Implementation of a Storage Mechanism for Untrusted DBMSs

Ernesto Damiani
Università di Milano
26013 Crema - Italy
damiani@dti.unimi.it

Sabrina De Capitani di Vimercati
Università di Milano
26013 Crema - Italy
decapita@dti.unimi.it

Mario Finetti
mario.finetti@flashnet.it

Stefano Paraboschi
Università di Bergamo
24044 Dalmine - Italy
parabosc@unibg.it

Pierangela Samarati
Università di Milano
26013 Crema - Italy
samarati@dti.unimi.it

Sushil Jajodia
George Mason University
Fairfax, VA 22030-4444
jajodia@gmu.edu

Abstract

Several architectures have been recently proposed that store relational data in encrypted form on untrusted relational databases. Such architectures permit the creation of novel Internet services and also offer an opportunity for a better construction of ASP solutions. Environments where there are limited resources that do not permit an efficient management of databases or where it is critical to offer a robust Internet access to private data may all benefit from the above architectures. In this paper we analyze the impact that this architecture has on the typical services of a database. The analysis is based on the experience gained in the construction of a prototype of a complete architecture for the management of encrypted databases. Specifically, we illustrate the impact on query translation and optimization, and the main components of the software architecture of the prototype.

1. Introduction

Recently, a new *deployment infrastructure* for e-services has been taking shape where Application Service Providers (ASP) host and manage complete e-business and e-government applications on behalf of enterprises and government agencies. This infrastructure requires new security techniques, especially designed for large-scale data hosting. Until recently, many organizations neglected static data protection, believing that database systems located inside their own perimeter defences were safe from attacks. Now, they are increasingly aware that databases hold a high concentration of critical information, and threats coming from insiders are considerably higher than the ones posed by remote users. In fact, while data hosting has many advantages, it may increase security risks, inasmuch critical data are deployed at an ASP remote location and are managed by ASP personnel, both of which are not under the full control of the data owner. If badly secured application data can be compromised even by unskilled hackers, hosting of unprotected databases is even worse because it allows untrusted ASP staff to access critical information such as payroll or sales data. The increasing success of application hosting is gradually shifting companies' and researchers' attention from well-understood encryption techniques for protecting data during network transit and within applications to those aimed at protecting static information residing in databases. If database service has to become a commodity widely available on the network, robust protection mechanisms, typically based on encryption, have to be designed.

Database encryption [4] consists in encrypting data stored within a database in order to protect it from being compromised. If the information managed by a hosted database is encrypted, a hacker who breaks into the ASP network will not be able to access it; furthermore (and perhaps more importantly) an ASP employee (e.g., a database operator) who either intentionally or accidentally displays critical data will not be able to understand them. Also, it is interesting to remark that while the general problem of encrypting database content is certainly not new, its statement given above is very different from the one proposed in the last century, where encryption was only performed at the physical level, while the DBMS server (and its operator) was trusted. Traditional approaches to protecting databases utilized software-based encryption, though recent research is highlighting the merits of hardware-based approaches for ASP architectures. The main effort of current research in this area is the design of a transparent mechanism that makes it possible to

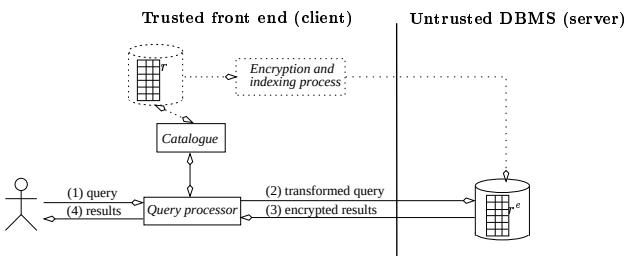


Figure 1. Scenario

use an encrypted database as a traditional one, with a guarantee on data confidentiality and with a loss in performance as little as possible. Available results [11] show that there is a clear trade-off between performance and confidentiality, but careful database design can combine good performance and a high level of data protection. In our previous work [3] we described a complete approach to database encryption and evaluated its robustness with respect to internal inference attacks, where an insider can compare the whole encrypted database with cleartext query results. In this paper we follow a different, though related, line of research focusing on implementation guidelines aimed at providing efficient and transparent database services by means of query translation and optimization techniques especially designed for an encrypted database setting. Our guidelines include functional decomposition of the query execution environment in a set of reusable components. The paper is organized as follows. Section 2 describes the basic concepts and the scenario we consider. Section 3 illustrates our techniques for computing the indexing information that allows for executing queries on encrypted data. Section 4 describes the architecture of the query processor. Section 5 illustrates the main classes of our implementation. Finally, Section 6 presents our conclusions.

2. Basic concepts and scenario

We briefly introduce preliminary definitions and concepts of relational model and encryption.

We assume standard notions from the relational database model. A *relation scheme* R is a finite set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of *attributes*. A *tuple* t is a mapping from a finite set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of attributes to a (possibly infinite) set $\mathcal{V}$ of *values*, where $t[A]$ denotes the mapping for attribute A in t. A *relation* r over relation scheme R is a finite set of tuples over R. A database $\mathcal{B}$ over a set of relation schemes $\{R_1, \ldots, R_n\}$ is a set of relations $\{r_1, \ldots, r_n\}$, where each r_i is a relation over R_i.

Figure 1 illustrates the basic scenario we consider. At the trusted front end we are given a relational database $\mathcal{B}$ to be outsourced to an external untrusted DBMS. The goal of our approach is to produce an encrypted database in such a way that *the untrusted DBMS can efficiently execute queries on the encrypted data*. A first approach for solving this problem was presented in [6]. The idea is to associate with each encrypted relation a set of *index attributes* that the untrusted DBMS can use to retrieve relevant data to be returned in response to a query. There is an index for each attribute A_i in the original relation r on which conditions needs to be evaluated in the execution of queries. Each relation r is then mapped into a relation r^e whose attributes are the encrypted tuple and the corresponding indexes.

In this scenario, illustrated in Figure 1, each query (1) is mapped onto a query on the encrypted data (2) that is executed at the untrusted DBMS. The result of this query is a set of encrypted tuples (3) that are then decrypted by the trusted front end (4). As we will see in Section 3, the front end may need to execute an additional query to discard spurious tuples that do not belong to the result set.

The application of encryption to a relational database $\mathcal{B}$ can be performed at different granularity levels (e.g., relation, tuple, or single elements). We consider encryption at the tuple level because it provides a good trade-off between query execution efficiency and front end workload [7]. We then assume that each relation r over schema $R = \{A_1, \ldots, A_n\}$ is stored at the untrusted DBMS as a relation r^e over schema $R^e = \{T^e, I_1, \ldots, I_m\}$, $m \leq n$, where T^e is an attribute for the encrypted tuple, and $I_i, i = 1, \ldots, m$, are attributes for the indexes. Therefore, each tuple $t[A_1, \ldots, A_n] \in r$ is mapped onto a tuple $t'[T^e, I_1, \ldots, I_m] \in r^e$, where $t'[T^e] = E_k(t)$ with $E_k()$ an invertible encryption function using k as key, and $t'[I_i] = f(t[A_j]), i = 1, \ldots, m$, for some $j = 1, \ldots, n$, with f an indexing function.

The major challenges in this scenario is how to compute and represent indexing information as well as how to efficiently use it in the execution of queries. These issues will be discussed in the remainder of the paper.

3. Data organization and indexing

Two different techniques can be used to compute the indexing information: *i)* encrypted attribute value, and *ii)* hash value. As we will see in the following, such indexes support an efficient execution of equality queries (i.e., queries in which the WHERE clause includes only equality conditions) but they do not support well interval-based queries. To avoid this problem, the indexing information is enriched with *auxiliary B+-trees* that are traditionally used in the relational DBMSs for supporting interval-based queries [1]. In the remainder of this paper, we refer our examples to

BILL

ID	Patient	Charge
p1	Ada	$10
p2	Burt	$50
p3	Burt	$20
p4	Clio	$50
p5	Dido	$30
p6	Elvira	$50

PATIENT

Name	Street	City	ZIP
Ada	Florence St.	Washington	98001
Burt	Main Av.	Miami	56345
Clio	University rd.	Miami	56345
Dido	Salt St.	Miami	56345
Elvira	Dance rd.	Philadelphia	12345
Fred	Salt St.	Washington	98001

Figure 2. An example of plaintext relations

BILL1^e

T^e	I_I	I_P	I_C
SeCS0U/7ZIY.A	α	η	μ
uRnZBBQcrRPGY	β	θ	ν
/WKu5y8laqK82	γ	θ	ρ
jzKzVi0D1as8E	δ	ι	ν
AXYaqohgyVObU	ϵ	κ	ξ
0CGZJvVV.zM4U	ζ	λ	ν

(a)

BILL2^e

T^e	I_I	I_P	I_C
SeCS0U/7ZIY.A	α	η	λ
uRnZBBQcrRPGY	β	θ	μ
/WKu5y8laqK82	γ	θ	λ
jzKzVi0D1as8E	δ	η	μ
AXYaqohgyVObU	ϵ	ι	λ
0CGZJvVV.zM4U	ζ	ι	μ

(b)

Figure 3. Encrypted relations corresponding to relation BILL in Figure 2 with indexes by encryption (a) and indexes by hashing (b)

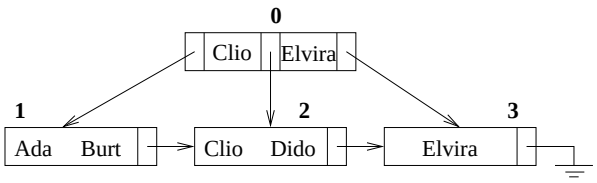
the outsourcing of an hospital billing service and the corresponding simple relations BILL and PATIENT in Figure 2.

3.1. Index by encryption

A simple approach for computing indexes consists in using an invertible encryption function $E_k()$ as indexing function. Formally, each tuple $t \in r$ is mapped to a tuple $t' \in r^e$ where $t'[I_i] = E_k(t[A_j]), i = 1, \ldots, m$ and for some $j = 1, \ldots, n$. As an example, the relation in Figure 2 would be mapped into relation BILL1^e (see Figure 3(a)). Here, for simplicity, we assume that there is an index for each attribute in the plaintext relation and the output of the indexing function is represented by Greek letters. Also, the encrypted relations and the index attributes have meaningful names to increase readability. Obviously, in a real application such names must be obfuscated. In this case, a query on a plaintext relation has to be transformed into a query on the corresponding encrypted relation by simply applying the encryption function on each value specified in the original query. This technique is simple and has the advantage of preserving the distinguishability between values. That is, the encryption function used to compute the index values is such that two different plaintext values are always transformed into two different encrypted values. In this way, the set of tuples returned by the transformed query, after decryption, corresponds to the set of tuples that the user could obtain by applying the original query to the plaintext relation. For instance, with reference to our example, query "SELECT patient FROM BILL WHERE charge=$10" is transformed in "SELECT T^e FROM BILL1^e WHERE $I_C = \mu$" which returns the first encrypted tuple. Before presenting the result to the user, the query processor has to decrypt such a tuple and perform a projection on `patient` attribute. The drawback of the indexing technique based on the invertible encryption function is that it is often possible to guess the correspondence between plaintext and encrypted values based on *frequency analysis*, that is, by comparing the distributions of the plaintext values in the plaintext relations with the corresponding distributions of the encrypted values. For instance, the correspondence $\nu = \$50$ for attribute `charge` can be easily pointed out as $50 (and ν) are the only values with three occurrences.

3.2. Index by hashing

An alternative approach to indexing consists in using a *secure hash function* h (i.e., a one-way function that takes some plaintex input and transforms it into a fixed-length encrypted output). Formally, each tuple $t \in r$ is mapped to a tuple $t' \in r^e$ where $t'[I_i] = h(t[A_j]), i = 1, \ldots, m$ and for some $j = 1, \ldots, n$. As an example, the BILL relation in Figure 2 would be mapped into relation BILL2^e (see Figure 3(b)). Given an input x, the secure hash function h computes a value $y = h(x)$ which has a smaller number of bits than x. This implies the possibility of collisions, that is, different plaintex values can be transformed by the function to the same index. Another important property of a secure hash function is that it uniformly covers its range (i.e., the output probabilities from the hash function are uniform). The combination of these two properties has the effect of flattening the attribute values' distribution and dispersing the attribute's values in $|B|$ *buckets*, where B is the codomain of the hash function, thus making frequency-based attacks ineffective. As an example, consider the encrypted relation BILL2^e in Figure 3(b). Here, patients' names have been distributed in three buckets, namely η, θ, and ι, and the values of attribute `Charge` have been distributed in two buckets, namely λ and μ. Like for the previous case, a query on a plaintext relation is transformed into a query on the corresponding encrypted relation by applying the hash function on each value specified in the original query. However, due to index collisions the transformed query may return spurious tuples, that is, tuples that do not belong to the result set of the original query. For instance, consider again query "SELECT patient FROM BILL WHERE charge=$10" transformed in "SELECT T^e FROM BILL2^e WHERE $I_C = \lambda$" which returns the first, third, and fifth encrypted tuple. The query processor has to: decrypt the returned tuples, apply the original condition (charge=$10) to discard possibly spurious tuples (the third and fifth tuple), and finally perform a projection on `patient` attribute.



(a)

B+table

ID	Node
0	(1,Clio,2,Elvira,3)
1	(Ada,Burt,2)
2	(Clio,Dido,3)
3	(Elvira,_,-1)

B+Enctable

ID	C
0	LxgEhc1bjrTJY
1	nfkwK5ZS85jus
2	yHxvBbn8.f8Tk
3	LhJLsrMkmlNhQ

(b)

Figure 4. An example of B+-tree with fan out 3 (a) and the corresponding plaintex and encrypted table used to store the B+tree (b)

These two index techniques allow an efficient evaluation of equality conditions at the untrusted DBMS. Therefore, if the same indexing function is used to compute indexes of different relations, they also support well equijoin. The drawback is that they do not support well interval-based queries. For instance, consider query "SELECT patient FROM BILL WHERE charge$\geq$ \$10 AND charge$\leq$ \$30" which should return Ada, Burt, and Dido. An efficient support for this kind of queries would be possible by using *order-preserving function* (imposing $h(t_i[A]) < h(t_j[A])$ whenever $t_i[A] < t_j[A]$). However, this solution is not viable as comparing the ordered sequences of index and plaintext values would lead an easy reconstruct the correspondence. A trivial approach consists in transforming a condition over an interval into a disjunction of equality conditions, one for each value in the interval. However, this solution is not efficient and is not always applicable. Our solution is based on enriching indexing information with B+-trees as discussed in the following.

3.3. Auxiliary B+-tree

We assume the reader is familiar with traditional B+-trees and only give a concise description of them to introduce the terms that are useful to describe their adoption in our context. A B+-tree with fan out n is a tree where every node can store up to $n-1$ search key values and n pointers, and, except for the root and leaf nodes, has at least $\lceil n/2 \rceil$ children. Given an internal node storing p key values ($p \leq n-1$), $k_1, \ldots, k_p$, each k_i is followed by a pointer a_i; k_1 is preceded by a pointer a_0. Pointer a_0 addresses the subtree that contains keys with values less than k_1, a_p addresses the subtree that contains keys with values greater than or equal to k_p, and each a_i addresses the subtree that contains keys with values included in the interval $[k_i, k_{i+1})$. Leaf nodes are linked by a chain that allows the efficient execution of interval queries. Figure 4(a) illustrates an example of B+-tree built on attribute `patient` of the BILL relation in Figure 2. To access a tuple with key value k', a search of value k' in the root node of the B+-tree is made. The tree is then traversed by using the following scheme: if $k' < k_1$, pointer a_0 is chosen; if $k' \geq k_p$, pointer a_p is chosen, otherwise if $k_i \leq k' < k_{i+1}$, pointer a_i is chosen. The process continues until a leaf node has been examined. If k' is not found in the leaf node then the desired tuple is not in the table.

As mentioned in the previous section, the indexing function is not order-preserving and therefore B+-trees are useful only if they are built on the plaintext values of the attributes. This means that they contain sensitive information that needs to be protected. Therefore, in our context, B+-trees are built by the trusted front end and must be encrypted (at the trusted front end) for its storing at the untrusted server. We consider encryption at the level of whole node to obfuscate the order relationship between index values. More precisely, a B+-tree is stored as a relation *b* over schema B+Enctable$= \{ID, C\}$, where ID is the identifier of the node, and C is the encrypted node. Figure 4(b) illustrates the plaintext and encrypted B+-tree table for the B+-tree in Figure 4(a). This way, B+-trees can be traversed only by the trusted front end. Therefore, to retrieve the tuples in the encrypted relation that are characterized by a given value k', it is necessary first to retrieve the root (tuple with ID 0) from the encrypted table. The content of the node is decrypted and k' is compared with the values present in the node. According to the search process described above, the trusted front end identifies the ID of the next node that has to be checked and communicates such a ID to the untrusted DBMS that in turn retrieves and sends it to the front end. This process is repeated until a leaf is retrieved. For instance, to retrieve the patient's names starting with a letter in the interval [C,E], the trusted front end has to generate three queries in order to access nodes 0, 2, and 3.

4. Query processor

4.1. Architecture

An interpretation of the architecture presented in the previous section can assume the presence of two data representations at different levels. At the lower level there is the encrypted representation on a remote and untrusted database. At the higher level, the virtual cleartext database can be materialized on the client after a transfer from the encrypted database.

The components needed for the management of queries in this environment are (see Figure 5):

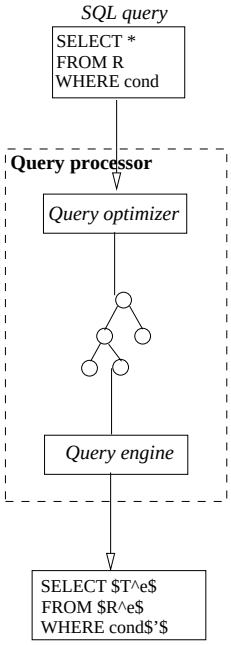
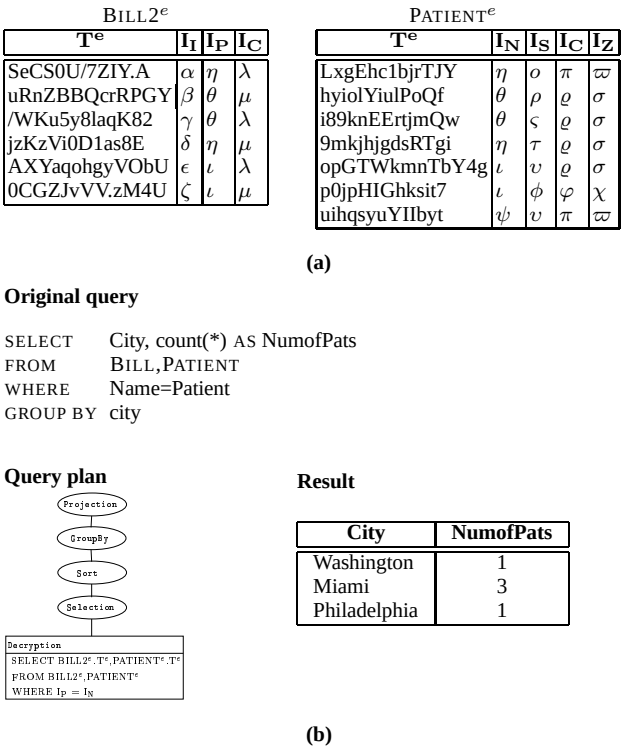


Figure 5. Architecture of the query processor



T^e	I_I	I_P	I_C
SeCS0U/7ZIY.A	α	η	λ
uRnZBBQcrRPGY	β	θ	μ
/WKu5y8laqK82	γ	θ	λ
jzKzVi0D1as8E	δ	η	μ
AXYaqohgyVObU	ϵ	ι	λ
0CGZJvVV.zM4U	ζ	ι	μ

T^e	I_N	I_S	I_C	I_Z
LxgEhc1bjrTJY	η	o	π	ϖ
hyiolYiulPoQf	θ	ρ	ϱ	σ
i89knEErtjmQw	θ	ς	ϱ	σ
9mkjhjgdsRTgi	η	τ	ϱ	σ
opGTWkmnTbY4g	ι	υ	ϱ	σ
p0jpHIGhksit7	ι	ϕ	φ	χ
uihqsyuYIIbyt	ψ	υ	π	ϖ

Figure 6. An example of query execution

- a *query engine* in the client, able to invoke queries on the remote untrusted database and to apply relational operators on the results of the queries;
- a *query optimizer*, able to produce an efficient strategy for query execution using the operators and the services offered by the query engine.

Every query is expressed by the application as if expressed on the clear-text database. The query optimizer analyzes the query and produces an execution plan that splits the query computation into many steps organized in a tree. Leaves of the tree represent accesses to the encrypted database, typically using a network access and expressing the query in SQL. All the other tree nodes represent a local query computation, which occurs after the data have been retrieved and decrypted, using traditional techniques of relational query engines. We observe that since the leaves of the tree, that represent the access to the remote database, are able to execute on the remote database arbitrary SQL queries, the structure of the tree will be simpler in this context than for comparable queries on a regular database. In our experience, most of the times queries can be represented by a left-deep pipeline, which facilitates the execution of the query in the engine.

The query on the clear-text database is expressed in SQL. We are not planning to support in our prototype the full SQL syntax. Our first aim is the support of selections with boolean expressions of comparison predicates, projections, arbitrary joins, `group by` on subsets of attributes and selections on the result of aggregate functions (`having` clause). We are deferring the implementation of queries using the binary set operators `union`, `intersect` and `minus` and of nested queries or other complex SQL structures. The prototype should demonstrate the feasibility of the approach, and a more robust and complete implementation of these services should start from an existing DBMS, where the possibility to use encryption for storage on an untrusted site should be introduced as an option of physical design.

The query engine offers a number of relational operators, each one accepting as input one or two relations and a parameter specifying the transformation to apply. We have implemented operators *selection*, *projection*, *sort*, *join*, and *group by*. As an example, consider the relations in Figure 2, and their encrypted version in Figure 6(a), where the indexing information has been built by using the same hashing function.[1] A user wants to *find, for each city, the number of patients who have to pay a bill*. The original SQL query is transformed in a tree as illustrated in Figure 6(b). The leaf of this tree is a SQL query on the encrypted relations that is executed at the untrusted DBMS. Due to indexing colli-

1 Note that for the readers convenience, we reproduce relation BILL2^e of Figure 3(b).

sions, the result of this query includes spurious tuples that the client discards, after decryption, via a *selection* operation whose the applied predicate is the original condition 'name=patient'. Subsequently, tuples are ordered according to the city (*sort* operation) and then they are grouped (*group by* operation) as specified in the original SQL query. Finally, a *projection* operation is applied.

An important difference with respect to traditional DBMS engines is the fact that in this context performance is not dominated by disk access costs. Disks are block-i/o devices and an adequate approximation of query cost is the number of block transfers required. In our context, I/Os on the remote database certainly are dominated by disk access costs, but it is often reasonable to assume that the most expensive resource is the network bandwidth and the costs of encryption/decryption. This moves the emphasis from the design and choice of operators that require a limited number of disk I/Os to the design and choice of operators that require limited computational and network resources. A goal of the query optimizer is indeed to transfer to the client only the portion of the database that is of interest for a specific query. To reduce the computational effort, we devised a lazy decryption approach that exploits the physical structure of the data to decrypt only the portions that are actually needed in decrypted form; we present this technique in Section 4.2. Another critical aspect is the reduction in the size of the intermediate results. Particularly critical is the identification of the physical design strategy that permits to reduce the complexity of join operators. We decided to disregard the costs of query execution on the remote DBMS; even if it is not correct in all contexts, it is a convenient hypothesis, since it permits to reduce the complexity of the problem and remove a component that is less critical than the minimization of network traffic and computational costs in the client; the minimization of these parameters produces queries that do not require an expensive computation on the untrusted DBMS. This separation allows us to use traditional techniques for the physical design of the untrusted DBMS.

Another important trade-off is the one between performance and protection of data confidentiality, particularly against inference attacks. We are in the process of building a model that characterizes the exposure to inference attacks of an encrypted database, depending on several assumptions on attacker's knowledge of the clear-text database. We are not formally exploring this issue in the paper.

4.2. Lazy decryption

Encryption is realized using a block encryption algorithm. The current prototype makes the choice of the algorithm a simple configuration parameter, which can be easily customized; the default choice in our experiments was AES. We observe that a stream cipher would produce a greater level of protection in the tuple, as tuples that would exhibit a different prologue but the same intermediate content would exhibit the same content in the encrypted representation, whereas they would exhibit different representations in the stream-cypher. Despite this, we opted for a block cipher algorithm because we want to be able to avoid the decryption of parts of the tuple that are not needed. To remove the disadvantages of the block cipher, making it difficult to identify identical subparts, we offer the option of introducing, for each block, a random component that creates different encryptions for all the encryptions of the same content.

The technique that permits to avoid the decryption of the tuple exploits the structure of the tuple. Each tuple is composed by a list of values, each corresponding with an attribute of a given domain. Since the space required for the representation of an attribute value is usually not constant (because the attribute is associated with a variable length domain or because the null value is admitted), it is not possible to simply identify the positions in the tuple representation where each attribute must appear. It is then necessary to introduce auxiliary information that permits to identify for each tuple the positions where the different attribute values are represented. This problem also occurs in traditional databases, where each block containing tuples is separated into a part containing a page dictionary and a part containing the tuples; the page dictionary is separated into a sequence of tuple dictionaries, one for each tuple, that describe where in the block each component of the tuple is stored. Here the situation is slightly different: there is no need to group the representation of distinct tuples, but a tuple dictionary should be used and it should occupy the first bytes in the representation of the tuple. The presence of the tuple dictionary in the first bytes allows the query engine to decrypt only the portion of the tuple that is really needed for the query in execution. For instance, if there is the need to access a tuple with a great number of attributes, and the query only needs the first and the last of the attributes, the lazy approach will decrypt the first few bytes in order to identify the starting positions of the attributes and it then will decrypt only the blocks that may produce information that is needed to answer the query. In Section 5.3 we present the class `PackedTuple` of our prototype that is responsible for the execution of this mechanism.

5. Implementation of the prototype

To demonstrate the applicability and the advantages of the solution, we implemented a prototype in Java. We briefly present the main classes of the prototype, to illustrate how the principles previously presented in the

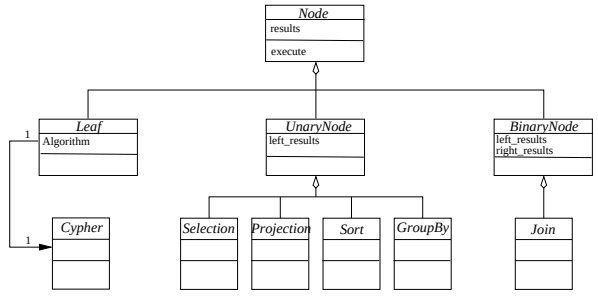


Figure 7. Class diagram of `Node`

paper are concretely realized.

5.1. Class `Node` and its subclasses

The most important classes are the ones that represent the operators of the query engine. A class `Node` represents the generic operator that produces a relation as result. The class diagram is reported in Figure 7. For conciseness, here we do not give a detailed description of such a class, rather we provide its functional decomposition. The term *node* is used because a generic query is conventionally translated into a tree that describes the execution plan, where each node of the tree represents a distinct application of an operator. Class `Node` presents a method `execute`, that starts the operator evaluation, and a property `results` that contains the results of the node computation. Class `Node` is specialized into three classes, that correspond to leaf nodes, unary operators and binary operators, respectively.

Class `Leaf` is responsible for extracting the data from the remote untrusted DBMS. We assume that the remote database is accessible using a JDBC connection. The main method introduced by the class is a constructor that requires the specification of all the properties that characterize each instance of `Leaf`. These properties are the parts of the SQL query that has to be executed remotely, the description of the attributes that have to be kept in the result, and the names and types of all the attributes in the result. Although our engine has a specialized operator for projection, class `Leaf` is able to realize a projection immediately; the justification is that this projection can be done directly on the encrypted data representation. The mechanism has been presented in Section 4.2 and its implementation is realized by class `PackedTuple`. Another important property of class `Leaf` is the encryption algorithm used and the key to use for decryption.

The encryption algorithm is specified by an instance of `Cypher` class, part of the Java Cryptography Extension (JCE [8]). Class `Leaf` is in fact the only class responsible for the decryption of database content. The result produced by `Leaf` is available to the client and is in clear-text.

Class `UnaryNode` represents a generic unary operator, which transforms a relation into another relation. Class `UnaryNode` adds the property `left_results`, which represents the relational input of the operator. Class `UnaryNode` is further specialized into several classes, each one associated with a relational operator, like `Selection`, `Projection`, `Sort` and `GroupBy`. The constructor of an instance of node `Selection` receives as parameter the predicate that characterizes the selection. Node `Projection` is characterized by the list of attributes that have to be kept in the result, among those present in the input. Node `Sort` is characterized by the ordered list of attributes on which the tuples have to be ordered; `Sort` is also responsible for duplicate elimination. Finally, node `GroupBy` is characterized by the attributes that have to be used for the grouping, and by the list of aggregate functions that have to be evaluated for each group. As it happens in many relational database engines, we assume that the grouping has as input a relation ordered on the list of attributes to be used for the grouping; this ordering may be a property derived by the physical storage structures used to retrieve the data or may be the result of a previous `Sort` operation.

Class `BinaryNode` is a specialization of `Node` and represents a generic binary operator which, based on the content of two relations, produces a relation as result. The class adds the two properties `left_results` and `right_results`, which represent the two operands. Class `BinaryNode` is further specialized into class `Join`. Other specializations that will be implemented are `Union`, `Intersection`, `Minus`, and any other binary operator that may be of interest. We observe that in this particular context operator `Join` is used less frequently than in traditional databases, as most join evaluations are done on the remote DBMS.

The tree is built constructing all the nodes and making the result of a node the input of the node above it in the tree. With this sharing of objects, what is produced by a node is immediately available as input on the node following it in the computation. As it is traditional for relational operators, nodes are distinguished into those operating a local computation, where the computation can be realized separately on every tuple (e.g., `Selection`), and those operating a global computation, that requires to have available all the input tuples before returning any output (e.g., `Sort`). Intermediate between the two is the behavior of the `GroupBy` operator, which returns a tuple only after it has finished the analysis of all the tuples of a group. When the tree is completed, method `execute` is invoked progressively from the leaves to the root. In the current implementation we associate a thread with each node and use the synchronization mechanisms of Java on the relation shared be-

tween a node and its predecessor to guarantee a regular flow of data. There are several advantages of a realization with a thread for each node. The first advantage is a greater simplicity of the code, since the switch from the computation of a node to that of another node is realized transparently by the thread scheduler, with no need to introduce specialized mechanisms. Also, when all the nodes in the tree are local, it is possible to create a flow of tuples from the leaves to the root, which minimizes the memory needed for the computation of the query.

5.2. Classes `Results` and `ResultRow`

To represent relations within the relational engine we created a class called `Results`. The class is similar to class `ResultSet` of JDBC, with a number of services to set and to query the structure and the content of the relation. We decided to implement from scratch this class, instead of directly reusing class `ResultSet` of JDBC, because in this way we had greater control on the behavior of the engine and were able to build a more compact structure. The class contains a set of tuples, represented by class `ResultRow`. Tuples have a flexible structure and can be composed of an arbitrary list of attributes of types extracted from those available in SQL-92. An instance of `Results` describes the structure of all the instances of `ResultRow` contained in it.

5.3. Class `PackedTuple`

Class `PackedTuple` is responsible of the translation from the encrypted format used to store them in the untrusted DBMS to the clear-text value that is used on the client. Tuples are encrypted using a block-cipher algorithm with a tuple structure that presents a *tuple dictionary* followed by the representation of the values of all the attributes. Class `PackedTuple` keeps two instances of class `ByteArray` (specialization of standard Java class `ByteBuffer`): one called `encrypted` stores the tuple as it is retrieved by the remote database, the other called `decrypted` stores all the blocks in the tuple that have already been decrypted. Every time there is the need to access a part of the tuple, the method verifies if the block is already available in `decrypted`; if the block is not present, the corresponding block in `encrypted` is passed to the decryption algorithm (passed from the instance of class `Leaf` containing `PackedTuple`) and is inserted into `decrypted`. The class offers all the methods that understand the structure of the tuple dictionary and that permit to access the separate components of the tuple.

6. Conclusions

Efficient implementation of encrypted database hosting is of paramount importance, and even more so since the Gramm-Leach-Bliley Act (GBLA) of 1999 (which went into effect in late 2002) has shown that U.S. Congress intends to make both U.S. and foreign businesses financially liable for unauthorized disclosure of customer data. On the other hand, it is widely acknowledged that Web services success will depend to a large extent on maximizing the use of customer-related information already available in databases. Dishonest or disgruntled employees are a growing problem that companies must face, and hosting might mean a substantial increase of insider security threats, since it brings untrusted ASP personnel into the picture. In this paper, we dealt with some specific requirements for the implementation of encrypted database technology in a Web-based setting, focusing on modular design of components that allow for an acceptable tradeoff between performance and security. Our approach prevents the severe performance hits that may occur when the information in indexed fields is encrypted naively. Also, our guidelines guarantee clean and effective functional decomposition, because they ensure that database administrators having access rights to data does not mean that they have access rights to the keys used to encrypt them.

7. Acknowledgments

The work reported in this paper has been partially supported by the Italian MURST within the KIWI and MAPS projects.

References

[1] P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Database Systems: Concepts, Languages, and Architecture*. McGraw Hill, 1999.

[2] L. Bouganim and P. Pucheral. Chip-secured data access: Confidential data on untrusted servers. In *Proc. of the 28th International Conference on Very Large Data Bases*, pages 131–142, Hong Kong, China, August 2002.

[3] E. Damiani, S. D. C. di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational dbmss. In *Proc. of the 10th ACM Conference on Computer and Communications Security*, Washington, DC, USA, October 2003. to appear.

[4] G. Davida, D. Wells, and J. Kam. A database encryption system with subkeys. *ACM Transactions on Database Systems*, 6(2):312–328, June 1981.

[5] S. Ghandeharizadeh and D. DeWitt. A multiuser performance analysis of alternative declustering strategies. In *Proc. of the 6th Int. Conf. on Data Engineering*, 1990.

[6] H. Hacigümüs, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider

model. In *Proc. of the ACM SIGMOD'2002*, Madison, Wisconsin, USA, June 2002.

[7] H. Hacigümüs, B. Iyer, C. Li, and S. Mehrotra. Providing database as a service. In *Proc. of the 18th International Conference on Data Engineering*, San Jose, California, USA, February 2002.

[8] JavaTM cryptography extension (JCE). http://java.sun.com/products/jce/.

[9] C. Jensen. Cryptocache: a secure sharable file cache for roaming users. In *Proc. of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 73–78, Kolding, Denmark, September 2000.

[10] S. Klein, A. Bookstein, and S. Deerwester. Text retrieval systems on CD-ROM: compression and encryption considerations. *ACM Transactions on Information Systems*, 7(3), July 1989.

[11] Rsa security: Securing data at rest: Developing a database encryption strategy (white paper). http://www.rsasecurity.com/bsafe/whitepapers/DDES_WP_0702.pdf.

[12] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2000.

[13] J. Ward, M. O'Sullivan, T. Shahoumian, and J. Wilkes. Appia: Automatic storage area network fabric design. In *Proc. of the Conference on File and Storage Technologies (FAST 2002)*, Monterey, CA, January 2002.

[14] E. Yang, J. Xu, and K. Bennett. Private information retrieval in the presence of malicious failures. In *Proc. of the 26th Annual International Computer Software and Applications Conference*, Oxford, England, August 2002.