Cumulus4j: A Provably Secure Database Abstraction Layer

Matthias Huber, Matthias Gabel, Marco Schulze, Alexander Bieber

Cumulus4j: A Provably Secure Database Abstraction Layer

Matthias Huber, Matthias Gabel
name.surname@kit.edu
Marco Schulze, Alexander Bieber
surname@nightlabs.de

Karlsruhe Institute of Technology (KIT), NightLabs Consulting GmbH

Abstract. Cloud Computing has huge impact on IT systems. It offers advantages like flexibility and reduced costs. Privacy and security issues, however, remain a major drawback. While data can be secured against external threats using standard techniques, service providers themselves have to be trusted to ensure privacy.
In this paper, we present a novel technique for secure database outsourcing. We provide the security notion Ind-ICP that focuses on hiding relations between values. We prove the security of our approach and present benchmarks showing the practicability of this solution. Furthermore, we developed a plug-in for a persistence layer. Our plug-in de- and encrypts sensitive data transparently to the client application. By using encrypted indices, queries can still be executed efficiently.

1 Introduction

Cloud Computing enables a more efficient use of storage and computing resources by sharing them between multiple applications and clients. Outsourcing and pay-per-use payment models cause reduced cost for IT infrastructures. Also, the risk of fatal data loss is minimized due to specialization of the individual providers.

Huge impediments for the adoption of Cloud Computing, however, are security concerns. A client using a cloud service, loses control over his data. He cannot control whether its data is copied or misused. A malicious system administrator may copy sensitive data and sell it to e.g. competitors. Providers may even be required by law to disclose the data of their clients to government agencies.

Current security measures of Cloud Computing providers focus on *external adversaries*. Authentication and authorization mechanisms prohibit access for unauthorized clients. Measures against *internal adversaries* mostly try to restrict physical access to servers to authorized staff. Examples like Wikileaks, however, show that legal insiders also have to be taken into account.

In this paper, we present a novel approach to secure database outsourcing. Intuitively, our scheme hides associations like $Alice \leftrightarrow Bob$ rather than hiding the values *Alice* and *Bob* themselves. In order to achieve security as well as

efficiency, we use a combination of encryption and index generation. We provide the security notion Ind-ICP that informally hides relations between data values. Furthermore, we prove that our approach fulfills this new security notion.

In order to demonstrate the practicability of our approach, we provide Cumulus4j [1], an implementation of our approach for data objects. We implemented Cumulus4j as a plug-in for DataNucleus [2]. DataNucleus is an open-source database abstraction layer that implements the latest versions of JPA [4] and JDO [3]. Our plug-in transparently encrypts data before it is stored in the database back-end, while providing provable security. The overhead introduced by Cumulus4j depends on the query type and the data structure. We define the overhead as the fraction time with Cumulus4j over time without Cumulus4j. Our benchmarks show that Cumulus4j has no impact on select and update queries but takes 5 times longer for inserts. Cumulus4j is open source under the GNU Affero General Public License, and can be downloaded from [1].

This paper is structured as follows: In Section 2, we will discuss related work. We present the data model used by Cumulus4j in Section 3 and the architecture in Section 4. In Section 5, we prove the security of Cumulus4j. We present and discuss benchmark results in Section 6. Section 7 concludes.

2 Related Work

The problem of secure database outsourcing emerged early in 2002 [13]. Consequently, there is a rich body of research on this problem. Most of the approaches rely either on indices or on fragmentation.

There are also well-known cryptographic schemes, that could be used to implement secure database outsourcing, namely secure multi party computations and fully homomorphic encryption [11, 10]. While providing provable security, the huge overhead of these schemes would cancel out the benefits of outsourcing.

More practical approaches try to find a trade-off between security, performance, and efficient support for as many query types as possible. An efficient and practical database outsourcing scheme should provide support for sub-linear query execution time in the size of the database [16].

To our knowledge, the first practical approach to the problem of secure database outsourcing was presented by Hacigümüs et. al. in [13]. Based on this approach, additional solutions have been suggested [12, 8, 14].

The idea of these solutions is to create indices in order to support efficient query processing. These indices are coarse-grained. They contain keywords, ranges, or other identifiers and row ids of the rows in the database where the keywords occur. An adapter handles query transformation and sorts out false positives. Depending on the indices and the queries this can lead to a large overhead. Furthermore, it is unclear what level of security these approaches provide for realistic scenarios.

Another idea found in literature is to achieve privacy by data partitioning. The approaches in [7, 9, 20] introduce privacy constraints that can be fulfilled by fragmenting or encrypting the data [7, 9] or fragmentation and addition of

dummy data [20]. In [18], Nergiz et. al. use the Anatomy approach [21] to achieve privacy in a database outsourcing scenario. The level of privacy achieved by these partition-based approaches depends on the actual data and user defined privacy constraints. They do not provide any security notion beyond that.

Different to the approaches described above is CryptDB [19]. It does not rely on explicit indices or fragmentation. It uses the interesting concept of onions of encryptions. CryptDB proposes to encrypt the database attribute wise. Furthermore, it uses different encryption schemes that for example allow for exact match or for range queries of encrypted values. These encryptions are nested with the clear text in the middle. If a query cannot be executed because of the current encryption level of an attribute, the database removes the outermost layer of all entries of this attribute. Therefore, the security of this approach depends on how the database is used (i.e. what type of queries are issued to the database). The CryptDB approach does not provide any security notion.

3 Data Model

In order to store data objects securely in a database, we need to transform them. In this section, we describe the data model of Cumulus4j, the transformation and encryption of data objects, as well as retrieval of stored data.

A difference between relational data and objects is that objects can hold references to other objects. In order to serialize objects we serialize these references. We also need to serialize references for the indices we create. We will explain this process in the next subsection.

3.1 Serializing References

In contrast to plain fields such as int, char or boolean, references to other objects cannot be encrypted directly. They need to be serialized first. Therefore, we replace the reference with a field containing the id of the referenced object. Consider Figure 1(a). There are two objects, each with an unique *id*. These ids are provided by DataNucleus as specified in JDO and JPA. The object A has a reference to object B. Cumulus4j replaces this reference with an additional field in object A' containing the object id of object B.

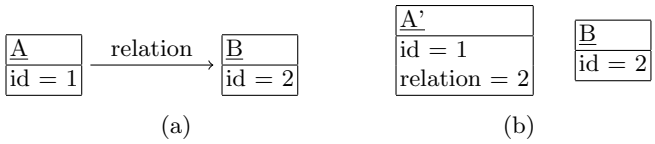


Fig. 1. Two objects A and B (a). Each object has a unique id. Object A has a reference *relation* to object B. In Figure (b), the reference has been replaced. A' contains an additional field with the id of object B.

Encrypting this field prevents an adversary from learning the relation of object A to object B. With the encryption key Cumulus4j can, however, reconstruct the original object A from the object A'.

Note that this procedure can also be recursively applied to more complex data structures.

3.2 The Cumulus4j transformation

Cumulus4j transforms data objects into the data structure depicted in Figure 2. This transformation is done in three steps:

1. **Data object transformation and encryption**
   Cumulus4j serializes data objects, encrypts the serialization, and stores it in wrapper objects of the type *DataEntry*. References to other objects are serialized by replacing them with their wrapper objects' IDs.
2. **Meta data class creation**
   For each data object class a *ClassMeta* object and for each field of each class a *FieldMeta* object is created. These objects contain metadata such as the name of the class/field. Additionally *ClassMeta* objects contain references to their *FieldMeta* objects.
3. **Index generation**
   For each field in a data object, an *IndexEntry* object is created. This object contains the value of the field together with serialized (cf. Section 3.1) and encrypted references to those *DataEntry* wrapper objects containing the indexed field value. There are different types of *IndexEntry* classes. For example an *IndexEntryDouble* objects contains a double value, while an *IndexEntryString* object contains a string value. Each *IndexEntry* object contains a reference to its *FieldMeta* object.

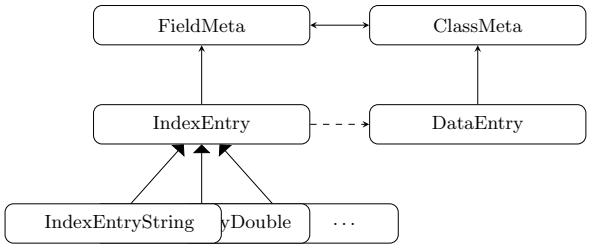


Fig. 2. The Cumulus4j data model stored in the back-end. The dashed arrow represents a serialized and encrypted reference. The *DataEntry* classes contain encrypted serializations of data objects.

After this transformation, due to the encrypted references in the *IndexEntry* objects and the encrypted data objects, an adversary only learns the field

values of the original objects, but not which values occurred in the same object. Cumulus4j, however, can reconstruct the original objects. Note, that this transformation can also be applied to relational data interpreted as data objects (cf. Section 5.1). We use this in our security proof in Section 5.3.

For increased security, the index generation step can be suppressed with annotations in data object classes. In this case no index will be generated. Cumulus4j allows for suppression of index generation on field level. This has implications for security as well as performance. Not generating an index results in faster writes and increased security. This is an option for fields that are confidential by itself, and will not be part of conditions in queries (e.g. credit card numbers).

3.3 Data Retrieval

In the last section, we described the transformation that is used in order to store data objects. In this section, we describe the retrieval of data objects. This is done by the Cumulus4j plug-in that intercepts issued queries and rewrites them. The transformation of data retrieval queries is depicted as pseudocode in Figure 3.

```
Data: retrieval query q with conditions C and boolean operators OP
Result: result set S
exp = {} ;                    /* empty expression of algebra of sets */
foreach c_i ∈ C do
    ind ← retrieve and decrypt IndexEntry objects with matching condition c_i;
    if i = 1 then exp = ind;
    else exp = exp op_{i-1} ind;
end
ids = evaluate exp ;                                   /* set of ids */
data ← retrieve and decrypt DataEntry objects with ids in ids;
S ← q(data) ;                  /* remaining evaluation of q with data */
return S;
```

Fig. 3. Pseudocode for data retrieval queries in Cumulus4j

Consider the following example selection query for objects of a class C: $\sigma_{A=x\ op\ B<y}(C)$, where A and B are attributes, x and y are attribute values from the domain of A and B, respectively, and op is a boolean operator. In order to execute this query, Cumulus4j issues queries to the IndexEntries for indices of Class C and field A with value x and for field B with value $< y$. This procedure can be parallelized. Then Cumulus4j decrypts the indices and applies the boolean operator op to the decrypted sets of indices. The result is the set of indices of DataEntries that satisfies the query. After retrieval and decryption of these DataEntries, Cumulus4j returns them. If the issued query contains an aggregation (e.g. count), we use in memory evaluation before returning the result.

4 Architecture

We implemented Cumulus4j as a plug-in for the database abstraction layer DataNucleus. Cumulus4j integrates into DataNucleus as an OSGi plug-in. Consider the architecture depicted in Figure 4. The application component accesses DataNucleus and DataNucleus accesses the underlying database. Cumulus4j enhances DataNucleus.

The integration of our plug-in, and therefore the encryption of the underlying database, is fully transparent to the application (except for a minimum of Cumulus4j API calls controlling the key management). Thus, existing applications using DataNucleus can be easily enhanced with Cumulus4j.

In one database, thousands or more different keys are used to encrypt different records. During a database operation, only those encryption keys are in the memory of the server Cumulus4j is deployed to which are needed for the current operation. Therefore, an attacker taking a memory dump can only compromise a subset of the keys and thus decrypt only a part of the database.

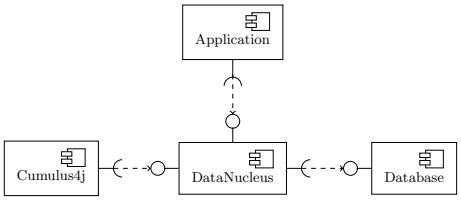


Fig. 4. Coarse architecture of DataNucleus with Cumulus4j. The integration of Cumulus4j as a plug-in for DataNucleus is transparent for the application.

Cumulus4j requires a key server that manages encryption keys. Cumulus4j supports deployment of the key server on the client as well as deployment of the key server on a separate server.

Key Server deployed on the Client In order to communicate with the server, the client establishes a communication channel with the server. Without a dedicated key server, the client will initiate a second communication channel. This channel is dedicated for key requests issued by the server and responses from the client (cf. Figure 5(a)). Key requests are issued by the server whenever it has to encrypt or decrypt data and therefore needs access to certain keys.

Dedicated Key Server Alternatively to holding the keys on every client, it is possible to run a dedicated key-server (cf. Figure 5(b)). Then, clients need to establish crypto sessions with the key server. Since the keys are password protected, the key server only sends issued keys to the server if there is a corresponding crypto session with a client.

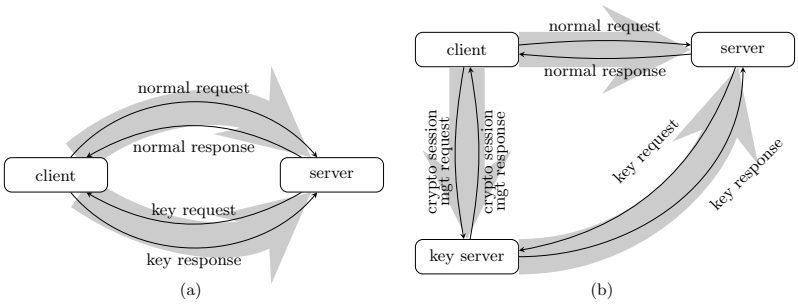


Fig. 5. Key management for different deployment Scenarios: Without a dedicated key server (a) clients handle key management. With a dedicated key-server (b) clients establish a crypto session with the key server.

To increase security, the presence of a crypto-session (i.e. an authenticated user) is not sufficient for the key server to respond to key requests. Only, if the client is currently expecting the application server to perform database operations on his behalf, access to the key server is granted. This is done by temporarily unlocking the current crypto-session. It is locked again immediately after the application server finished its database operation. Therefore, a user does not need to log out in order to deny access to the keys on his behalf. His crypto-session is always locked whenever the user is not actively using the application.

5 Security of Cumulus4j

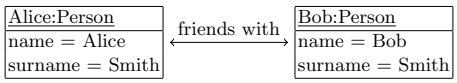
In this section, we introduce a novel security notion for outsourced databases, called Ind-ICP (Indistinguishability under Independent Column Permutation). Informally, this notion hides relations between attribute values. Furthermore, we argue that the Cumulus4j transformation we presented in Section 3.2 adheres this security notion. In order to do so, we interpret objects as rows in a table with a scheme defined by the class of the object.

5.1 Interpretation of Objects as Relational Data

Data objects contain either primitive values (e.g. int, boolean, char) or references to other objects. In order to prove the security of Cumulus4j, we interpret objects as rows in tables of a relational database. The representation we choose is as follows: Tables represent classes, attributes represent fields of classes, and attribute values represent field values of objects. Therefore, an object is represented by a row in the table of its class. Consider for example Figures 6(a) and 6(b).

Figure 6(a) depicts two objects of the class Person. These objects have the field *name*, *surname* and a *friends with* reference to each other.

These objects can be represented by rows in the table depicted in figure 6(b). Each row has a unique *id*. The references are replaced with the *id*s of the objects or rather rows.



(a)

id	name	surname	friends with
1	Alice	Smith	2
2	Bob	Smith	1

(b)

Fig. 6. An example of two objects of type *Person* (a) and a tabular representation (b) of these objects.

Note, that the mapping of data objects to relational data is bijective. Therefore, a proof of security properties of objects represented this way, also applies to the objects itself.

5.2 A Security Notion for Outsourced Databases: Ind-ICP

In this section, we will present Ind-ICP, a security notion applicable to secure database outsourcing. We already introduced Ind-ICP in [15]. In this paper, we will do this more formally. We do not distinguish between a table and a database, since a (finite) set of tables always can be normalized to a universal relation. For the definition of our security notion, we define a database as a multiset of tuples:

Definition 1. *A* database $d = \{t_1, t_2, \ldots, t_n\}$ *is a multiset of tuples with attributes* $A_1, A_2, \ldots, A_m$*. We denote the value of attribute* A_i *in tuple* t_j *as* $d_{i,j}$*. We call the set of all* i*-th elements of each tuple in* d *the* i*-th column of* d*. We call the set of all databases* DB*.*

Throughout this paper, we use the terms tuple and row interchangeably. When interpreting objects as rows of a database, the Cumulus4j transformation presented in Section 3.2 takes databases as input and outputs also databases. We define this process as a database transformation:

Definition 2. *A* database transformation *is a transformation* $f : DB \to DB$*. We call* $\mathcal{F}$ *the set of all database transformations. If* f *is a function, we call* f *a* database function*.*

Note that not all database transformations are functions. Consider for example transformations that involve probabilistic processes. In order to define a security notion for database outsourcing, we define a relation for databases that should be indistinguishable after the Cumulus4j transformation. This relation is implied by the following database function:

Definition 3. *Let* $\Pi \subset \mathcal{F}$ *be the set of database functions* $\mathfrak{p} : DB \to DB$ *such that each* $\mathfrak{p} \in \Pi$ *independently permutes the entries within each column of a database. We call* $\mathfrak{p}$ *an* independent column permutation *(ICP).*

Consider for example the databases depicted in Figure 7. The right database (b) is the result of applying the independent column permutation $\mathfrak{p} = (\mathfrak{p}_1, \mathfrak{p}_2, \mathfrak{p}_3)$ with $\mathfrak{p}_1 = (13)$, $\mathfrak{p}_2 = (123)$, and $\mathfrak{p}_3 = id$ to the one (a).

Based on Definition 3, we define the following experiment:

Definition 4. *Let $d \in DB$ be a database, $f \in \mathcal{F}$ a database transformation, $\mathfrak{p} \in \Pi$ a independent column permutation, $\mathcal{A}$ an adversary and $i \in \{0,1\}$. We define the experiment Ind-ICP$^i_{\mathcal{A}}$ as follows:*

Ind-ICP$^i_{\mathcal{A}}(d)$:
$d_0 \leftarrow f(d)$
$d_1 \leftarrow f(\mathfrak{p}(d))$
$b \leftarrow \mathcal{A}(d_i)$
return b

In this experiment, an adversary $\mathcal{A}$ guesses whether an ICP $\mathfrak{p}$ has been applied to a database d before applying f. Now, we can define our security notion:

Definition 5. *For a database transformation f,* Indistinguishability under Independent Column Permutation *(Ind-ICP) holds iff for each polynomially restricted adversary $\mathcal{A}$, for each database $d \in DB$, and each ICP $\mathfrak{p} \in \Pi$ the following holds:*

$$\mathrm{Adv}^{Ind\text{-}ICP}_{\mathcal{A}}(d) := \left| Pr[Ind\text{-}ICP^1_{\mathcal{A}}(d) = 1] - Pr[Ind\text{-}ICP^0_{\mathcal{A}}(d) = 1] \right|$$

is negligible.

Informally, a database transformation fulfills Ind-ICP if, given the transformed database, an adversary cannot infer the relations of the attribute values of the original database, since the relations are destroyed by independent column permutations. This security notion is weaker than classical cryptographic security notion (e.g. Ind-CCA). It, however, allows for schemes that support efficient execution of queries on the *encrypted* data.

Please note, that Ind-ICP does not compose with background knowledge. According to the No-Free-Lunch Theorem [17], this is not possible for an scheme that maintains some utility. Because of this, in our implementation we introduced annotations that allow for suppression of index generation.

5.3 Cumulus4j fulfills Ind-ICP

In Section 5.1, we argued that data objects also can be interpreted as relational data. In this section, we apply the Cumulus4j transformation to arbitrary relational data and prove that it adheres Ind-ICP. For this proof, we assume that

α	a	$\mathfrak{a}$
β	b	$\mathfrak{b}$
γ	c	$\mathfrak{c}$

(a)

γ	c	$\mathfrak{a}$
β	a	$\mathfrak{b}$
α	b	$\mathfrak{c}$

(b)

Fig. 7. An example for a database before (a) and after (b) an independent column permutation.

all tables are sorted lexicographically and that the adversary has no background knowledge about the database.

Consider Figure 8: Figure 8(a) depicts an arbitrary table with attributes a_i and attribute values $\mathrm{v}_{i,j}$. Figure 8(b) depicts the table from Figure 8(a) after applying an arbitrary independent column permutation (cf. Section 5.2). We now apply the Cumulus4j transformation to these two tables and show, that the results are indistinguishable for any polynomial time adversary. Thus, proving that the Cumulus4j transformation fulfills Ind-ICP.

a_1	a_2	...
$\mathrm{v}_{1,1}$	$\mathrm{v}_{2,1}$	...
$\mathrm{v}_{1,2}$	$\mathrm{v}_{2,2}$	...
...	...	...

(a)

a_1	a_2	...
$\mathrm{v}_{1,\pi_1(1)}$	$\mathrm{v}_{2,\pi_2(1)}$	...
$\mathrm{v}_{1,\pi_1(2)}$	$\mathrm{v}_{2,\pi_2(2)}$	...
...	...	...

(b)

Fig. 8. A table with attributes a_i and values $\mathrm{v}_{i,j}$ (a) and the same table after applying an independent column permutation (b). Not depicted: The meta data associated with the tables and its attributes.

Interpreting the tables in Figure 8 as objects, applying the Cumulus4j transformation (cf. Section 3.2) and reinterpreting the result as relational data yields tables depicted in Figures 9 and 10. For the sake of simplicity and w.l.o.g., we assume each attribute value to be unique.

fID	**cID**	**metadata**
1	1	$\mathrm{fm}_{1,1}$
2	1	$\mathrm{fm}_{1,2}$
...	...	...

(a) FieldMeta

cID	**metadata**
1	cm_1

(b) ClassMeta

key	**fID**	**IDlist**
$\mathrm{v}_{1,1}$	1	E(1)
$\mathrm{v}_{1,2}$	2	E(2)
$\mathrm{v}_{2,1}$	1	E(1)
$\mathrm{v}_{2,2}$	2	E(2)
...	...	...

(c) IndexEntry

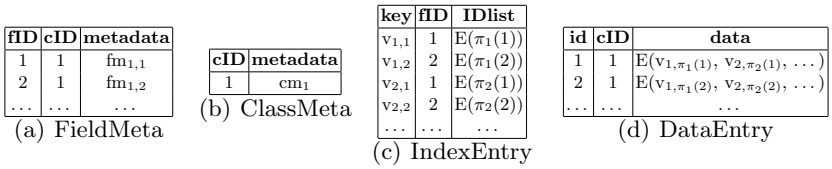
id	**cID**	**data**
1	1	E($\mathrm{v}_{1,1}$, $\mathrm{v}_{2,1}$, ...)
2	1	E($\mathrm{v}_{1,2}$, $\mathrm{v}_{2,2}$, ...)
...	...	...

(d) DataEntry

Fig. 9. A relational representation of the result of the application of the Cumulus4j transformation (cf. Section 3.2) to the table in Figure 8(a) assuming the attribute values $\mathrm{v}_{i,j}$ are different.

Cumulus4j encrypts each row of the original table (*DataEntry*), generates indices (*IndexEntry*) with encrypted keys for every attribute value, and generates the meta data tables. The meta data tables contain keys, meta data about the original table, as well as meta data about attributes.

The tables in Figures 9 and 10 are identical except for the encrypted entries. Therefore, breaking the Ind-ICP property of Cumulus4j implies breaking the underlying encryption scheme.



fID	cID	metadata
1	1	$fm_{1,1}$
2	1	$fm_{1,2}$
...	...	...

(a) FieldMeta

cID	metadata
1	cm_1

(b) ClassMeta

key	fID	IDlist
$v_{1,1}$	1	$E(\pi_1(1))$
$v_{1,2}$	2	$E(\pi_1(2))$
$v_{2,1}$	1	$E(\pi_2(1))$
$v_{2,2}$	2	$E(\pi_2(2))$
...	...	...

(c) IndexEntry

id	cID	data
1	1	$E(v_{1,\pi_1(1)}, v_{2,\pi_2(1)}, \dots)$
2	1	$E(v_{1,\pi_1(2)}, v_{2,\pi_2(2)}, \dots)$
...	...	...

(d) DataEntry

Fig. 10. The result of the application of the Cumulus4j transformation to the table in Figure 8(b) assuming the attribute values $v_{i,j}$ are different.

6 Performance

In order to predict the performance of an application after migrating to Cumulus4j, we measure the performance with a generic benchmark suite. For this, we use the open-source database benchmark suite PolePosition [5], which can be easily used with DataNucleus. We compare two engines: DataNucleus without Cumulus4j and DataNucleus with Cumulus4j. The benchmark is run multiple times for each engine and averaged. This way, effects of the OS and the Java garbage collection are minimized. Different scenarios (called *circuits*) are considered in order to better understand the impact on a complex application later. The full source code – including the benchmarks – can be downloaded at [1]. We measure the time for four different circuits (for details cf. [6]):

- **Complex** uses a deep object graph of different classes with an inheritance hierarchy of five levels.
- **Flatobject** uses simple flat objects with indexed fields.
- **Inheritancehierarchy** operates on objects of a class hierarchy with a depth of five levels.
- **Nestetdlists** uses a deep graph of lists for traversing.

We describe the operations performed in the different circuits:

- **write** stores all objects into an initially empty database.
- **read** loads all attached objects into memory and traverses them, by calculating a checksum over all objects.
- **query** queries for instances over an indexed field.
- **update** traverses all objects, updates a field in each object.
- **delete** traverses all objects and deletes each object individually.
- **queryIndexedString** simulates querying for a number of flat objects by an indexed string member.
- **queryIndexedInt** simulates querying for a number of flat objects by an indexed int member.

DataNucleus translates all above queries into rapidly processable exact match queries (`attr = param`).

The setup is as follows: The benchmark is run on an application and benchmark server (Intel Core i7-2700K, 3.5 GHz, 16 GB RAM). A database server

(AMD Phenom II X6, 0.8 GHz, 8 GB RAM) hosts the MySQL 5.5 database. They are connected via Gigabit Ethernet. The setup divides the servers into two zones: The trusted zone running the DataNucleus+Cumulus4j middleware and the untrusted storage back-end.

The results are shown in Figure 11 in appendix A. In oder to cope with outliers, the mean time for all operations is shown on a logarithmic scale in order to cope with outliers. The introduction of the Cumulus4j plug-in makes the creation of the initial datasets more complex, thus consuming more time. Data encryption and the creation of indices result in an overhead of factor 5. MySQL does not use any indices per default. But due to the implicit use of indices in Cumulus4j, query operations run faster on the Cumulus4j engine for all circuits. Updates take almost the same amount of time while deletes are marginally slower with Cumulus4j. It is currently unknown why the read operations in the *Inheritancehierarchy* are much slower than without Cumulus4j.

We can confer that the use of the Cumulus4j plug-in in DataNucleus does not introduce high overhead in any scenario. It is best suited for applications with many read and few write queries, but it can be used in every scenario.

When using Cumulus4j in web application, the user would probably not notice any slowdown at all, because the overall response time is only marginally affected by Cumulus4j.

7 Conclusion and future work

In this paper, we introduced a new approach for securing outsourced databases. While taking insider-attacks into account, we focused on privacy – a major concern when dealing with Cloud Computing.

We presented Cumulus4j, a plug-in for the database abstraction layer DataNucleus. It serializes and encrypts data objects and generates meta data that enables fast query processing. Cumulus4j handles encryption, decryption and query translation transparently and requires only minimal changes to the original application. We defined a novel security notion, Ind-ICP, which informally is fulfilled if relations between attribute values are hidden. We proved that Cumulus4j achieves this kind of security. Finally, we showed that the solution is practical by comparing it with an unsecured database in several benchmark scenarios.

Planned future improvements are optimizations of the index structure as well as additional annotations in order to allow for even faster queries execution.

8 Acknowledgements

We wish to thank Andy Jefferson and Jan Mortensen of NightLabs Consulting GmbH their support with the implementation of Cumulus4j and with the benchmarks. This work has been partially funded by the Federal Ministry of Education and Research, Germany (BMBF, Contract No. 01IS10037) and KASTEL (BMBF, Contract No. 16BY1172). The responsibility for the content of this article lies solely with the authors.

References

1. Cumulus4j. *http://www.cumulus4j.org/*.
2. Data Nucleus Access Plattform. *http://www.datanucleus.org/*.
3. Java Data Objects (JDO). *http://db.apache.org/jdo/*.
4. Java Persistence API (JPA). *http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html*.
5. PolePosition. *http://polepos.sourceforge.net*.
6. PolePosition Circuits. *http://polepos.sourceforge.net/circuits.html*.
7. G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu. Two can keep a secret: A distributed architecture for secure database services. *CIDR 2005*.
8. E. Damiani, S. D. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs, 2003.
9. S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Fragments and loose associations: respecting privacy in data publishing. *Proc. VLDB Endow.*, 3(1-2):1370–1381, Sept. 2010.
10. C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.
11. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229, New York, NY, USA, 1987. ACM.
12. H. Hacigümüs, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 216–227. ACM, 2002.
13. H. Hacigümüs, B. Iyer, and S. Mehrotra. Providing database as a service. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, page 29, Washington, DC, USA, 2002. IEEE Computer Society.
14. B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 720–731. VLDB Endowment, 2004.
15. M. Huber, C. Henrich, J. Müller-Quade, and C. Kempka. Towards secure cloud computing through a separation of duties. In *Informatik 2011*, 2011.
16. M. Kantarcioglu and C. Clifton. Security issues in querying encrypted data. Technical report, 2004.
17. D. Kifer and A. Machanavajjhala. No free lunch in data privacy. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 193–204, New York, NY, USA, 2011. ACM.
18. A. Nergiz and C. Clifton. Query processing in private data outsourcing using anonymization. In Y. Li, editor, *Data and Applications Security and Privacy XXV*, volume 6818 of *Lecture Notes in Computer Science*, pages 138–153. Springer, Berlin / Heidelberg, 2011.
19. R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

20. A. T. Soodejani, M. A. Hadavi, and R. Jalili. k-anonymity-based horizontal fragmentation to preserve privacy in data outsourcing. In N. Cuppens-Boulahia, F. Cuppens, and J. Garca-Alfaro, editors, *DBSec*, volume 7371 of *Lecture Notes in Computer Science*, pages 263–273. Springer, 2012.
21. X. Xiao and Y. Tao. Anatomy: simple and effective privacy preservation. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 139–150. VLDB Endowment, 2006.

A Benchmark Results

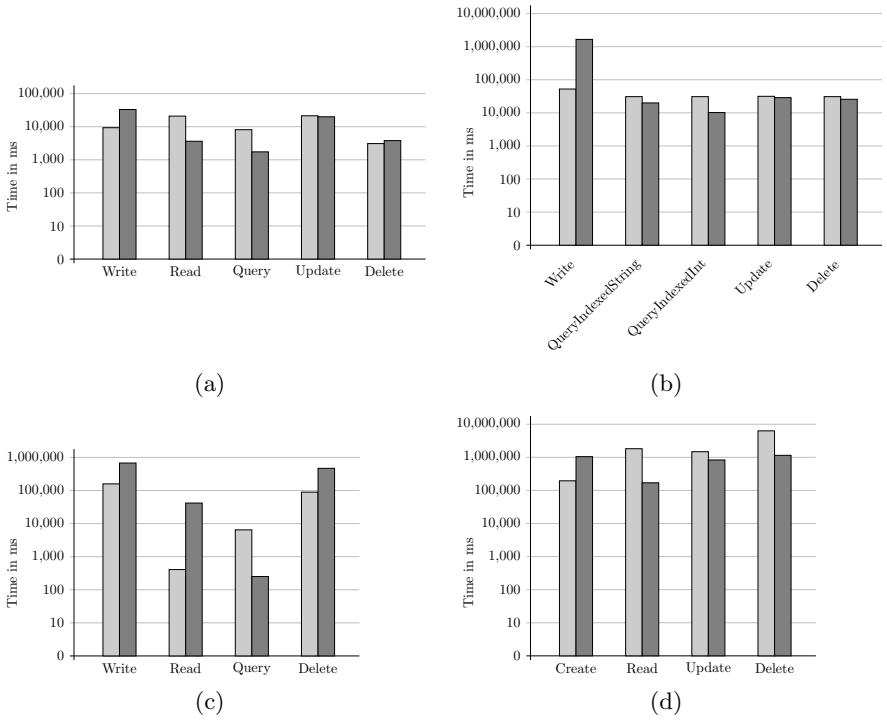


Fig. 11. Benchmark of circuit (a) *Complex*, (b) *Flatobject*, (c) *Inheritancehierarchy*, and (d) *Nestedlists* of JDO/DataNucleus without (light gray) and with the Cumulus4j plug-in (dark gray).