time of the query. We report the results in Figure 8(a) and (b).

Figure 8(a) suggests that range encoding is very effective in bounding the length of the ID list: without it, the size of ID list would keep increasing as the selectivity of a query increases, whereas with ranges the list size starts decreasing after selectivity 50%. After this, IDs start to become more dense and therefore more consecutive, leading to best-case compression at selectivity 100%. We can also see that the combination of VB and Diff-encoding is very effective in reducing the size of the ID list, and Deflate compression [5] further reduces the size of the list.

The performance hit incurred by each encoding method is depicted in Figure 8(b). To our advantage, we found that, in all cases except with Deflate optimized for high compression ratio, the better-performing algorithms also provided more compressed ID lists. Based on the above, we picked the following combination of encodings as the ID list construction algorithm in Seabed: Range-encoding, VB encoding, Diff-encoding, and Deflate compression (optimized for speed). This is what we used for all the other experiments.

OPE: The OPE scheme we use introduces some over-

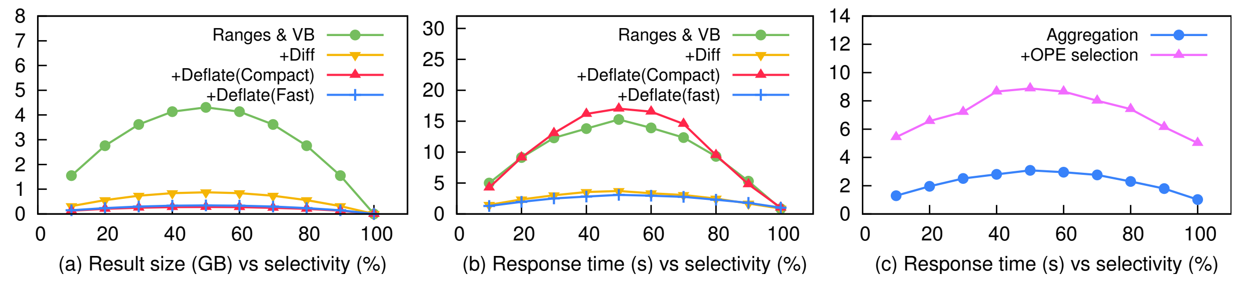


Figure 8: Result size and response time vs selectivity over 1.75 billion rows.

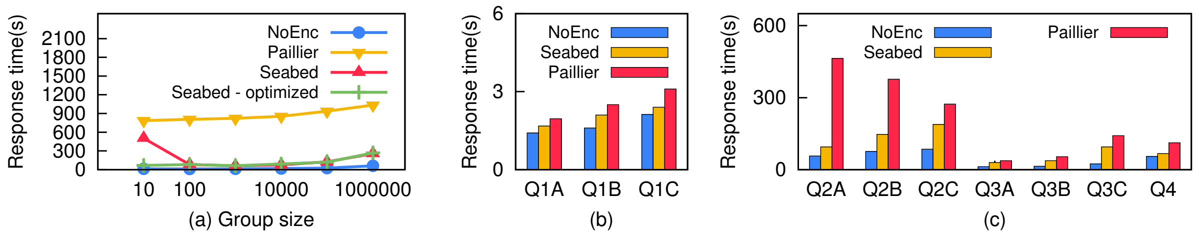


Figure 9: (a) Microbenchmark results for group-by queries. (b-c) Response time for the Big Data Benchmark queries.

head because comparison between OPE ciphertexts is not as fast as comparing two plaintext integers. This is because OPE comparison involves searching for the first bit position where two 64-bit integers differ.

To measure the cost of OPE, we used the same synthetic dataset as for ASHE with 1.75 billion rows, but we added one more integer column encrypted with OPE. We repeat the selectivity experiment above, but with the query performing an OPE comparison. Figure 8(c) indicates that OPE introduces more overhead, of about a factor of 5s, compared to the ASHE ID list construction.

6.5 Microbenchmark: Group-by

So far, we have evaluated only simple aggregation queries that involved minimal network communication: each Spark worker computes a sum and a compressed ID list per partition, and the reducers concatenate the lists into the final result. While aggregation is a major component of analytical query workloads, many queries also use the group-by operation, which causes more data to be shuffled across workers. In this section, we examine how Seabed performs for queries that involve group-by.

For this experiment, we used the synthetic dataset from the previous sections, but we added one more integer column. We then aggregated the value field while doing a group-by on the new column. We varied the number of groups from 10 to 1 million; Figure 9(a) shows the results.

The Seabed line shows the performance we get when we use VB and Diff-encoding for group-by queries. A very small number of groups in the result (10 in Fig. 9(a)) leads to increased latency because of the bandwidth bottleneck described in Section 4.5. The Seabed-optimized line shows that we can effectively deal with this inefficiency by artificially increasing the number of groups to 100 (Section 4.5).

Since all IDs are included in the result, Seabed group-by queries involve a significant amount of data shuffling. As a consequence, the benefits Seabed enjoys when compared to Paillier are lower. Yet, Seabed (optimized) does seem to be faster than Paillier by 5x to 10x. As the number of groups increases, Seabed's gain over Paillier drops from 10x to 5x. This is because the network shuffle time becomes a more significant part of the server response time. This indicates that Seabed will be less effective for group-by queries with a huge number of groups (hundreds of millions), something we observe in Section 6.6.

6.6 Ad-Analytics Workload

To assess the performance of Seabed on real-world data and queries, we evaluated it using the AmpLab Big Data Benchmark [1] and using a real-world large-scale advertising analytics application. We begin with a discussion of the latter.

For this series of experiments, we used data from an advertising analytics application deployed at an enterprise. This application is used by a team of experts for analytical tasks such as determining behavioral trends of advertisers, understanding ad revenue growth, and flagging anomalous trends in measures such as revenue and number of clicks. The data characteristics are shown in Table 5. We also obtained a set of queries that were per-

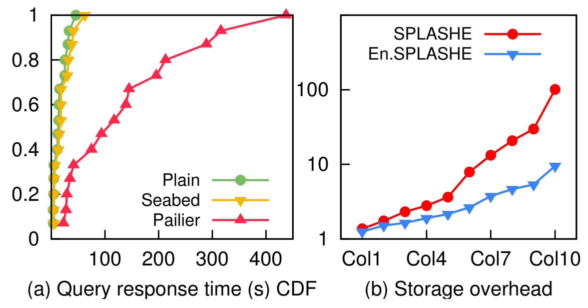


Figure 10: Seabed on the Ad Analytics workload: (a) query response-time CDF and (b) storage overhead due to SPLASHE.

formed for this application; this set consists of 168,352 queries issued between Feb 1, 2016 and Feb 25, 2016. The queries are all aggregations that calculate sums of various measures while grouping by timestamp (hour-of-day). The number of groups in a typical query is quite small, varying between 1 and 12 in most cases.

Performance: We first evaluated Seabed's performance on this dataset. We pick a set of 15 queries: five queries each for groups of size 1, 4, and 8. We ran each query ten times, and we calculated the median response time per query. All experiments were run with 100 cores.

Figure 10(a) shows the cumulative distribution function of response times for NoEnc, Seabed and Paillier. Seabed's response time ranges from 1.08 to 1.45 times that of NoEnc. The median response time for Seabed is 17.8s, whereas for NoEnc it is 13.8s. Thus Seabed's response time is only 27% higher than NoEnc's. On the other hand, the median response time for Paillier is 6.7× that of Seabed. To understand this result in more detail, we looked at the characteristics of the query responses. The average number of rows aggregated for a query across all groups was 210 million, the average size of the ID list was only 163.5KB, and the average number of AES operations required for decryption was roughly 26,000. This shows that there is a lot of contiguity of IDs in the ASHE ciphertext lists. Therefore, while queries could theoretically choose rows at random and thus create huge ID lists, our real-world dataset shows that this does not necessarily happen in practice: the data is stored in a certain order, and Seabed benefits from that order.

In all our experiments, the Seabed client used a high-bandwidth link to connect to the server. To measure the effect of lower-bandwidth and higher-latency links, we artificially changed the network bandwidth/latency between server and client to 100Mbps/10ms and 10Mbps/100ms. This increased the median response time by only 1% in the former case and 12% in the latter case, as the ID lists that need to be transferred are quite small.

Storage: We also used this dataset to quantify SPLASHE's overall storage overhead. Through conversations with operators, we determined that 10 out of 33 dimensions and 10 out of 18 measures require encryption. We used the procedure outlined in Section 3.4 to calculate the storage overhead for these 10 dimensions.

Figure 10(b) shows the cumulative storage overhead for each of the 10 dimensions in our dataset, sorted by the number of unique values in the dimension. The graph shows that if we restrict the storage overhead to a factor of two, we can encrypt only one dimension with Basic SPLASHE, whereas we can encrypt two dimensions with Enhanced SPLASHE. With a storage overhead of three, we can encrypt only three dimensions with Basic SPLASHE, whereas we can encrypt 6 with Enhanced SPLASHE. In this case, roughly 92% of all queries involve at least one column that uses enhanced SPLASHE.

6.7 AmpLab Big Data Benchmark

The AmpLab benchmark includes four types of queries (scan, aggregation, join and external script). Some of them come in different variants based on the result/join size, so there are ten queries in total. For this experiment we used 32 cores and loaded the entire Big Data Benchmark dataset (table 5) into the workers' memory. We measured the time to perform the query and store the results back into cache memory. Since the Big Data Benchmark is not designed for interactive queries, most of the result sets are huge and cannot fit into one machine's memory. Hence, for this section we do not measure the client-side cost of any of the compared systems.

We had to make a few simplifications to the query set in order to support it. Queries 2 and 4 require substring-search over a column and a text file, respectively. Existing searchable encryption techniques do not efficiently support this operation. Hence we simplified query 2 by matching over deterministically encrypted prefixes, and we simplified query 4 by keeping the text file as plaintext. Query 3 involves sorting based on aggregated values; since this can only be done on the client, and given that we measured only server-side overhead in this experiment, we omitted the sorting step.

Figure 9(b) shows the results. Query 1 does not use group-by or aggregation, so all tested systems had much faster response times. Both Seabed and Paillier were slower than NoEnc because of OPE overheads. On the remaining queries Seabed was consistently faster than Paillier, though not as much as we had shown in Sections 6.2 and with the Ad Analytics workload. This is because the queries results contained millions of groups and, as we saw in Section 6.5, Seabed is slower on result sets with a very small or a very large number of groups. Nevertheless, the results show that Seabed is better than

Paillier even for these workloads and is close to NoEnc performance for most queries.

7 Related Work

Homomorphic Encryption. Homomorphic encryption allows computations to be performed on encrypted data such that the computed result, when decrypted, matches the result of the equivalent computation performed on unencrypted data. The first construction of a fully homomorphic scheme that allows arbitrary computations on encrypted data was shown in [26]. However, fully homomorphic schemes are far from practical even today. For example, the amortized cost of performing AES encryption homomorphically is about 2s [28] but this is still 10^8 times slower than AES over plain text (Section 4).

There are also partially homomorphic schemes that allow selected computations on encrypted data. For example, Paillier [38] allows addition of encrypted data while BGN [18] supports one multiplication and several additions. However, these schemes incur significant cost in terms of both computation and storage space. Algorithms to reduce storage overhead by packing multiple integer values into a single Paillier encrypted value are proposed in [25] and implemented in [44].

Encrypted databases. CryptDB [39] leverages partially homomorphic encryption schemes to support SQL queries efficiently over encrypted data, and Monomi [39] introduced a split client-server computation model to extend support for most of the TPC-H queries over encrypted data. However, as we show in this paper, the partially homomorphic encryption schemes used in CryptDB and Monomi are not efficient enough to support interactive queries when applied to large datasets.

Trusted hardware. Hardware support for trusted computing primitives, such as Intel SGX [35], secure coprocessors [30], and FPGA-based solutions [9], are available today. These solutions allow client software to execute in the cloud without providing visibility of client data to the cloud OS. Several prior systems – such as Cipherbase [9], TrustedDB [11], M2R [24] and VC3 [41] – rely on secure trusted hardware to provide privacy-preserving database or MapReduce operations in the cloud.

The use of trusted hardware has the potential to provide secure computations at minimal performance overhead. However the client has to trust that the hardware is free of errors, bugs, or backdoors. It is difficult to confirm that this is indeed the case, since errors can be introduced in both the design of the hardware and in the fabrication process, which is frequently outsourced [31]. In fact, hardware backdoors have been found in real-world military-grade hardware chips [42], and hardware trojan detection is an active research field in the hardware community [16]. We believe that it is useful to develop alternatives that rely only on cryptographic primitives.

Frequency attacks on property-preserving encryption. Property-preserving encryption schemes by definition leak a particular property of the encrypted data. For example, deterministic encryption [13] leaks whether two ciphertexts are equal, and order-preserving encryption [17] leaks the order between the ciphertexts. Naveed et al. [36] used auxiliary information and frequency analysis to show that one can infer the plain text from ciphertexts that have been encrypted using such property-preserving encryption schemes.

8 Conclusion

We have described Seabed, a system for performing Big Data Analytics over Encrypted Data. We have introduced two novel encryption schemes: ASHE for fast aggregations over encrypted data, and SPLASHE to protect against frequency attacks. Our evaluation on real-world datasets shows that ASHE is about an order of magnitude faster than existing techniques, and that its overhead compared to a plaintext system is within 45%.

9 Acknowledgments

We would like to thank Sriram Rajamani, Kapil Vaswani and Chandu Thekkath for helpful suggestions on early versions of this manuscript. We would also like to thank the OSDI reviewers for their constructive feedback and Bryan Ford for shepherding this paper. This work was supported in part by NSF grants CNS-1513694 and CNS-1054229.

References

[1] AmpLab Big Data Benchmark. `https://amplab.cs.berkeley.edu/benchmark/`.

[2] Apache Spark. `http://spark.apache.org/`.

[3] Google Protocol Buffers. `https://developers.google.com/protocol-buffers/`.

[4] PowerBI. `https://powerbi.microsoft.com/en-us/features/`.

[5] RFC - DEFLATE Compressed Data Format Specification. `https://tools.ietf.org/html/rfc1951`.

[6] Tableau Online. `http://www.tableau.com/products/cloud-bi`.

[7] Watson Analytics. `http://www.ibm.com/analytics/watson-analytics/us-en/`.

[8] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order-preserving encryption for numeric data. In *Proc. ICMD*, 2004.

[9] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *Proc. of CIDR*, 2013.

[10] ARM. ARM Security Technology Building a Secure System using TrustZone Technology. ARM Technical White Paper, 2009.

[11] S. Bajaj and R. Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering*, 26(3):752–765, 2014.

[12] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In *Proc. CRYPTO 2007*.

[13] M. Bellare, A. Boldyreva, and A. ONeill. Deterministic and efficiently searchable encryption. In *Proc. CRYPTO, 2007*.

[14] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proc. CCS '93*.

[15] R. Bhagwan, R. Kumar, R. Ramjee, G. Varghese, S. Mohapatra, H. Manoharan, and P. Shah. Adtributor: Revenue debugging in advertising systems. In *Proc. USENIX NSDI*, 2014.

[16] S. Bhasin and F. Regazzoni. A survey on hardware trojan detection techniques. In *Proc. IEEE ISCAS*, 2015.

[17] A. Boldyreva, N. Chenette, Y. Lee, and A. Oneill. Order-preserving symmetric encryption. In *Proc. EUROCRYPT, 2009*.

[18] D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-dnf formulas on ciphertexts. In *Theory of cryptography*, pages 325–341. Springer, 2005.

[19] D. Boneh, K. Lewi, M. Raykova, A. Sahai, M. Zhandry, and J. Zimmerman. Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. In *Proc. EUROCRYPT*, 2015.

[20] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 2015.

[21] N. Chenette, K. Lewi, S. A. Weis, and D. J. Wu. Practical order-revealing encryption with limited leakage. In *Proc. FSE, 2016*.

[22] V. Costan and S. Devadas. Intel sgx explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 2016. `http://eprint.iacr.org`.

[23] S. Davenport and R. Ford. Sgx: the good, the bad and the downright ugly. *Virus Bulletin*, 2014.

[24] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *Proc. USENIX Security, 2015*.

[25] T. Ge and S. Zdonik. Answering aggregation queries in a secure system model. In *Proc. VLDB, 2007*.

[26] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. ACM STOC*, 2009.

[27] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. In *Proc. CRYPTO, 2012*.

[28] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES Circuit (Updated Implementation), 2015. `https://eprint.iacr.org/2012/099.pdf`.

[29] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, 1996.

[30] IBM Corporation. IBM Systems cryptographic hardware products. `http://www-03.ibm.com/security/cryptocards/`.

[31] F. Imeson, A. Emtenan, S. Garg, and M. Tripunitara. Securing computer hardware using 3d integrated circuit (ic) technology and split manufacturing for obfuscation. In *Proc. USENIX Security*, 2013.

[32] K. Kambatla, G. Kollias, V. Kumar, and A. Grama. Trends in big data analytics. *Journal of Parallel and Distributed Computing*, 74(7):2561–2573, 2014.

[33] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy. The unified logging infrastructure for data analytics at twitter. In *Proc. VLDB, 2012*.

[34] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[35] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[36] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proc. ACM CCS*, 2015.

[37] R. Ostrovsky. Efficient computation on oblivious rams. In *Proc. ACM STOC*, 1990.

[38] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. EUROCRYPT, 1999.*

[39] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proc. ACM SOSP*, 2011.

[40] A. Raghunathan, G. Segev, and S. P. Vadhan. Deterministic public-key encryption for adaptively chosen plaintext distributions. In *Proc. EUROCRYPT 2013.*

[41] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proc. IEEE Security and Privacy, 2015.*

[42] S. Skorobogatov and C. Woods. Breakthrough silicon scanning discovers backdoor in military chip. In *Proc. International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2012.

[43] R. Sumbaly, J. Kreps, and S. Shah. The big data ecosystem at LinkedIn. In *Proc. ACM ICMD*, 2013.

[44] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proc. VLDB*, 2013.

A Encryption Schemes

A.1 Additive Symmetric Homomorphic Encryption (ASHE)

ASHE is a symmetric encryption scheme that is additively homomorphic. The scheme works over any additive group (for example, $\mathbb{Z}_n$, the integers mod n, for a positive integer n). The plaintext space is $\mathbb{Z}_n$. We will make use of a pseudorandom function (PRF) F_k drawn from a function family $F : \{0,1\}^\ell \times \{0,1\}^t \to \mathbb{Z}_n$. F_k is a keyed function (with key $k \in \{0,1\}^\ell$). It maps strings of length t to elements in $\mathbb{Z}_n$. The security of PRFs guarantee that, given oracle access to F_k, no probabilistic polynomial time (PPT) adversary can distinguish the outputs of F_k from a truly random function (from $\{0,1\}^t \to \mathbb{Z}_n$).

A.1.1 Scheme Description

The encryption scheme is stateful and must pick a unique identifier $\mathsf{id} \in \{0,1\}^t$ for every ciphertext created[1]. We now describe the three algorithms: $\mathsf{KeyGen}, \mathsf{Enc}$, and Dec associated with ASHE. λ is the security parameter.

- $\mathsf{KeyGen}(1^\lambda)$: This generates a symmetric key k for the encryption scheme, which is subsequently used as a key to the PRF.
- $\mathsf{Enc}_k(m)$: To encrypt $m \in \mathbb{Z}_n$, pick a unique $\mathsf{id} \in \{0,1\}^t$. Set the ciphertext $c = (\mathsf{id}, m - F_k(\mathsf{id}) + F_k(\mathsf{id} - 1))$, where $+$ and $-$ denote group addition and subtraction, respectively.
- $\mathsf{Dec}_k(c)$: To decrypt $c = (\mathsf{id}, v)$, output $m = v + F_k(\mathsf{id}) - F_k(\mathsf{id} - 1)$.

We first observe that the scheme is additively homomorphic (where the ciphertext length grows with the number of additive operations performed). If $c_1 = (\mathsf{id}_1, m_1 - F_k(\mathsf{id}_1) + F_k(\mathsf{id}_1 - 1))$ and $c_2 = (\mathsf{id}_2, m_2 - F_k(\mathsf{id}_2) + F_k(\mathsf{id}_2 - 1))$, then $c_1 + c_2 = (\mathsf{id}_1, \mathsf{id}_2, m_1 + m_2 - F_k(\mathsf{id}_1) - F_k(\mathsf{id}_2) + F_k(\mathsf{id}_1 - 1) + F_k(\mathsf{id}_2 - 1))$. An important observation is that if we add w ciphertexts $c_1, \cdots, c_w$ such that c_i encrypts m_i with identifier id_i and $\mathsf{id}_{i+1} = \mathsf{id}_i + 1$, for all $1 \leq i \leq w - 1$, then the resultant ciphertext has the form: $(\mathsf{id}_1, \cdots, \mathsf{id}_w, \Sigma_{i=1}^{w} m_i - F_k(\mathsf{id}_w) + F_k(\mathsf{id}_1 - 1))$. In this special case, the ciphertext size does not grow with w, the number of additive operations performed, as the list $\mathsf{id}_1, \cdots, \mathsf{id}_w$ can be represented compactly by storing only id_1 and id_w.

A.1.2 Proof of Security

Lemma 1. *ASHE is a semantically secure encryption scheme.*

Proof. Define F' to be a new function as follows: $F'_k(\mathsf{id}) = F_k(\mathsf{id}) - F_k(\mathsf{id} - 1)$. We will first prove that F' is a secure pseudorandom function.

Claim 1. *If F is a pseudorandom function, then F' is also a pseudorandom function.*

Proof. Given a PPT adversary $\mathcal{A}$ that can distinguish the outputs of F' from random, we will construct an adversary $\mathcal{B}$ that can distinguish the outputs of F from random. $\mathcal{B}$ plays the role of the challenger in the security

[1]We can easily make the scheme stateless by picking id randomly from a large enough space, which will ensure that the value will be unique except with negligible probability.

game against adversary $\mathcal{A}$ that breaks the security of the pseudorandom function F'. $\mathcal{B}$ takes part in a security game to break the security of the pseudorandom function F by interacting with a challenger $\mathcal{C}$.

For every query id made by the adversary $\mathcal{A}$, $\mathcal{B}$ responds as follows: it queries the challenger $\mathcal{C}$ with (id) and gets as output R_{id} where R_{id} is chosen by the challenger $\mathcal{C}$ either uniformly at random or as the output of the pseudorandom function $F_k(\mathsf{id})$ for some key k. Then, $\mathcal{A}$ queries the challenger with $(\mathsf{id} - 1)$ and gets as output $R_{(\mathsf{id}-1)}$ where again $R_{(\mathsf{id}-1)}$ is either equal to $F_k(\mathsf{id}-1)$ or chosen uniformly at random. Now, $\mathcal{A}$ sends $(R_{id} - R_{(\mathsf{id}-1)})$ as output to $\mathcal{B}$. Note that if the same query is asked again, $\mathcal{C}$ (and subsequently $\mathcal{B}$) give the same response. Observe that if the response of $\mathcal{C}$ was the output of the pseudorandom function F with key k, then the response of $\mathcal{B}$ to $\mathcal{A}$ is the output of the function F' with the same key k. On the other hand, if the response of $\mathcal{C}$ is chosen uniformly at random, then the response of $\mathcal{B}$ is also chosen uniformly at random. Notice that for every id, we rely only on the fact that $F_k(\mathsf{id})$ is pseudorandom to argue that $F'_k(\mathsf{id})$ is pseudorandom. That is, the pseudorandomness of $F_k(\mathsf{id} - 1)$ is not used to argue that $F'_k(\mathsf{id})$ is random. Analogously, the pseudorandomness of $F_k(\mathsf{id})$ is used only once - to argue that $F'_k(\mathsf{id})$ is pseudorandom and not to argue about the pseudorandomness of $F'_k(\mathsf{id} + 1)$ as well.

Therefore, if $\mathcal{A}$ breaks the security of the pseudorandom function F' by correctly guessing whether $\mathcal{B}$ responds with the output of F' or random with probability $1/2 + \epsilon$ (where ϵ is non-negligible), then $\mathcal{B}$ makes the same guess to the challenger $\mathcal{C}$ and breaks the security of the pseudorandom function F with probability $1/2 + \epsilon$. □

Given Claim 1, the proof of Lemma 1 follows in a straightforward manner from the fact that for every ciphertext, we choose a unique identifier id. That is, given a PPT adversary $\mathcal{A}$ that can break the semantic security of the encryption scheme, we will construct a PPT adversary $\mathcal{B}$ that can distinguish the outputs of F' from random (thereby contradicting Claim 1). $\mathcal{B}$ plays the role of the challenger in the security game against adversary $\mathcal{A}$ that breaks the semantic security of the encryption scheme. $\mathcal{B}$ takes part in a security game to break the security of the pseudorandom function F' by interacting with a challenger $\mathcal{C}$.

For every message m queried by the adversary $\mathcal{A}$, $\mathcal{B}$ chooses an identifier id at random (this identifier can also be chosen by adversary $\mathcal{A}$ as long as it is uniquely chosen for every ciphertext). Then, it queries the challenger with id and gets as a response R_{id} where R_{id} is chosen by the challenger $\mathcal{C}$ either uniformly at random or as the output of the pseudorandom function $F'_k(\mathsf{id})$ for some key k. $\mathcal{B}$ then sends the pair $(\mathsf{id}, m - R_{\mathsf{id}})$ to $\mathcal{A}$ as the ciphertext for message m. Finally, $\mathcal{A}$ sends two messages m_0 and m_1 not queried earlier. Now, $\mathcal{B}$ has to send back a ciphertext for one of them. $\mathcal{B}$ chooses an identifier id^* at random and queries the challenger with id^*. It gets a response R_{id^*}. $\mathcal{B}$ then chooses a bit $b \in \{0, 1\}$ uniformly at random and sends the pair $(\mathsf{id}^*, m_b - R_{\mathsf{id}^*})$ to $\mathcal{A}$. Now, $\mathcal{A}$ has to output a bit b' indicating that the ciphertext encrypts m'_b. If $\mathcal{A}$ outputs $b' = b$, $\mathcal{B}$ tells the challenger $\mathcal{C}$ that his responses are pseudorandom and if $\mathcal{A}$ outputs $b' \neq b$, $\mathcal{B}$ tells the challenger $\mathcal{C}$ that his responses are random. Suppose $\mathcal{A}$ guesses $b' = b$ correctly. That is, $\mathcal{A}$ breaks the semantic security of the encryption scheme with probability $(1/2 + \epsilon)$ where ϵ is non-negligible. Note that if the response of the challenger R_{id^*} was chosen uniformly at random, then no matter how powerful $\mathcal{A}$ is, he cannot guess whether $b = 0$ or $b = 1$ with probability better than $1/2$. Therefore, if the response of $\mathcal{C}$ is random, then the probability that $\mathcal{A}$ guesses $b' = b$ is $1/2$. However, if R_{id^*} is the output of the pseudorandom function F', then $\mathcal{A}$ can guess $b' = b$ correctly with probability $1/2 + \epsilon$. The different events are shown below:

	$b' = b$	$b' \neq b$
$R_{\mathsf{id}^*} \in R$	$1/2 + \epsilon$	$1/2 - \epsilon$
$R_{\mathsf{id}^*} \in PRF$	$1/2$	$1/2$

Here, $R_{\mathsf{id}^*} \in R$ means that R_{id^*} is random while $R_{\mathsf{id}^*} \in PRF$ means that it is the output of the pseudorandom function F'. Since the output of the challenger is either the output of F' or is uniformly random each with probability $1/2$, we can observe that the adversary $\mathcal{B}$ breaks the security of the pseudorandom function with probability $((1/2 * 1/2) + 1/2 * (1/2 + \epsilon)) = (1/2 + \epsilon/2)$ (where $\epsilon/2$ is non-negligible if ϵ is non-negligible). Thus, the encryption scheme is semantically secure. □

A.2 SPLayed Additive Symmetric Homomorphic Encryption (SPLASHE)

Databases, whose columns are encrypted with deterministic encryption are subject to frequency attacks, as illustrated by Naveed *et al.* [36]. In more detail, let $\mathcal{D}$ be a dimension in the database - for example, $\mathcal{D}$ could be a column such as gender, country, or zip code. Let $\mathcal{M}$ be a measure in the database - for example, $\mathcal{M}$ could be revenue, salary, or number of customers. Now, suppose we are likely to make queries of the form "`select SUM(revenue) where gender = Male`". One way to allow such queries to be made over encrypted data, is to encrypt the dimension `gender` using a deterministic encryption (this would map all encryptions of `male` to the same ciphertext and all encryptions of `female` to the same ciphertext), and encrypt the measure `revenue` using an additively homomorphic encryption scheme (such as ASHE described in the previous section). Then, one can filter the rows in the database by the ciphertext corresponding to `male` and then use the additively homomorphic property of the encryption scheme to compute an encrypted copy of `SUM(revenue)`. While this is a meaningful method of achieving this query, it leaks information about the dimension column, since deterministic encryption is used. For example, simply by looking at the database, a malicious attacker can tell the ratio of entries belonging to the two values of the `gender` categorical attribute. In many cases, it is easy to map the ciphertext exactly to either male or female, as the distribution of males and females is quite often known.

A.2.1 Scheme Description

SPLASHE is a specialized encryption method that is designed to defeat frequency attacks on dimensions in databases. Our method leaks no information on frequency counts of dimension values and achieves full semantic security. SPLASH comes in two flavors: basic SPLASHE is efficient when the number of distinct values the dimension can take is low, whereas enhanced SPLASHE is efficient when the number of distinct frequently occurring values is low. We first describe the basic version of SPLASHE and then show the enhanced version.

Basic SPLASHE. The idea is as follows: we will expand both the dimension and measure columns by a factor of d, where d denotes the number of unique values that the dimension can take. Denote the expanded columns by $\mathcal{D}_1, \cdots, \mathcal{D}_d$ and $\mathcal{M}_1, \cdots, \mathcal{M}_d$. In our example above, the dimension (gender) could take $d = 2$ values and hence we expand the gender and revenue columns to get $\mathcal{D}_1 = \mathsf{Gender}_{\mathrm{male}}, \mathcal{D}_2 = \mathsf{Gender}_{\mathrm{female}}$ and $\mathcal{M}_1 = \mathsf{Revenue}_{\mathrm{male}}, \mathcal{M}_2 = \mathsf{Revenue}_{\mathrm{female}}$. Now, suppose we want to encrypt the t^{th} row that has `gender = male` and `revenue = 100`, we set $\mathsf{Gender}_{\mathrm{male}}[t] = 1, \mathsf{Gender}_{\mathrm{female}}[t] = 0, \mathsf{Revenue}_{\mathrm{male}}[t] = 100, \mathsf{Revenue}_{\mathrm{female}}[t] = 0$, and encrypt all the four columns using an additively homomorphic encryption scheme. Note that we use a semantically secure encryption scheme and hence no frequency counts are revealed here. Now, if the client wishes to execute the query "`select SUM(revenue) where gender = Male`", she can do so by simply executing the query "`select SUM(`$\mathsf{Revenue}_{\mathrm{male}}$`)`", which results in the same value (since the value $\mathsf{Revenue}_{\mathrm{male}} = 0$ for all rows with `gender = female`). Furthermore, if the client wishes to hide the dimension value on which the query was made, she can execute both "`select SUM(`$\mathsf{Revenue}_{\mathrm{male}}$`)`" as well as "`select SUM(`$\mathsf{Revenue}_{\mathrm{female}}$`)`" and decrypting whichever value she is interested in.

Reducing SPLASHE overhead. One concern about basic SPLASHE is that the size of the database can grow significantly, especially when the splayed dimension takes multiple values (e.g., a "country" dimension). We now describe a first attempt to reduce the size blow-up of such a scenario. Suppose we knew a-priori that the most frequently occurring countries in the database were going to be USA and Canada. We now create 4 columns for the dimension country and 3 columns for each measure (here we consider revenue to be one measure). We get $\mathcal{D}_1 = \mathsf{Country}_{\mathrm{USA}}, \mathcal{D}_2 = \mathsf{Country}_{\mathrm{Canada}}, \mathcal{D}_3 = \mathsf{Country}_{\mathrm{others}}, \mathcal{D}_4 = \mathsf{CountryDet}$, and $\mathcal{M}_1 = \mathsf{Revenue}_{\mathrm{USA}}, \mathcal{M}_2 = \mathsf{Revenue}_{\mathrm{Canada}}, \mathcal{M}_3 = \mathsf{Revenue}_{\mathrm{others}}$. For entries with `Country = Canada` and `Country = USA`, we encrypt similar to the method described earlier in Basic SPLASHE (i.e., for the t^{th} row, if `country = USA` and `revenue = 100`, we encrypt $\mathcal{D}_1[t] = 1, \mathcal{D}_2[t] = 0, \mathcal{D}_3[t] = 0, \mathcal{M}_1[t] = 100, \mathcal{M}_2[t] = 0, \mathcal{M}_3[t] = 0$, all using ASHE). For entries which have a country other than USA or Canada, we do the following: we encrypt $\mathcal{D}_1[t] = 0, \mathcal{D}_2[t] = 0, \mathcal{D}_3[t] = 1$ and the entry in the field $\mathcal{D}_4 = \mathsf{CountryDet}$ is deterministically encrypted with the name of the country. Also we encrypt $\mathcal{M}_1[t] = 0, \mathcal{M}_2[t] = 0$, and $\mathcal{M}_3[t] = 100$ (if the revenue of this row was 100).

Note, that at this point, we have not specified what to encrypt in the field $\mathcal{D}_4 = \mathsf{CountryDet}$, when `country` is `USA` or `Canada`. If we place a deterministic encryption of 0 (or some other fixed value), then the adversary can a) Learn whether a particular row's country was USA/Canada or some other country; and b) Launch frequency attacks as in [36] on the more infrequently occurring countries.

Enhanced SPLASHE. In order to overcome the security vulnerabilities of the above scheme, we carefully choose how to encrypt these elements. First, observe that what we need is the following: for every country apart from USA and Canada, we need to deterministically encrypt that country a *fixed* number of times in $\mathcal{D}_4$, with this number being *independent* of the number of times that country actually appeared in the database. This will ensure that, even given a frequency count of every country, the adversary cannot launch any attack using the deterministically encrypted entries of the third column.

To accomplish this, we will use the "dummy" entries in $\mathcal{D}_4$ corresponding to rows that have either USA or

Canada as their country. In these rows we can store deterministic encryptions of the other country values, taking care to choose each encryption in a way that balances the frequency counts. For instance, suppose we had a database of 2500 entries, with `USA` and `Canada` occurring 1000 times each, and the remaining 50 countries occurring fewer than 50 times each. Since we have a total of 50 other countries and column $\mathcal{D}_4$ holds 2500 deterministic encryptions, each of the other countries has to be encrypted 50 times. If `India` occurs, say, 30 times, we need to choose 20 of the rows corresponding to `USA` or `Canada` uniformly at random, and store the deterministic encryption for `India` in their $\mathcal{D}_4$ (i.e., CountryDet) column. One can easily see that the "dummy" entries do not affect the correctness of revenue aggregates: the false `India` entries already have an encryption of 0 in the $\mathsf{Revenue}_{\mathrm{others}}$ field. Therefore, when we sum up the revenue of all rows with `country` set to `India`, we get the correct sum of the real `India` entries. Once the frequency counts of all countries (except USA and Canada) have been equally balanced, the $\mathcal{D}_4$ column of any remaining rows (e.g., if there were 2510 rows in the database) is filled with an encryption of a randomly selected country (again, not USA or Canada).

Of course, the above technique was possible because there were enough rows available for storing "dummy" entries. Interestingly, this condition can always be met; one can appropriately choose the number of columns to splay a dimension to, based on the a-priori estimate of the distribution of values in the column. More concretely, lets consider a threshold parameter t that separates the values of the splayed column into *frequent* and *infrequent*. Let us also assign a separate column to the frequent values – that is, those which occur more than t times. The infrequent (occurring less than t times) are all stored in the "others" column using deterministic encryption. Then, there is always a threshold value t which provides enough "dummy" entries to balance out the distribution of the column.

A.2.2 Proof of Security

First, observe that for a given dimension (let's say country), the only column for which we have to argue security is the CountryDet column. This is because, the entries in the remaining columns are already secure by the security of the underlying additively homomorphic encryption scheme. In order to prove the security of Enhanced SPLASHE, we make a simplifying assumption that the *input/storage order* of the rows of the database are uniformly distributed (essentially, we require that the order is independent of the dimension under consideration). If this is not true of the data itself, then one can achieve this by randomly permuting the rows of the database when storing it (either truly randomly or through the use of a secure pseudorandom permutation).

Quantifying information leakage in Enhanced SPLASHE. We now quantify the information that is already known to the adversary. We assume that the adversary has access to the number of rows in the database - n, and hence this is public information. Now, consider a dimension $\mathcal{D}$ that takes k unique values $v_1, \cdots, v_k$. Let us say each v_i occurs x_i times in the database and that $x_1, \ldots, x_k$ are sorted in non-decreasing order. Let t be a threshold. That is, for each v_i, if $x_i > t$, we create a separate column for v_i. Let us say the first j entries have $x_i > t$. Enhanced SPLASHE will then produce $(j+1)$ columns in total $-$ the first j columns for each v_i with $i \leq j$ and the last column for "other". By design, the last column will only contain deterministic encryptions of values v_i with $i > j$. Therefore, the adversary can count how many distinct deterministic encryptions are in this column, and find out the number of infrequently occurring values $c = (k - j)$. Moreover, the adversary can infer j by counting how many columns are used for the most frequently occurring countries.

Additionally, the adversary can learn the threshold t because of the way it is chosen in the Enhanced SPLASHE algorithm. To see why, consider the condition that should be satisfied by t, so that the scheme is both realizable and secure. Our scheme is designed such that we encrypt all values v_i for $i > j$, a total of t times and then for the remaining rows, we sample a value v_i for $i > j$, uniformly at random. Hence, the number of entries of each infrequently occurring value has to be padded to a total of t entries, and there should be at least as many available entries in the last column. But the number of available entries is equal to the number of rows corresponding to more frequently occurring values. So, we get:

$$\sum_{i=j+1}^{i=k} (t - x_i) \leq \sum_{i=1}^{j} x_i$$

Adding $\sum x_i$ for $j+1 \leq i \leq k$, to both sides, we get

$$\sum_{i=j+1}^{i=k} t \leq \sum_{i=1}^{k} x_i$$

The right hand side is n. The left hand side is $(t \times c)$ where $(c = k - j)$ is the number of countries that don't have a column for themselves. Therefore, the condition is $t \leq \frac{n}{c}$.

To summarize, the adversary gets the following information : the number of rows n, the number of infrequently occurring values c, and the number of frequently occurring values, j. We remark here that we need not leak the precise value of the threshold t to the adversary, but the adversary does know that $t \leq \frac{n}{c}$.

Security Definition. The definition of security which Enhanced SPLASHE satisfies follows the simulation-based security framework. Informally, we will require the real distribution of the encrypted database to be indistinguishable, to any probabilistically polynomial time (PPT) adversary, from the ideal distribution of a simulated encrypted database, created by a simulator that knows *only* n, c and j. This will prove that any adversary can learn only n, c and j from the encrypted database. More formally,

Definition 1. *(Simulation-Based Security) An encryption scheme* E *is said to be secure with respect to the simulation-based security, if, for any database* $\mathcal{D}$ *with* n *rows,* c *infrequently occurring dimensions such that* $t \leq (n/c)$*, and* j *frequently occurring dimensions, the view of any probabilistic polynomial time adversary* $\mathcal{A}$ *in the real world (where the database is encrypted as described in Enhanced SPLASHE) is computationally indistinguishable from the view in the ideal world where a polynomial time simulator* Sim *produces a (simulated copy) of an encrypted database, given only public information* n, c *and* j*. Notationally,* $\mathsf{E.Enc}(\mathcal{D}) \approx_c \mathsf{Sim}(n, c, j)$*.*

Lemma 2. *Assuming ideal security of the deterministic encryption scheme, Enhanced SPLASHE is secure with respect to the simulation-based security (as in Definition* 1*).*

Proof (Sketch). The strategy for the simulator $\mathsf{Sim}(n, c, j)$ to output a simulated copy of an encrypted database is as follows. The simulator creates j columns $\mathsf{Dimension}_1, \cdots, \mathsf{Dimension}_j$ for the j frequently occurring dimension values and the column $\mathsf{Dimension}_{\text{others}}$. The simulator (probabilistically) encrypts all values in these columns with zeroes. Similarly, for all measure columns $\mathsf{Measure}_1, \cdots, \mathsf{Measure}_j, \mathsf{Measure}_{\text{others}}$, the simulator probabilistically encrypts all values in these columns with zeroes. Now, for the column $\mathsf{DimensionDet}$, note that there are c values that infrequently occur. The simulator picks a key for the deterministic encryption scheme. Call this key k_{S}. Now, for each value i in the range 1 to c, $\mathsf{Sim}(n, c, j)$ does the following: i) Pick $\frac{n}{c}$ empty rows uniformly at random; ii) For each of these $\frac{n}{c}$ rows, deterministically encrypt the value i (with key k_{S}) and store this in the dimension column. If there are any remaining empty rows, then for each remaining empty row, pick a value $1 \leq i \leq c$ at random and deterministically encrypt this value to store. Queries to the database are handled in a natural manner by the simulator.

In the ideal world, for each infrequently occurring value v of the dimension, the adversary can only see (n/c) rows chosen uniformly at random with the deterministic encryption of some index i in the dimension column. In the real world, for each infrequently occurring value v of the dimension, the adversary can only see (n/c) rows with the deterministic encryption of value v in the dimension column. Based on the assumption about the rows of our database, we know that these (n/c) rows are distributed uniformly at random. Thus, through a reduction to the ideal security of the deterministic encryption scheme [12, 40] (which can be attained in the random oracle model [14]), the view of the adversary in the real world is computationally indistinguishable from his view in the ideal world. □

A.3 Order Preserving/Revealing Encryption

Order Preserving Encryption (OPE) allows the encryption of messages, with the property that given $c_1 = \mathsf{OPEEnc}(m_1)$ and $c_2 = \mathsf{OPEEnc}(m_2)$, $c_1 < c_2$ if and only $m_1 < m_2$ (if $m_1 = m_2$, then $c_1 = c_2$). OPE was first introduced by Agrawal *et al.* [8], and it was cryptographically first defined by Boldyreva *et al.* [17]. In many OPE schemes, it is hard to quantify the information that the adversary can learn about the plaintext, given the ciphertext (and hence security is defined for specific distribution of messages, such as messages chosen uniformly at random). Popa *et al.* [39] presented an OPE scheme with ideal security – given n ciphertexts $c_1, \cdots, c_n$, the only thing that the adversary can learn is the relative ordering of the plaintexts. However, the scheme of Popa *et al.* is stateful and needs to know the set of messages $m_1, \cdots, m_n$ being encrypted *all at once*. This is undesirable for many application and especially in our case while dealing with large volumes of data. The related notion of Order Revealing Encryption (ORE) was introduced by Boneh *et al.* [19]; informally, in an ORE, there exists an algorithm $\mathsf{Compare}$ that takes as input two ciphertexts c_1 and c_2 and outputs which of the underlying plaintexts is greater (this algorithm is simply the comparison function in the case of OPE). Recently, Chenette *et al.* [21] presented a construction of an ORE scheme with precise quantifiable leakage. Moreover, their construction is based only on cryptographic pseudorandom functions (PRFs) and is practical. We use this construction in our system.

The ORE scheme of Chenette *et al.* for the set of n bit messages, has the following leakage function $\mathcal{L}$: $\mathcal{L}(m_1, \ldots, m_k) = \{(m_i < m_j, \mathsf{ind}_{\mathsf{diff}}(m_i, m_j)) : 1 \leq i \leq j \leq t\}$. Here, $\mathsf{ind}_{\mathsf{diff}}(m_i, m_j)$ refers to the index of the most significant bit at which the two messages differ. That is, for every pair of messages, the leakage is which message is larger and which bit do they first differ at. We now briefly describe their scheme here. Fix a security parameter λ. Let $\mathsf{F} : \mathcal{K} \times ([n] \times \{0,1\}^{n-1}) \rightarrow Z_3$ be a secure pseudorandom function (PRF).

- $\mathsf{Setup}(1^\lambda)$: Choose a PRF key $k \in \mathcal{K}$ uniformly at random and set this as the secret key sk.
- $\mathsf{Encrypt}(m, \mathsf{sk})$: Let $b_1 \ldots b_n$ be the binary representation of m. For each $i \in [n]$, compute $u_i = (\mathsf{F}(k, (i, b_1 b_2 \ldots b_{i-1} || 0^{n-i})) + b_i) \bmod 3$, where F is the PRF. The ciphertext ct is $(u_1, \ldots, u_n)$.
- $\mathsf{Compare}(\mathsf{ct}_1, \mathsf{ct}_2)$: Let $\mathsf{ct}_1 = (u_1, \ldots, u_n)$ and $\mathsf{ct}_2 = (u'_1, \ldots, u'_n)$. Find the smallest index i such that $u_i \neq u'_i$. If no such i exists, both messages are equal. If $u_i = (u'_i + 1) \bmod 3$, output that ct_1 encrypts the larger message. Else, output that ct_2 encrypts the larger message.

B Analysis of MDX queries supported by Seabed

As part of our argument that Seabed supports an important range of big data analytics, we analyzed MDX and the Spark API, two widely-used programming interfaces for analytics (section 5). This section details our findings from the analysis of MDX. The analysis reveals that Seabed has four different ways of supporting MDX queries. The reader can find the number of MDX queries that fall into each of these categories in table 4. Below, we describe the four categories.

Support completely on server: Seabed's encryption techniques can support operations such as computing sum, average, count, min, and max with no client-side support. This type of query is denoted by S in Table 6.

Support with client pre-processing: Seabed can also support quadratic computation necessary for more complex analytics such as variance, standard deviation, and covariance. These queries can be supported in Seabed too; the client can precompute certain quadratic terms and upload them in encrypted format using ASHE. This type of query is denoted by CPre in Table 6.

Support with client post-processing: MDX allows users to specify arbitrary user-defined functions. When these functions are complex, Seabed cannot support them directly or with client pre-processing. However, even complex functions can be supported by separating queries into server-side and client-side parts. This is akin to how Monomi supports complex analytical queries. This type of query is denoted by CPost in Table 6.

Support with two client round-trips: Some queries require the client to compute an intermediate result, encrypt it, and send it back to the server. This type of query is denoted by 2R in Table 6.

Table 6 present a more detailed analysis of the individual MDX queries and how Seabed supports these queries.

S. No	Query Type	Description	How Seabed supports it	Seabed Type
1	Aggregate	Aggregates of measures	ASHE for Sum, Count; OPE for Max, Min	S
2	Avg	Average of measures	ASHE for Sum, Count; Client does division	S
3	CalculationCurrentPass	Current calculation pass of cube	Independent of Seabed	S
4	CalculationPassValue	Returns MDX expression value after current pass	Independent of Seabed	S
5	CoalesceEmpty	Updates empty value to numeric/string	Can be done with extra counter with identity	CPre
6	Correlation	Correlation Coefficient of two series X, Y	ASHE & precomputation of XY; Client does division	CPre
7	Count(Dimensions)	Number of dimensions in cube	Independent of Seabed	S
8	Count(Hierarchy Levels)	Number of levels in hierarchy	Independent of Seabed	S
9	Count(Set)	Number of elements in a set	Using DE or SPLASHE	S
10	Count(Tuple)	Number of dimensions in tuple	Independent of Seabed	S
11	Covariance	Covariance of X, Y	Same as Correlation	CPre
12	CovarianceN	Covariance of X, Y (with division by N-1)	Same as Correlation	CPre
13	DistinctCount	Counts distinct elements	Using DE or SPLASHE	S
14	IIf	One of two values based on logical test	Two values sent back to the client	CPost
15	LinRegIntercept	Intercept in the Regression Line using Least Squares Method	Data sent back to client for every iteration	2R
16	LinRegPoint	y in the regression line	Same as LinRegIntercept	2R
17	LinRegR2	Coefficient of Determination	Same as LinRegIntercept	2R
18	LinRegSlope	Slope of the regression line	Same as LinRegIntercept	2R
19	LinRegVariance	Var. associated with regression line	Same as LinRegIntercept	2R
20	LookupCube	MDX expression over a cube	Data sent back to client for computation	CPost
21	Max	Maximum value in set	Using OPE	S
22	Median	Median value in set	Using OPE	S
23	Min	Minimum value in set	Using OPE	S
24	Ordinal	Zero-based ordinal value	Using OPE	S
25	Predict	Value of expression over data mining model	Data sent back to client for computation	CPost
26	Rank	One-based rank of set	Using OPE	S
27	RollupChildren	Value generated by rolling up values of children	Data sent back to client for computation	CPost
28	Stddev	Standard deviation of a set X	ASHE when Client uploads encrypted X^2 terms	CPre
29	StddevP	Std. Dev. using biased pop. formula	Same as Stddev	CPre
30	Stdev	Alias for Stddev	Same as Stddev	CPre
31	StdevP	Alias for StddevP	Same as Stddev	CPre
32	StrToValue	Value of MDX-formatted string	Independent of Seabed	S
33	Sum	Sum over a set	Using ASHE	S
34	Value	Value of a measure as a string	Independent of Seabed	S
35	Var	Variance of a set X	Same as Stddev	CPre
36	Variance	Alias for Var	Same as Stddev	CPre
37	VarianceP	Alias for VarP	Same as Stddev	CPre
38	VarP	Var. using biased pop. formula	Same as Stddev	CPre

Table 6: Number of MDX queries that Seabed supports.